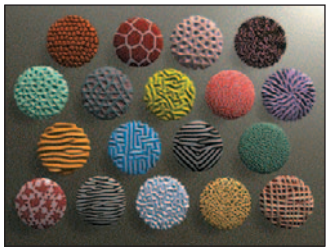
COMPUTER GRAPHICS

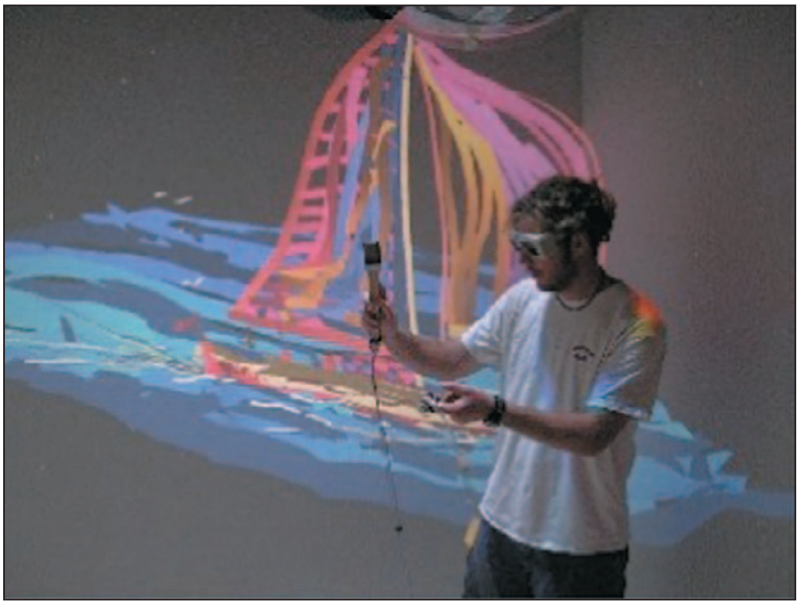
PRINCIPLES AND PRACTICE

THIRD EDITION

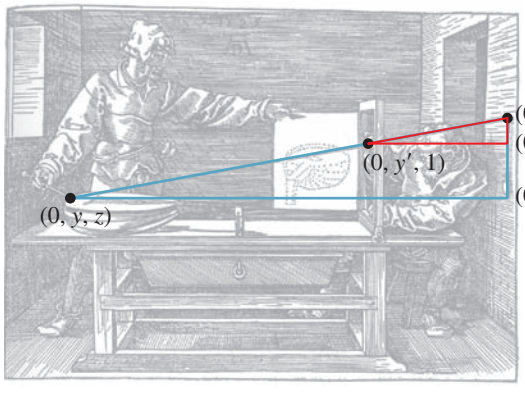
JOHN F. HUGHES · ANDRIES VAN DAM · MORGAN MCGUIRE
DAVID F. SKLAR · JAMES D. FOLEY · STEVEN K. FEINER · KURT AKELEY

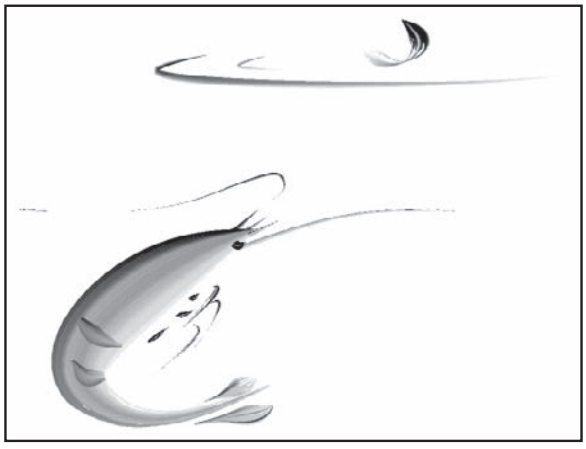


Top: Courtesy of Michael Kass, Pixar and Andrew Witkin, © 1991 ACM, Inc. Reprinted by permission. Bottom: Courtesy of Greg Turk, © 1991 ACM, Inc. Reprinted by permission.

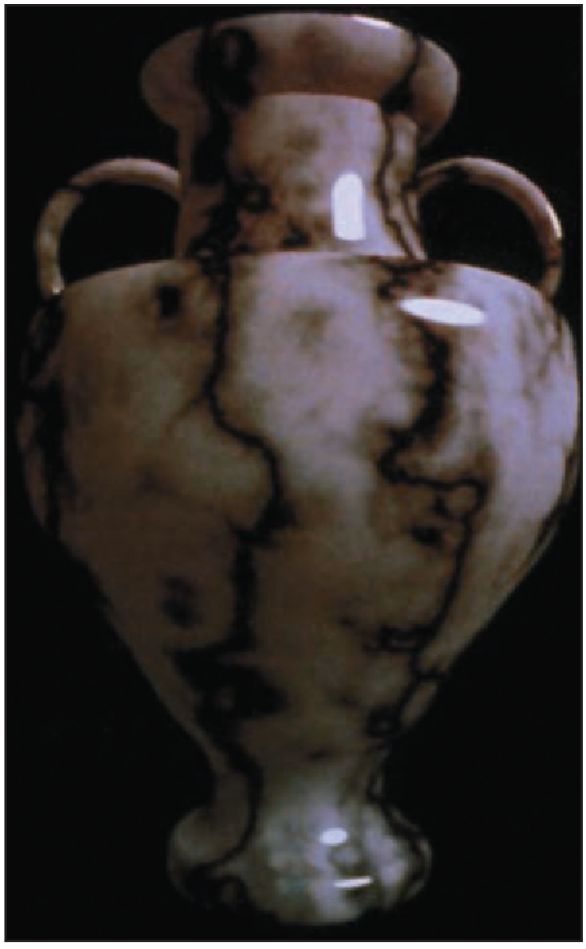


Courtesy of Daniel Keefe, University of Minnesota.

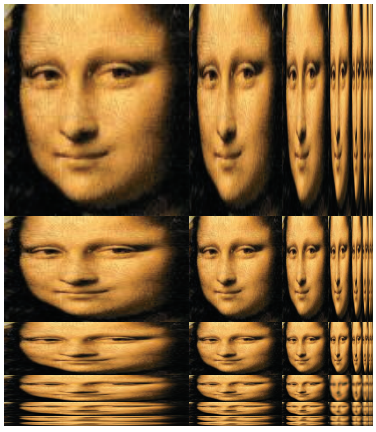




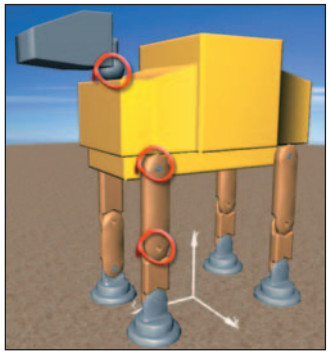
Courtesy of Steve Strassmann. © 1986 ACM, Inc. Reprinted by permission.



Courtesy of Ken Perlin, © 1985 ACM, Inc. Reprinted by permission.



Courtesy of Ramesh Raskar; © 2004 ACM, Inc. Reprinted by permission.



Courtesy of Stephen Marschner, © 2002 ACM, Inc. Reprinted by permission.

Courtesy of Seungyong Lee, © 2007 ACM, Inc. Reprinted by permission.

Computer Graphics

Third Edition

Computer Graphics

Principles and Practice

Third Edition

JOHN F. HUGHES
ANDRIES VAN DAM
MORGAN MCGUIRE
DAVID F. SKLAR
JAMES D. FOLEY
STEVEN K. FEINER
KURT AKELEY

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Hughes, John F., 1955–
Computer graphics : principles and practice / John F. Hughes, Andries van Dam, Morgan McGuire, David F. Sklar, James D. Foley, Steven K. Feiner, Kurt Akeley.—Third edition.
pages cm
Revised ed. of: Computer graphics / James D. Foley. . . [et al.].—2nd ed. – Reading, Mass. : Addison-Wesley, 1995.
Includes bibliographical references and index.
ISBN 978-0-321-39952-6 (hardcover : alk. paper)—ISBN 0-321-39952-8 (hardcover : alk. paper)
1. Computer graphics. I. Title.
T385.C5735 2014
006.6–dc23

2012045569

ISBN-13: 978-0-321-39952-6
ISBN-10: 0-321-39952-8
Text printed in the United States on recycled paper at RR Donnelley in Willard, Ohio.
First printing, July 2013

To my family, my teacher Rob Kirby, and my parents and Jim Arvo in memoriam.
—John F. Hughes

To my long-suffering wife, Debbie, who once again put up with never-ending work on "the book," and to my father, who was the real scientist in the family.
—Andries Van Dam

To Sarah, Sonya, Levi, and my parents for their constant support; and to my mentor Harold Stone for two decades of guidance through life in science.
—Morgan McGuire

To my parents in memoriam for their limitless sacrifices to give me the educational opportunities they never enjoyed; and to my dear wife Siew May for her unflinching forbearance with the hundreds of times I retreated to my "man cave" for Skype sessions with Andy.
—David Sklar

To Marylou, Heather, Jenn, my parents in memoriam, and all my teachers—especially Bert Herzog, who introduced me to the wonderful world of Computer Graphics!
—Jim Foley

To Michele, Maxwell, and Alex, and to my parents and teachers.
—Steve Feiner

To Pat Hanrahan, for his guidance and friendship.
—Kurt Akeley

This page intentionally left blank

Contents at a Glance

Contents ix

Preface xxxv

About the Authors xlv

1 Introduction 1

2 Introduction to 2D Graphics Using WPF 35

3 An Ancient Renderer Made Modern 61

4 A 2D Graphics Test Bed 81

5 An Introduction to Human Visual Perception 101

6 Introduction to Fixed-Function 3D Graphics and Hierarchical Modeling 117

7 Essential Mathematics and the Geometry of 2-Space and 3-Space 149

8 A Simple Way to Describe Shape in 2D and 3D 187

9 Functions on Meshes 201

10 Transformations in Two Dimensions 221

11 Transformations in Three Dimensions 263

12 A 2D and 3D Transformation Library for Graphics 287

13 Camera Specifications and Transformations 299

14 Standard Approximations and Representations 321

15 Ray Casting and Rasterization 387

16 Survey of Real-Time 3D Graphics Platforms 451

17 Image Representation and Manipulation 481

18 Images and Signal Processing 495

19 Enlarging and Shrinking Images 533

20 Textures and Texture Mapping 547
21 Interaction Techniques 567
22 Splines and Subdivision Curves 595
23 Splines and Subdivision Surfaces 607
24 Implicit Representations of Shape 615
25 Meshes 635
26 Light 669
27 Materials and Scattering 711
28 Color 745
29 Light Transport 783
30 Probability and Monte Carlo Integration 801
31 Computing Solutions to the Rendering Equation: Theoretical Approaches 825
32 Rendering in Practice 881
33 Shaders 927
34 Expressive Rendering 945
35 Motion 963
36 Visibility Determination 1023
37 Spatial Data Structures 1065
38 Modern Graphics Hardware 1103

List of Principles 1145
Bibliography 1149
Index 1183

Contents

Preface xxxv

About the Authors xlv

1 Introduction 1

Graphics is a broad field; to understand it, you need information from perception, physics, mathematics, and engineering. Building a graphics application entails user-interface work, some amount of modeling (i.e., making a representation of a shape), and rendering (the making of pictures of shapes). Rendering is often done via a "pipeline" of operations; one can use this pipeline without understanding every detail to make many useful programs. But if we want to render things accurately, we need to start from a physical understanding of light. Knowing just a few properties of light prepares us to make a first approximate renderer.

1.1 An Introduction to Computer Graphics 1
- 1.1.1 The World of Computer Graphics 4
- 1.1.2 Current and Future Application Areas 4
- 1.1.3 User-Interface Considerations 6

1.2 A Brief History 7

1.3 An Illuminating Example 9

1.4 Goals, Resources, and Appropriate Abstractions 10
- 1.4.1 Deep Understanding versus Common Practice 12

1.5 Some Numbers and Orders of Magnitude in Graphics 12
- 1.5.1 Light Energy and Photon Arrival Rates 12
- 1.5.2 Display Characteristics and Resolution of the Eye 13
- 1.5.3 Digital Camera Characteristics 13
- 1.5.4 Processing Demands of Complex Applications 14

1.6 The Graphics Pipeline 14
- 1.6.1 Texture Mapping and Approximation 15
- 1.6.2 The More Detailed Graphics Pipeline 16

1.7 Relationship of Graphics to Art, Design, and Perception 19

1.8 Basic Graphics Systems 20
- 1.8.1 Graphics Data 21

1.9 Polygon Drawing As a Black Box 23

1.10 Interaction in Graphics Systems 23

1.11 Different Kinds of Graphics Applications 24
1.12 Different Kinds of Graphics Packages 25
1.13 Building Blocks for Realistic Rendering: A Brief Overview 26
1.13.1 Light 26
1.13.2 Objects and Materials 27
1.13.3 Light Capture 29
1.13.4 Image Display 29
1.13.5 The Human Visual System 29
1.13.6 Mathematics 30
1.13.7 Integration and Sampling 31
1.14 Learning Computer Graphics 31

2 Introduction to 2D Graphics Using WPF 35

A graphics platform acts as the intermediary between the application and the underlying graphics hardware, providing a layer of abstraction to shield the programmer from the details of driving the graphics processor. As CPUs and graphics peripherals have increased in speed and memory capabilities, the feature sets of graphics platforms have evolved to harness new hardware features and to shoulder more of the application development burden. After a brief overview of the evolution of 2D platforms, we explore a modern package (Windows Presentation Foundation), showing how to construct an animated 2D scene by creating and manipulating a simple hierarchical model. WPF's declarative XML-based syntax, and the basic techniques of scene specification, will carry over to the presentation of WPF's 3D support in Chapter 6.

2.1 Introduction 35
2.2 Overview of the 2D Graphics Pipeline 36
2.3 The Evolution of 2D Graphics Platforms 37
2.3.1 From Integer to Floating-Point Coordinates 38
2.3.2 Immediate-Mode versus Retained-Mode Platforms 39
2.3.3 Procedural versus Declarative Specification 40
2.4 Specifying a 2D Scene Using WPF 41
2.4.1 The Structure of an XAML Application 41
2.4.2 Specifying the Scene via an Abstract Coordinate System 42
2.4.3 The Spectrum of Coordinate-System Choices 44
2.4.4 The WPF Canvas Coordinate System 45
2.4.5 Using Display Transformations 46
2.4.6 Creating and Using Modular Templates 49
2.5 Dynamics in 2D Graphics Using WPF 55
2.5.1 Dynamics via Declarative Animation 55
2.5.2 Dynamics via Procedural Code 58
2.6 Supporting a Variety of Form Factors 58
2.7 Discussion and Further Reading 59

3 An Ancient Renderer Made Modern 61

We describe a software implementation of an idea shown by Dürer. Doing so lets us create a perspective rendering of a cube, and introduces the notions of transforming meshes by transforming vertices, clipping, and multiple coordinate systems. We also encounter the need for visible surface determination and for lighting computations.

3.1 A Dürer Woodcut 61
3.2 Visibility 65
3.3 Implementation 65
3.3.1 Drawing 68
3.4 The Program 72
3.5 Limitations 75
3.6 Discussion and Further Reading 76
3.7 Exercises 78

4 A 2D Graphics Test Bed 81

We want you to rapidly test new ideas as you learn them. For most ideas in graphics, even 3D graphics, a simple 2D program suffices. We describe a test bed, a simple program that's easy to modify to experiment with new ideas, and show how it can be used to study corner cutting on polygons. A similar 3D program is available on the book's website.

4.1 Introduction 81
4.2 Details of the Test Bed 82
4.2.1 Using the 2D Test Bed 82
4.2.2 Corner Cutting 83
4.2.3 The Structure of a Test-Bed-Based Program 83
4.3 The C# Code 88
4.3.1 Coordinate Systems 90
4.3.2 WPF Data Dependencies 91
4.3.3 Event Handling 92
4.3.4 Other Geometric Objects 93
4.4 Animation 94
4.5 Interaction 95
4.6 An Application of the Test Bed 95
4.7 Discussion 98
4.8 Exercises 98

5 An Introduction to Human Visual Perception 101

The human visual system is the ultimate "consumer" of most imagery produced by graphics. As such, it provides design constraints and goals for graphics systems. We introduce the visual system and some of its characteristics, and relate them to engineering decisions in graphics.

The visual system is both tolerant of bad data (which is why the visual system can make sense of a child's stick-figure drawing), and at the same time remarkably sensitive. Understanding both aspects helps us better design graphics algorithms and systems. We discuss basic visual processing, constancy, and continuation, and how different kinds of visual cues help our brains form hypotheses about the world. We discuss primarily static perception of shape, leaving discussion of the perception of motion to Chapter 35, and of the perception of color to Chapter 28.

5.1 Introduction 101
5.2 The Visual System 103
5.3 The Eye 106
5.3.1 Gross Physiology of the Eye 106
5.3.2 Receptors in the Eye 107

5.4 Constancy and Its Influences 110
5.5 Continuation 111
5.6 Shadows 112
5.7 Discussion and Further Reading 113
5.8 Exercises 115

6 Introduction to Fixed-Function 3D Graphics and Hierarchical Modeling 117

The process of constructing a 3D scene to be rendered using the classic fixed-function graphics pipeline is composed of distinct steps such as specifying the geometry of components, applying surface materials to components, combining components to form complex objects, and placing lights and cameras. WPF provides an environment suitable for learning about and experimenting with this classic pipeline. We first present the essentials of 3D scene construction, and then further extend the discussion to introduce hierarchical modeling.

6.1 Introduction 117
6.1.1 The Design of WPF 3D 118
6.1.2 Approximating the Physics of the Interaction of Light with Objects 118
6.1.3 High-Level Overview of WPF 3D 119
6.2 Introducing Mesh and Lighting Specification 120
6.2.1 Planning the Scene 120
6.2.2 Producing More Realistic Lighting 124
6.2.3 "Lighting" versus "Shading" in Fixed-Function Rendering 127
6.3 Curved-Surface Representation and Rendering 128
6.3.1 Interpolated Shading (Gouraud) 128
6.3.2 Specifying Surfaces to Achieve Faceted and Smooth Effects 130
6.4 Surface Texture in WPF 130
6.4.1 Texturing via Tiling 132
6.4.2 Texturing via Stretching 132
6.5 The WPF Reflectance Model 133
6.5.1 Color Specification 133
6.5.2 Light Geometry 133
6.5.3 Reflectance 133
6.6 Hierarchical Modeling Using a Scene Graph 138
6.6.1 Motivation for Modular Modeling 138
6.6.2 Top-Down Design of Component Hierarchy 139
6.6.3 Bottom-Up Construction and Composition 140
6.6.4 Reuse of Components 144
6.7 Discussion 147

7 Essential Mathematics and the Geometry of 2-Space and 3-Space 149

We review basic facts about equations of lines and planes, areas, convexity, and parameterization. We discuss inside-outside testing for points in polygons. We describe barycentric coordinates, and present the notational conventions that are used throughout the book, including the notation for functions. We present a graphics-centric view of vectors, and introduce the notion of covectors.

7.1 Introduction 149
7.2 Notation 150
7.3 Sets 150
7.4 Functions 151
7.4.1 Inverse Tangent Functions 152
7.5 Coordinates 153
7.6 Operations on Coordinates 153
7.6.1 Vectors 155
7.6.2 How to Think About Vectors 156
7.6.3 Length of a Vector 157
7.6.4 Vector Operations 157
7.6.5 Matrix Multiplication 161
7.6.6 Other Kinds of Vectors 162
7.6.7 Implicit Lines 164
7.6.8 An Implicit Description of a Line in a Plane 164
7.6.9 What About $y = mx + b$? 165
7.7 Intersections of Lines 165
7.7.1 Parametric-Parametric Line Intersection 166
7.7.2 Parametric-Implicit Line Intersection 167
7.8 Intersections, More Generally 167
7.8.1 Ray-Plane Intersection 168
7.8.2 Ray-Sphere Intersection 170
7.9 Triangles 171
7.9.1 Barycentric Coordinates 172
7.9.2 Triangles in Space 173
7.9.3 Half-Planes and Triangles 174
7.10 Polygons 175
7.10.1 Inside/Outside Testing 175
7.10.2 Interiors of Nonsimple Polygons 177
7.10.3 The Signed Area of a Plane Polygon: Divide and Conquer 177
7.10.4 Normal to a Polygon in Space 178
7.10.5 Signed Areas for More General Polygons 179
7.10.6 The Tilting Principle 180
7.10.7 Analogs of Barycentric Coordinates 182
7.11 Discussion 182
7.12 Exercises 182

8 A Simple Way to Describe Shape in 2D and 3D 187

The triangle mesh is a fundamental structure in graphics, widely used for representing shape. We describe 1D meshes (polylines) in 2D and generalize to 2D meshes in 3D. We discuss several representations for triangle meshes, simple operations on meshes such as computing the boundary, and determining whether a mesh is oriented.

8.1 Introduction 187
8.2 "Meshes" in 2D: Polylines 189
8.2.1 Boundaries 190
8.2.2 A Data Structure for 1D Meshes 191
8.3 Meshes in 3D 192

8.3.1 Manifold Meshes 193
8.3.2 Nonmanifold Meshes 195
8.3.3 Memory Requirements for Mesh Structures 196
8.3.4 A Few Mesh Operations 197
8.3.5 Edge Collapse 197
8.3.6 Edge Swap 197
8.4 Discussion and Further Reading 198
8.5 Exercises 198

9 Functions on Meshes 201

A real-valued function defined at the vertices of a mesh can be extended linearly across each face by barycentric interpolation to define a function on the entire mesh. Such extensions are used in texture mapping, for instance. By considering what happens when a single vertex value is 1, and all others are 0, we see that all our piecewise-linear extensions are combinations of certain basic piecewise-linear mesh functions; replacing these basis functions with other, smoother functions can lead to smoother interpolation of values.

9.1 Introduction 201
9.2 Code for Barycentric Interpolation 203
9.2.1 A Different View of Linear Interpolation 207
9.2.2 Scanline Interpolation 208
9.3 Limitations of Piecewise Linear Extension 210
9.3.1 Dependence on Mesh Structure 211
9.4 Smoother Extensions 211
9.4.1 Nonconvex Spaces 211
9.4.2 Which Interpolation Method Should I Really Use? 213
9.5 Functions Multiply Defined at Vertices 213
9.6 Application: Texture Mapping 214
9.6.1 Assignment of Texture Coordinates 215
9.6.2 Details of Texture Mapping 216
9.6.3 Texture-Mapping Problems 216
9.7 Discussion 217
9.8 Exercises 217

10 Transformations in Two Dimensions 221

Linear and affine transformations are the building blocks of graphics. They occur in modeling, in rendering, in animation, and in just about every other context imaginable. They are the natural tools for transforming objects represented as meshes, because they preserve the mesh structure perfectly. We introduce linear and affine transformations in the plane, because most of the interesting phenomena are present there, the exception being the behavior of rotations in three dimensions, which we discuss in Chapter 11. We also discuss the relationship of transformations to matrices, the use of homogeneous coordinates, the uses of hierarchies of transformations in modeling, and the idea of coordinate "frames."

10.1 Introduction 221
10.2 Five Examples 222

10.3 Important Facts about Transformations 224
10.3.1 Multiplication by a Matrix Is a Linear Transformation 224
10.3.2 Multiplication by a Matrix Is the *Only* Linear Transformation 224
10.3.3 Function Composition and Matrix Multiplication Are Related 225
10.3.4 Matrix Inverse and Inverse Functions Are Related 225
10.3.5 Finding the Matrix for a Transformation 226
10.3.6 Transformations and Coordinate Systems 229
10.3.7 Matrix Properties and the Singular Value Decomposition 230
10.3.8 Computing the SVD 231
10.3.9 The SVD and Pseudoinverses 231
10.4 Translation 233
10.5 Points and Vectors Again 234
10.6 Why Use 3 $\times$ 3 Matrices Instead of a Matrix and a Vector? 235
10.7 Windowing Transformations 236
10.8 Building 3D Transformations 237
10.9 Another Example of Building a 2D Transformation 238
10.10 Coordinate Frames 240
10.11 Application: Rendering from a Scene Graph 241
10.11.1 Coordinate Changes in Scene Graphs 248
10.12 Transforming Vectors and Covectors 250
10.12.1 Transforming Parametric Lines 254
10.13 More General Transformations 254
10.14 Transformations versus Interpolation 259
10.15 Discussion and Further Reading 259
10.16 Exercises 260

11 Transformations in Three Dimensions 263

Transformations in 3-space are analogous to those in the plane, except for rotations: In the plane, we can swap the order in which we perform two rotations about the origin without altering the result; in 3-space, we generally cannot. We discuss the group of rotations in 3-space, the use of quaternions to represent rotations, interpolating between quaternions, and a more general technique for interpolating among any sequence of transformations, provided they are "close enough" to one another. Some of these techniques are applied to user-interface designs in Chapter 21.

11.1 Introduction 263
11.1.1 Projective Transformation Theorems 265
11.2 Rotations 266
11.2.1 Analogies between Two and Three Dimensions 266
11.2.2 Euler Angles 267
11.2.3 Axis-Angle Description of a Rotation 269
11.2.4 Finding an Axis and Angle from a Rotation Matrix 270
11.2.5 Body-Centered Euler Angles 272
11.2.6 Rotations and the 3-Sphere 273
11.2.7 Stability of Computations 278
11.3 Comparing Representations 278
11.4 Rotations versus Rotation Specifications 279
11.5 Interpolating Matrix Transformations 280
11.6 Virtual Trackball and Arcball 280

11.7 Discussion and Further Reading 283
11.8 Exercises 284

12 A 2D and 3D Transformation Library for Graphics 287

Because we represent so many things in graphics with arrays of three floating-point numbers (RGB colors, locations in 3-space, vectors in 3-space, covectors in 3-space, etc.) it's very easy to make conceptual mistakes in code, performing operations (like adding the coordinates of two points) that don't make sense. We present a sample mathematics library that you can use to avoid such problems. While such a library may have no place in high-performance graphics, where the overhead of type checking would be unreasonable, it can be very useful in the development of programs in their early stages.

12.1 Introduction 287
12.2 Points and Vectors 288
12.3 Transformations 288
12.3.1 Efficiency 289
12.4 Specification of Transformations 290
12.5 Implementation 290
12.5.1 Projective Transformations 291
12.6 Three Dimensions 293
12.7 Associated Transformations 294
12.8 Other Structures 294
12.9 Other Approaches 295
12.10 Discussion 297
12.11 Exercises 297

13 Camera Specifications and Transformations 299

To convert a model of a 3D scene to a 2D image seen from a particular point of view, we have to specify the view precisely. The rendering process turns out to be particularly simple if the camera is at the origin, looking along a coordinate axis, and if the field of view is 90° in each direction. We therefore transform the general problem to the more specific one. We discuss how the virtual camera is specified, and how we transform any rendering problem to one in which the camera is in a standard position with standard characteristics. We also discuss the specification of parallel (as opposed to perspective) views.

13.1 Introduction 299
13.2 A 2D Example 300
13.3 Perspective Camera Specification 301
13.4 Building Transformations from a View Specification 303
13.5 Camera Transformations and the Rasterizing Renderer Pipeline 310
13.6 Perspective and z-values 313
13.7 Camera Transformations and the Modeling Hierarchy 313
13.8 Orthographic Cameras 315
13.8.1 Aspect Ratio and Field of View 316
13.9 Discussion and Further Reading 317
13.10 Exercises 318

14 Standard Approximations and Representations 321

The real world contains too much detail to simulate efficiently from first principles of physics and geometry. Models make graphics computationally tractable but introduce restrictions and errors. We explore some pervasive approximations and their limitations. In many cases, we have a choice between competing models with different properties.

14.1 Introduction 321
14.2 Evaluating Representations 322
14.2.1 The Value of Measurement 323
14.2.2 Legacy Models 324
14.3 Real Numbers 324
14.3.1 Fixed Point 325
14.3.2 Floating Point 326
14.3.3 Buffers 327
14.4 Building Blocks of Ray Optics 330
14.4.1 Light 330
14.4.2 Emitters 334
14.4.3 Light Transport 335
14.4.4 Matter 336
14.4.5 Cameras 336
14.5 Large-Scale Object Geometry 337
14.5.1 Meshes 338
14.5.2 Implicit Surfaces 341
14.5.3 Spline Patches and Subdivision Surfaces 343
14.5.4 Heightfields 344
14.5.5 Point Sets 345
14.6 Distant Objects 346
14.6.1 Level of Detail 347
14.6.2 Billboards and Impostors 347
14.6.3 Skyboxes 348
14.7 Volumetric Models 349
14.7.1 Finite Element Models 349
14.7.2 Voxels 349
14.7.3 Particle Systems 350
14.7.4 Fog 351
14.8 Scene Graphs 351
14.9 Material Models 353
14.9.1 Scattering Functions (BSDFs) 354
14.9.2 Lambertian 358
14.9.3 Normalized Blinn-Phong 359
14.10 Translucency and Blending 361
14.10.1 Blending 362
14.10.2 Partial Coverage (α) 364
14.10.3 Transmission 367
14.10.4 Emission 369
14.10.5 Bloom and Lens Flare 369
14.11 Luminaire Models 369
14.11.1 The Radiance Function 370
14.11.2 Direct and Indirect Light 370

14.11.3 Practical and Artistic Considerations 370
14.11.4 Rectangular Area Light 377
14.11.5 Hemisphere Area Light 378
14.11.6 Omni-Light 379
14.11.7 Directional Light 380
14.11.8 Spot Light 381
14.11.9 A Unified Point-Light Model 382
14.12 Discussion 384
14.13 Exercises 385

15 Ray Casting and Rasterization 387

A 3D renderer identifies the surface that covers each pixel of an image, and then executes some shading routine to compute the value of the pixel. We introduce a set of coverage algorithms and some straw-man shading routines, and revisit the graphics pipeline abstraction. These are practical design points arising from general principles of geometry and processor architectures.

For coverage, we derive the ray-casting and rasterization algorithms and then build the complete source code for a render on top of it. This requires graphics-specific debugging techniques such as visualizing intermediate results. Architecture-aware optimizations dramatically increase the performance of these programs, albeit by limiting abstraction. Alternatively, we can move abstractions above the pipeline to enable dedicated graphics hardware. APIs abstracting graphics processing units (GPUs) enable efficient rasterization implementations. We port our render to the programmable shading framework common to such APIs.

15.1 Introduction 387
15.2 High-Level Design Overview 388
15.2.1 Scattering 388
15.2.2 Visible Points 390
15.2.3 Ray Casting: Pixels First 391
15.2.4 Rasterization: Triangles First 391
15.3 Implementation Platform 393
15.3.1 Selection Criteria 393
15.3.2 Utility Classes 395
15.3.3 Scene Representation 400
15.3.4 A Test Scene 402
15.4 A Ray-Casting Renderer 403
15.4.1 Generating an Eye Ray 404
15.4.2 Sampling Framework: Intersect and Shade 407
15.4.3 Ray-Triangle Intersection 408
15.4.4 Debugging 411
15.4.5 Shading 412
15.4.6 Lambertian Scattering 413
15.4.7 Glossy Scattering 414
15.4.8 Shadows 414
15.4.9 A More Complex Scene 417
15.5 Intermezzo 417
15.6 Rasterization 418
15.6.1 Swapping the Loops 418
15.6.2 Bounding-Box Optimization 420
15.6.3 Clipping to the Near Plane 422
15.6.4 Increasing Efficiency 422

15.6.5 Rasterizing Shadows 428
15.6.6 Beyond the Bounding Box 429
15.7 Rendering with a Rasterization API 432
15.7.1 The Graphics Pipeline 432
15.7.2 Interface 434
15.8 Performance and Optimization 444
15.8.1 Abstraction Considerations 444
15.8.2 Architectural Considerations 444
15.8.3 Early-Depth-Test Example 445
15.8.4 When Early Optimization Is Good 446
15.8.5 Improving the Asymptotic Bound 447
15.9 Discussion 447
15.10 Exercises 449

16 Survey of Real-Time 3D Graphics Platforms 451

There is great diversity in the feature sets and design goals among 3D graphics platforms. Some are thin layers that bring the application as close to the hardware as possible for optimum performance and control; others provide a thick layer of data structures for the storage and manipulation of complex scenes; and at the top of the power scale are the game-development environments that additionally provide advanced features like physics and joint/skin simulation. Platforms supporting games render with the highest possible speed to ensure interactivity, while those used by the special effects industry sacrifice speed for the utmost in image quality. We present a broad overview of modern 3D platforms with an emphasis on the design goals behind the variations.

16.1 Introduction 451
16.1.1 Evolution from Fixed-Function to Programmable Rendering Pipeline 452
16.2 The Programmer's Model: OpenGL Compatibility (Fixed-Function) Profile 454
16.2.1 OpenGL Program Structure 455
16.2.2 Initialization and the Main Loop 456
16.2.3 Lighting and Materials 458
16.2.4 Geometry Processing 458
16.2.5 Camera Setup 460
16.2.6 Drawing Primitives 461
16.2.7 Putting It All Together—Part 1: Static Frame 462
16.2.8 Putting It All Together—Part 2: Dynamics 463
16.2.9 Hierarchical Modeling 463
16.2.10 Pick Correlation 464
16.3 The Programmer's Model: OpenGL Programmable Pipeline 464
16.3.1 Abstract View of a Programmable Pipeline 464
16.3.2 The Nature of the Core API 466
16.4 Architectures of Graphics Applications 466
16.4.1 The Application Model 466
16.4.2 The Application-Model-to-IM-Platform Pipeline (AMIP) 468
16.4.3 Scene-Graph Middleware 474
16.4.4 Graphics Application Platforms 477
16.5 3D on Other Platforms 478
16.5.1 3D on Mobile Devices 479
16.5.2 3D in Browsers 479
16.6 Discussion 479

17 Image Representation and Manipulation 481

Much of graphics produces *images* as output. We describe how images are stored, what information they can contain, and what they can represent, along with the importance of knowing the precise meaning of the pixels in an image file. We show how to composite images (i.e., blend, overlay, and otherwise merge them) using coverage maps, and how to simply represent images at multiple scales with MIP mapping.

17.1 Introduction 481
17.2 What *Is* an Image? 482
17.2.1 The Information Stored in an Image 482
17.3 Image File Formats 483
17.3.1 Choosing an Image Format 484
17.4 Image Compositing 485
17.4.1 The Meaning of a Pixel During Image Compositing 486
17.4.2 Computing $\boldsymbol{U}$ over $\boldsymbol{V}$ 486
17.4.3 Simplifying Compositing 487
17.4.4 Other Compositing Operations 488
17.4.5 Physical Units and Compositing 489
17.5 Other Image Types 490
17.5.1 Nomenclature 491
17.6 MIP Maps 491
17.7 Discussion and Further Reading 492
17.8 Exercises 493

18 Images and Signal Processing 495

The pattern of light arriving at a camera sensor can be thought of as a function defined on a 2D rectangle, the value at each point being the light energy density arriving there. The resultant image is an array of values, each one arrived at by some sort of averaging of the input function. The relationship between these two functions—one defined on a continuous 2D rectangle, the other defined on a rectangular grid of points—is a deep one. We study the relationship with the tools of Fourier analysis, which lets us understand what parts of the incoming signal can be accurately captured by the discrete signal. This understanding helps us avoid a wide range of image problems, including "jaggies" (ragged edges). It's also the basis for understanding other phenomena in graphics, such as moiré patterns in textures.

18.1 Introduction 495
18.1.1 A Broad Overview 495
18.1.2 Important Terms, Assumptions, and Notation 497
18.2 Historical Motivation 498
18.3 Convolution 500
18.4 Properties of Convolution 503
18.5 Convolution-like Computations 504
18.6 Reconstruction 505
18.7 Function Classes 505
18.8 Sampling 507
18.9 Mathematical Considerations 508
18.9.1 Frequency-Based Synthesis and Analysis 509
18.10 The Fourier Transform: Definitions 511

18.11 The Fourier Transform of a Function on an Interval 511
18.11.1 Sampling and Band Limiting in an Interval 514
18.12 Generalizations to Larger Intervals and All of R 516
18.13 Examples of Fourier Transforms 516
18.13.1 Basic Examples 516
18.13.2 The Transform of a Box Is a Sinc 517
18.13.3 An Example on an Interval 518
18.14 An Approximation of Sampling 519
18.15 Examples Involving Limits 519
18.15.1 Narrow Boxes and the Delta Function 519
18.15.2 The Comb Function and Its Transform 520
18.16 The Inverse Fourier Transform 520
18.17 Properties of the Fourier Transform 521
18.18 Applications 522
18.18.1 Band Limiting 522
18.18.2 Explaining Replication in the Spectrum 523
18.19 Reconstruction and Band Limiting 524
18.20 Aliasing Revisited 527
18.21 Discussion and Further Reading 529
18.22 Exercises 532

19 Enlarging and Shrinking Images 533

We apply the ideas of the previous two chapters to a concrete example—enlarging and shrinking of images—to illustrate their use in practice. We see that when an image, conventionally represented, is shrunk, problems will arise unless certain high-frequency information is removed before the shrinking process.

19.1 Introduction 533
19.2 Enlarging an Image 534
19.3 Scaling Down an Image 537
19.4 Making the Algorithms Practical 538
19.5 Finite-Support Approximations 540
19.5.1 Practical Band Limiting 541
19.6 Other Image Operations and Efficiency 541
19.7 Discussion and Further Reading 544
19.8 Exercises 545

20 Textures and Texture Mapping 547

Texturing, and its variants, add visual richness to models without introducing geometric complexity. We discuss basic texturing and its implementation in software, and some of its variants, like bump mapping and displacement mapping, and the use of 1D and 3D textures. We also discuss the creation of texture correspondences (assigning texture coordinates to points on a mesh) and of the texture images themselves, through techniques as varied as "painting the model" and probabilistic texture-synthesis algorithms.

20.1 Introduction 547
20.2 Variations of Texturing 549

20.2.1 Environment Mapping 549
20.2.2 Bump Mapping 550
20.2.3 Contour Drawing 551
20.3 Building Tangent Vectors from a Parameterization 552
20.4 Codomains for Texture Maps 553
20.5 Assigning Texture Coordinates 555
20.6 Application Examples 557
20.7 Sampling, Aliasing, Filtering, and Reconstruction 557
20.8 Texture Synthesis 559
20.8.1 Fourier-like Synthesis 559
20.8.2 Perlin Noise 560
20.8.3 Reaction-Diffusion Textures 561
20.9 Data-Driven Texture Synthesis 562
20.10 Discussion and Further Reading 564
20.11 Exercises 565

21 Interaction Techniques 567

Certain interaction techniques use a substantial amount of the mathematics of transformations, and therefore are more suitable for a book like ours than one that concentrates on the design of the interaction itself, and the human factors associated with that design. We illustrate these ideas with three 3D manipulators—the arcball, trackball, and Unicam—and with a a multitouch interface for manipulating images.

21.1 Introduction 567
21.2 User Interfaces and Computer Graphics 567
21.2.1 Prescriptions 571
21.2.2 Interaction Event Handling 573
21.3 Multitouch Interaction for 2D Manipulation 574
21.3.1 Defining the Problem 575
21.3.2 Building the Program 576
21.3.3 The Interactor 576
21.4 Mouse-Based Object Manipulation in 3D 580
21.4.1 The Trackball Interface 580
21.4.2 The Arcball Interface 584
21.5 Mouse-Based Camera Manipulation: Unicam 584
21.5.1 Translation 585
21.5.2 Rotation 586
21.5.3 Additional Operations 587
21.5.4 Evaluation 587
21.6 Choosing the Best Interface 587
21.7 Some Interface Examples 588
21.7.1 First-Person-Shooter Controls 588
21.7.2 3ds Max Transformation Widget 588
21.7.3 Photoshop's Free-Transform Mode 589
21.7.4 Chateau 589
21.7.5 Teddy 590
21.7.6 Grabcut and Selection by Strokes 590
21.8 Discussion and Further Reading 591
21.9 Exercises 593

22 Splines and Subdivision Curves 595

Splines are, informally, curves that pass through or near a sequence of "control points." They're used to describe shapes, and to control the motion of objects in animations, among other things. Splines make sense not only in the plane, but also in 3-space and in 1-space, where they provide a means of interpolating a sequence of values with various degrees of continuity. Splines, as a modeling tool in graphics, have been in part supplanted by subdivision curves (which we saw in the form of corner-cutting curves in Chapter 4) and subdivision surfaces. The two classes—splines and subdivision—are closely related. We demonstrate this for curves in this chapter; a similar approach works for surfaces.

22.1 Introduction 595
22.2 Basic Polynomial Curves 595
22.3 Fitting a Curve Segment between Two Curves: The Hermite Curve 595
22.3.1 Bézier Curves 598
22.4 Gluing Together Curves and the Catmull-Rom Spline 598
22.4.1 Generalization of Catmull-Rom Splines 601
22.4.2 Applications of Catmull-Rom Splines 602
22.5 Cubic B-splines 602
22.5.1 Other B-splines 604
22.6 Subdivision Curves 604
22.7 Discussion and Further Reading 605
22.8 Exercises 605

23 Splines and Subdivision Surfaces 607

Spline surfaces and subdivision surfaces are natural generalizations of spline and subdivision curves. Surfaces are built from rectangular patches, and when these meet four at a vertex, the generalization is reasonably straightforward. At vertices where the degree is not four, certain challenges arise, and dealing with these "exceptional vertices" requires care. Just as in the case of curves, subdivision surfaces, away from exceptional vertices, turn out to be identical to spline surfaces. We discuss spline patches, Catmull-Clark subdivision, other subdivision approaches, and the problems of exceptional points.

23.1 Introduction 607
23.2 Bézier Patches 608
23.3 Catmull-Clark Subdivision Surfaces 610
23.4 Modeling with Subdivision Surfaces 613
23.5 Discussion and Further Reading 614

24 Implicit Representations of Shape 615

Implicit curves are defined as the level set of some function on the plane; on a weather map, the isotherm lines constitute implicit curves. By choosing particular functions, we can make the shapes of these curves controllable. The same idea applies in space to define implicit surfaces. In each case, it's not too difficult to convert an implicit representation to a mesh representation that approximates the surface. But the implicit representation itself has many advantages. Finding a ray-shape intersection with an implicit surface reduces to root finding, for instance, and it's easy to combine implicit shapes with operators that result in new shapes without sharp corners.

24.1 Introduction 615
24.2 Implicit Curves 616
24.3 Implicit Surfaces 619
24.4 Representing Implicit Functions 621
24.4.1 Interpolation Schemes 621
24.4.2 Splines 623
24.4.3 Mathematical Models and Sampled Implicit Representations 623
24.5 Other Representations of Implicit Functions 624
24.6 Conversion to Polyhedral Meshes 625
24.6.1 Marching Cubes 628
24.7 Conversion from Polyhedral Meshes to Implicits 629
24.8 Texturing Implicit Models 629
24.8.1 Modeling Transformations and Textures 630
24.9 Ray Tracing Implicit Surfaces 631
24.10 Implicit Shapes in Animation 631
24.11 Discussion and Further Reading 632
24.12 Exercises 633

25 Meshes 635

Meshes are a dominant structure in today's graphics. They serve as approximations to smooth curves and surfaces, and much mathematics from the smooth category can be transferred to work with meshes. Certain special classes of meshes—heightfield meshes, and very regular meshes—support fast algorithms particularly well. We discuss level of detail in the context of meshes, where practical algorithms abound, but also in a larger context. We conclude with some applications.

25.1 Introduction 635
25.2 Mesh Topology 637
25.2.1 Triangulated Surfaces and Surfaces with Boundary 637
25.2.2 Computing and Storing Adjacency 638
25.2.3 More Mesh Terminology 641
25.2.4 Embedding and Topology 642
25.3 Mesh Geometry 643
25.3.1 Mesh Meaning 644
25.4 Level of Detail 645
25.4.1 Progressive Meshes 649
25.4.2 Other Mesh Simplification Approaches 652
25.5 Mesh Applications 1: Marching Cubes, Mesh Repair, and Mesh Improvement 652
25.5.1 Marching Cubes Variants 652
25.5.2 Mesh Repair 654
25.5.3 Differential or Laplacian Coordinates 655
25.5.4 An Application of Laplacian Coordinates 657
25.6 Mesh Applications 2: Deformation Transfer and Triangle-Order Optimization 660
25.6.1 Deformation Transfer 660
25.6.2 Triangle Reordering for Hardware Efficiency 664
25.7 Discussion and Further Reading 667
25.8 Exercises 668

26 Light 669

We discuss the basic physics of light, starting from blackbody radiation, and the relevance of this physics to computer graphics. In particular, we discuss both the wave and particle descriptions of light, polarization effects, and diffraction. We then discuss the measurement of light, including the various units of measure, and the continuum assumption implicit in these measurements. We focus on the radiance, from which all other radiometric terms can be derived through integration, and which is constant along rays in empty space. Because of the dependence on integration, we discuss solid angles and integration over these. Because the radiance field in most scenes is too complex to express in simple algebraic terms, integrals of radiance are almost always computed stochastically, and so we introduce stochastic integration. Finally, we discuss reflectance and transmission, their measurement, and the challenges of computing integrals in which the integrands have substantial variation (like the specular and nonspecular parts of the reflection from a glossy surface).

26.1 Introduction 669
26.2 The Physics of Light 669
26.3 The Microscopic View 670
26.4 The Wave Nature of Light 674
26.4.1 Diffraction 677
26.4.2 Polarization 677
26.4.3 Bending of Light at an Interface 679
26.5 Fresnel's Law and Polarization 681
26.5.1 Radiance Computations and an "Unpolarized" Form of Fresnel's Equations 683
26.6 Modeling Light as a Continuous Flow 683
26.6.1 A Brief Introduction to Probability Densities 684
26.6.2 Further Light Modeling 686
26.6.3 Angles and Solid Angles 686
26.6.4 Computations with Solid Angles 688
26.6.5 An Important Change of Variables 690
26.7 Measuring Light 692
26.7.1 Radiometric Terms 694
26.7.2 Radiance 694
26.7.3 Two Radiance Computations 695
26.7.4 Irradiance 697
26.7.5 Radiant Exitance 699
26.7.6 Radiant Power or Radiant Flux 699
26.8 Other Measurements 700
26.9 The Derivative Approach 700
26.10 Reflectance 702
26.10.1 Related Terms 704
26.10.2 Mirrors, Glass, Reciprocity, and the BRDF 705
26.10.3 Writing L in Different Ways 706
26.11 Discussion and Further Reading 707
26.12 Exercises 707

27 Materials and Scattering 711

The appearance of an object made of some material is determined by the interaction of that material with the light in the scene. The interaction (for fairly homogeneous materials) is described by the

reflection and transmission distribution functions, at least for at-the-surface scattering. We present several different models for these, ranging from the purely empirical to those incorporating various degrees of physical realism, and observe their limitations as well. We briefly discuss scattering from volumetric media like smoke and fog, and the kind of subsurface scattering that takes place in media like skin and milk. Anticipating our use of these material models in rendering, we also discuss the software interface a material model must support to be used effectively.

27.1 Introduction ... 711
27.2 Object-Level Scattering ... 711
27.3 Surface Scattering ... 712
27.3.1 Impulses ... 713
27.3.2 Types of Scattering Models ... 713
27.3.3 Physical Constraints on Scattering ... 713
27.4 Kinds of Scattering ... 714
27.5 Empirical and Phenomenological Models for Scattering ... 717
27.5.1 Mirror "Scattering" ... 717
27.5.2 Lambertian Reflectors ... 719
27.5.3 The Phong and Blinn-Phong Models ... 721
27.5.4 The Lafortune Model ... 723
27.5.5 Sampling ... 724
27.6 Measured Models ... 725
27.7 Physical Models for Specular and Diffuse Reflection ... 726
27.8 Physically Based Scattering Models ... 727
27.8.1 The Fresnel Equations, Revisited ... 727
27.8.2 The Torrance-Sparrow Model ... 729
27.8.3 The Cook-Torrance Model ... 731
27.8.4 The Oren-Nayar Model ... 732
27.8.5 Wave Theory Models ... 734
27.9 Representation Choices ... 734
27.10 Criteria for Evaluation ... 734
27.11 Variations across Surfaces ... 735
27.12 Suitability for Human Use ... 736
27.13 More Complex Scattering ... 737
27.13.1 Participating Media ... 737
27.13.2 Subsurface Scattering ... 738
27.14 Software Interface to Material Models ... 740
27.15 Discussion and Further Reading ... 741
27.16 Exercises ... 743

28 Color ... 745

While color appears to be a physical property—that book is blue, that sun is yellow—it is, in fact, a perceptual phenomenon, one that's closely related to the spectral distribution of light, but by no means completely determined by it. We describe the perception of color and its relationship to the physiology of the eye. We introduce various systems for naming, representing, and selecting colors. We also discuss the perception of brightness, which is nonlinear as a function of light energy, and the consequences of this for the efficient representation of varying brightness levels, leading to the notion

of *gamma,* an exponent used in compressing brightness data. We also discuss the gamuts (range of colors) of various devices, and the problems of color interpolation.

28.1 Introduction ... 745
28.1.1 Implications of Color ... 746
28.2 Spectral Distribution of Light ... 746
28.3 The Phenomenon of Color Perception and the Physiology of the Eye ... 748
28.4 The Perception of Color ... 750
28.4.1 The Perception of Brightness ... 750
28.5 Color Description ... 756
28.6 Conventional Color Wisdom ... 758
28.6.1 Primary Colors ... 758
28.6.2 Purple Isn't a Real Color ... 759
28.6.3 Objects Have Colors; You Can Tell by Looking at Them in White Light ... 759
28.6.4 Blue and Green Make Cyan ... 760
28.6.5 Color Is RGB ... 761
28.7 Color Perception Strengths and Weaknesses ... 761
28.8 Standard Description of Colors ... 761
28.8.1 The CIE Description of Color ... 762
28.8.2 Applications of the Chromaticity Diagram ... 766
28.9 Perceptual Color Spaces ... 767
28.9.1 Variations and Miscellany ... 767
28.10 Intermezzo ... 768
28.11 White ... 769
28.12 Encoding of Intensity, Exponents, and Gamma Correction ... 769
28.13 Describing Color ... 771
28.13.1 The RGB Color Model ... 772
28.14 CMY and CMYK Color ... 774
28.15 The YIQ Color Model ... 775
28.16 Video Standards ... 775
28.17 HSV and HLS ... 776
28.17.1 Color Choice ... 777
28.17.2 Color Palettes ... 777
28.18 Interpolating Color ... 777
28.19 Using Color in Computer Graphics ... 779
28.20 Discussion and Further Reading ... 780
28.21 Exercises ... 780

29 Light Transport ... 783

Using the formal descriptions of radiance and scattering, we derive the *rendering equation,* an integral equation characterizing the radiance field, given a description of the illumination, geometry, and materials in the scene.

29.1 Introduction ... 783
29.2 Light Transport ... 783
29.2.1 The Rendering Equation, First Version ... 786
29.3 A Peek Ahead ... 787
29.4 The Rendering Equation for General Scattering ... 789
29.4.1 The Measurement Equation ... 791

29.5 Scattering, Revisited 792
29.6 A Worked Example 793
29.7 Solving the Rendering Equation 796
29.8 The Classification of Light-Transport Paths 796
29.8.1 Perceptually Significant Phenomena and Light Transport 797
29.9 Discussion 799
29.10 Exercise 799

30 Probability and Monte Carlo Integration 801

Probabilistic methods are at the heart of modern rendering techniques, especially methods for estimating integrals, because solving the rendering equation involves computing an integral that's impossible to evaluate exactly in any but the simplest scenes. We review basic discrete probability, generalize to continuum probability, and use this to derive the single-sample estimate for an integral and the importance-weighted single-sample estimate, which we'll use in the next two chapters.

30.1 Introduction 801
30.2 Numerical Integration 801
30.3 Random Variables and Randomized Algorithms 802
30.3.1 Discrete Probability and Its Relationship to Programs 803
30.3.2 Expected Value 804
30.3.3 Properties of Expected Value, and Related Terms 806
30.3.4 Continuum Probability 808
30.3.5 Probability Density Functions 810
30.3.6 Application to the Sphere 813
30.3.7 A Simple Example 813
30.3.8 Application to Scattering 814
30.4 Continuum Probability, Continued 815
30.5 Importance Sampling and Integration 818
30.6 Mixed Probabilities 820
30.7 Discussion and Further Reading 821
30.8 Exercises 821

31 Computing Solutions to the Rendering Equation: Theoretical Approaches 825

The rendering equation can be approximately solved by many methods, including ray tracing (an approximation to the series solution), radiosity (an approximation arising from a finite-element approach), Metropolis light transport, and photon mapping, not to mention basic polygonal renderers using direct-lighting-plus-ambient approximations. Each method has strengths and weaknesses that can be analyzed by considering the nature of the materials in the scene, by examining different classes of light paths from luminaires to detectors, and by uncovering various kinds of approximation errors implicit in the methods.

31.1 Introduction 825
31.2 Approximate Solutions of Equations 825
31.3 Method 1: Approximating the Equation 826
31.4 Method 2: Restricting the Domain 827
31.5 Method 3: Using Statistical Estimators 827
31.5.1 Summing a Series by Sampling and Estimation 828

31.6 Method 4: Bisection ... 830
31.7 Other Approaches ... 831
31.8 The Rendering Equation, Revisited ... 831
31.8.1 A Note on Notation ... 835
31.9 What Do We Need to Compute? ... 836
31.10 The Discretization Approach: Radiosity ... 838
31.11 Separation of Transport Paths ... 844
31.12 Series Solution of the Rendering Equation ... 844
31.13 Alternative Formulations of Light Transport ... 846
31.14 Approximations of the Series Solution ... 847
31.15 Approximating Scattering: Spherical Harmonics ... 848
31.16 Introduction to Monte Carlo Approaches ... 851
31.17 Tracing Paths ... 855
31.18 Path Tracing and Markov Chains ... 856
31.18.1 The Markov Chain Approach ... 857
31.18.2 The Recursive Approach ... 861
31.18.3 Building a Path Tracer ... 864
31.18.4 Multiple Importance Sampling ... 868
31.18.5 Bidirectional Path Tracing ... 870
31.18.6 Metropolis Light Transport ... 871
31.19 Photon Mapping ... 872
31.19.1 Image-Space Photon Mapping ... 876
31.20 Discussion and Further Reading ... 876
31.21 Exercises ... 879

32 Rendering in Practice ... 881

We describe the implementation of a path tracer, which exhibits many of the complexities associated with ray-tracing-like renderers that attempt to estimate radiance by estimating integrals associated to the rendering equations, and a photon mapper, which quickly converges to a biased but consistent and plausible result.

32.1 Introduction ... 881
32.2 Representations ... 881
32.3 Surface Representations and Representing BSDFs Locally ... 882
32.3.1 Mirrors and Point Lights ... 886
32.4 Representation of Light ... 887
32.4.1 Representation of Luminaires ... 888
32.5 A Basic Path Tracer ... 889
32.5.1 Preliminaries ... 889
32.5.2 Path-Tracer Code ... 893
32.5.3 Results and Discussion ... 901
32.6 Photon Mapping ... 904
32.6.1 Results and Discussion ... 910
32.6.2 Further Photon Mapping ... 913
32.7 Generalizations ... 914
32.8 Rendering and Debugging ... 915
32.9 Discussion and Further Reading ... 919
32.10 Exercises ... 923

33 Shaders 927

On modern graphics cards, we can execute small (and not-so-small) programs that operate on model data to produce pictures. In the simplest form, these are vertex shaders and fragment shaders, the first of which can do processing based on the geometry of the scene (typically the vertex coordinates), and the second of which can process fragments, which correspond to pieces of polygons that will appear in a single pixel. To illustrate the more basic use of shaders we describe how to implement basic Phong shading, environment mapping, and a simple nonphotorealistic renderer.

33.1 Introduction 927
33.2 The Graphics Pipeline in Several Forms 927
33.3 Historical Development 929
33.4 A Simple Graphics Program with Shaders 932
33.5 A Phong Shader 937
33.6 Environment Mapping 939
33.7 Two Versions of Toon Shading 940
33.8 Basic XToon Shading 942
33.9 Discussion and Further Reading 943
33.10 Exercises 943

34 Expressive Rendering 945

Expressive rendering is the name we give to renderings that do not aim for photorealism, but rather aim to produce imagery that communicates with the viewer, conveying what the creator finds important, and suppressing what's unimportant. We summarize the theoretical foundations of expressive rendering, particularly various kinds of abstraction, and discuss the relationship of the "message" of a rendering and its style. We illustrate with a few expressive rendering techniques.

34.1 Introduction 945
34.1.1 Examples of Expressive Rendering 948
34.1.2 Organization of This Chapter 948
34.2 The Challenges of Expressive Rendering 949
34.3 Marks and Strokes 950
34.4 Perception and Salient Features 951
34.5 Geometric Curve Extraction 952
34.5.1 Ridges and Valleys 956
34.5.2 Suggestive Contours 957
34.5.3 Apparent Ridges 958
34.5.4 Beyond Geometry 959
34.6 Abstraction 959
34.7 Discussion and Further Reading 961

35 Motion 963

An animation is a sequence of rendered frames that gives the perception of smooth motion when displayed quickly. The algorithms to control the underlying 3D object motion generally interpolate between key poses using splines, or simulate the laws of physics by numerically integrating velocity and acceleration. Whereas rendering primarily is concerned with surfaces, animation algorithms require a model with additional properties like articulation and mass. Yet these models still simplify

the real world, accepting limitations to achieve computational efficiency. The hardest problems in animation involve artificial intelligence for planning realistic character motion, which is beyond the scope of this chapter.

35.1 Introduction 963
35.2 Motivating Examples 966
35.2.1 A Walking Character (Key Poses) 966
35.2.2 Firing a Cannon (Simulation) 969
35.2.3 Navigating Corridors (Motion Planning) 972
35.2.4 Notation 973
35.3 Considerations for Rendering 975
35.3.1 Double Buffering 975
35.3.2 Motion Perception 976
35.3.3 Interlacing 978
35.3.4 Temporal Aliasing and Motion Blur 980
35.3.5 Exploiting Temporal Coherence 983
35.3.6 The Problem of the First Frame 984
35.3.7 The Burden of Temporal Coherence 985
35.4 Representations 987
35.4.1 Objects 987
35.4.2 Limiting Degrees of Freedom 988
35.4.3 Key Poses 989
35.4.4 Dynamics 989
35.4.5 Procedural Animation 990
35.4.6 Hybrid Control Schemes 990
35.5 Pose Interpolation 992
35.5.1 Vertex Animation 992
35.5.2 Root Frame Motion 993
35.5.3 Articulated Body 994
35.5.4 Skeletal Animation 995
35.6 Dynamics 996
35.6.1 Particle 996
35.6.2 Differential Equation Formulation 997
35.6.3 Piecewise-Constant Approximation 999
35.6.4 Models of Common Forces 1000
35.6.5 Particle Collisions 1008
35.6.6 Dynamics as a Differential Equation 1012
35.6.7 Numerical Methods for ODEs 1017
35.7 Remarks on Stability in Dynamics 1020
35.8 Discussion 1022

36 Visibility Determination 1023

Efficient determination of the subset of a scene that affects the final image is critical to the performance of a renderer. The first approximation of this process is conservative determination of surfaces visible to the eye. This problem has been addressed by algorithms with radically different space, quality, and time bounds. The preferred algorithms vary over time with the cost and performance of hardware architectures. Because analogous problems arise in collision detection, selection,

global illumination, and document layout, even visibility algorithms that are currently out of favor for primary rays may be preferred in other applications.

36.1 Introduction 1023
36.1.1 The Visibility Function 1025
36.1.2 Primary Visibility 1027
36.1.3 (Binary) Coverage 1027
36.1.4 Current Practice and Motivation 1028
36.2 Ray Casting 1029
36.2.1 BSP Ray-Primitive Intersection 1030
36.2.2 Parallel Evaluation of Ray Tests 1032
36.3 The Depth Buffer 1034
36.3.1 Common Depth Buffer Encodings 1037
36.4 List-Priority Algorithms 1040
36.4.1 The Painter's Algorithm 1041
36.4.2 The Depth-Sort Algorithm 1042
36.4.3 Clusters and BSP Sort 1043
36.5 Frustum Culling and Clipping 1044
36.5.1 Frustum Culling 1044
36.5.2 Clipping 1045
36.5.3 Clipping to the Whole Frustum 1047
36.6 Backface Culling 1047
36.7 Hierarchical Occlusion Culling 1049
36.8 Sector-based Conservative Visibility 1050
36.8.1 Stabbing Trees 1051
36.8.2 Portals and Mirrors 1052
36.9 Partial Coverage 1054
36.9.1 Spatial Antialiasing (xy) 1055
36.9.2 Defocus (uv) 1060
36.9.3 Motion Blur (t) 1061
36.9.4 Coverage as a Material Property (α) 1062
36.10 Discussion and Further Reading 1063
36.11 Exercise 1063

37 Spatial Data Structures 1065

Spatial data structures like bounding volume hierarchies provide intersection queries and set operations on geometry embedded in a metric space. Intersection queries are necessary for light transport, interaction, and dynamics simulation. These structures are classic data structures like hash tables, trees, and graphs extended with the constraints of 3D geometry.

37.1 Introduction 1065
37.1.1 Motivating Examples 1066
37.2 Programmatic Interfaces 1068
37.2.1 Intersection Methods 1069
37.2.2 Extracting Keys and Bounds 1073
37.3 Characterizing Data Structures 1077
37.3.1 1D Linked List Example 1078
37.3.2 1D Tree Example 1079

37.4 Overview of *k*d Structures 1080
37.5 List 1081
37.6 Trees 1083
37.6.1 Binary Space Partition (BSP) Trees 1084
37.6.2 Building BSP Trees: oct tree, quad tree, BSP tree, *k*d tree 1089
37.6.3 Bounding Volume Hierarchy 1092
37.7 Grid 1093
37.7.1 Construction 1093
37.7.2 Ray Intersection 1095
37.7.3 Selecting Grid Resolution 1099
37.8 Discussion and Further Reading 1101

38 Modern Graphics Hardware 1103

We describe the structure of modern graphics cards, their design, and some of the engineering trade-offs that influence this design.

38.1 Introduction 1103
38.2 NVIDIA GeForce 9800 GTX 1105
38.3 Architecture and Implementation 1107
38.3.1 GPU Architecture 1108
38.3.2 GPU Implementation 1111
38.4 Parallelism 1111
38.5 Programmability 1114
38.6 Texture, Memory, and Latency 1117
38.6.1 Texture Mapping 1118
38.6.2 Memory Basics 1121
38.6.3 Coping with Latency 1124
38.7 Locality 1127
38.7.1 Locality of Reference 1127
38.7.2 Cache Memory 1129
38.7.3 Divergence 1132
38.8 Organizational Alternatives 1135
38.8.1 Deferred Shading 1135
38.8.2 Binned Rendering 1137
38.8.3 Larrabee: A CPU/GPU Hybrid 1138
38.9 GPUs as Compute Engines 1142
38.10 Discussion and Further Reading 1143
38.11 Exercises 1143

List of Principles 1145
Bibliography 1149
Index 1183

This page intentionally left blank

Preface

This book presents many of the important ideas of computer graphics to students, researchers, and practitioners. Several of these ideas are not new: They have already appeared in widely available scholarly publications, technical reports, textbooks, and lay-press articles. The advantage of writing a textbook sometime after the appearance of an idea is that its long-term impact can be understood better and placed in a larger context. Our aim has been to treat ideas with as much sophistication as possible (which includes omitting ideas that are no longer as important as they once were), while still introducing beginning students to the subject lucidly and gracefully.

This is a second-generation graphics book: Rather than treating all prior work as implicitly valid, we evaluate it in the context of today's understanding, and update the presentation as appropriate.

Even the most elementary issues can turn out to be remarkably subtle. Suppose, for instance, that you're designing a program that must run in a low-light environment—a darkened movie theatre, for instance. Obviously you cannot use a bright display, and so using brightness contrast to distinguish among different items in your program display would be inappropriate. You decide to use color instead. Unfortunately, color perception in low-light environments is not nearly as good as in high-light environments, and some text colors are easier to read than others in low light. Is your cursor still easy to see? Maybe to make that simpler, you should make the cursor constantly jitter, exploiting the motion sensitivity of the eye. So what seemed like a simple question turns out to involve issues of interface design, color theory, and human perception.

This example, simple as it is, also makes some unspoken assumptions: that the application uses graphics (rather than, say, tactile output or a well-isolated audio earpiece), that it does not use the regular theatre screen, and that it does not use a head-worn display. It makes explicit assumptions as well—for instance, that a cursor will be used (some UIs intentionally don't use a cursor). Each of these assumptions reflects a user-interface choice as well.

Unfortunately, this interrelatedness of things makes it impossible to present topics in a completely ordered fashion and still motivate them well; the subject is simply no longer linearizable. We *could* have covered all the mathematics, theory of perception, and other, more abstract, topics first, and only then moved on to their graphics applications. Although this might produce a better reference work (you know just where to look to learn about generalized cross products,

for instance), it doesn't work well for a textbook, since the motivating applications would all come at the end. Alternatively, we could have taken a case-study approach, in which we try to complete various increasingly difficult tasks, and introduce the necessary material as the need arises. This makes for a natural progression in some cases, but makes it difficult to give a broad organizational view of the subject. Our approach is a compromise: We start with some widely used mathematics and notational conventions, and then work from topic to topic, introducing supporting mathematics as needed. Readers already familiar with the mathematics can safely skip this material without missing any computer graphics; others may learn a good deal by reading these sections. Teachers may choose to include or omit them as needed. The topic-based organization of the book entails some redundancy. We discuss the graphics pipeline multiple times at varying levels of detail, for instance. Rather than referring the reader back to a previous chapter, sometimes we redescribe things, believing that this introduces a more natural flow. Flipping back 500 pages to review a figure can be a substantial distraction.

The other challenge for a textbook author is to decide how encyclopedic to make the text. The first edition of this book really did cover a very large fraction of the published work in computer graphics; the second edition at least made passing references to much of the work. This edition abandons any pretense of being encyclopedic, for a very good reason: When the second edition was written, a single person could carry, under one arm, all of the proceedings of SIGGRAPH, the largest annual computer graphics conference, and these constituted a fair representation of all technical writings on the subject. Now the SIGGRAPH proceedings (which are just one of many publication venues) occupy several cubic feet. Even a telegraphic textbook cannot cram all that information into a thousand pages. Our goal in this book is therefore to lead the reader to the point where he or she can read and reproduce many of today's SIGGRAPH papers, albeit with some caveats:

- First, computer graphics and computer vision are overlapping more and more, but there is no excuse for us to write a computer vision textbook; others with far greater knowledge have already done so.
- Second, computer graphics involves programming; many graphics applications are quite large, but we do not attempt to teach either programming or software engineering in this book. We do briefly discuss programming (and especially debugging) approaches that are unique to graphics, however.
- Third, most graphics applications have a user interface. At the time of this writing, most of these interfaces are based on windows with menus, and mouse interaction, although touch-based interfaces are becoming commonplace as well. There was a time when user-interface research was a part of graphics, but it's now an independent community—albeit with substantial overlap with graphics—and we therefore assume that the student has some experience in creating programs with user interfaces, and don't discuss these in any depth, except for some 3D interfaces whose implementations are more closely related to graphics.

Of course, research papers in graphics differ. Some are highly mathematical, others describe large-scale systems with complex engineering tradeoffs, and still others involve a knowledge of physics, color theory, typography, photography, chemistry, zoology... the list goes on and on. Our goal is to prepare the reader to understand the computer graphics in these papers; the other material may require considerable external study as well.

Historical Approaches

The history of computer graphics is largely one of ad hoc approaches to the immediate problems at hand. Saying this is in no way an indictment of the people who took those approaches: They had jobs to do, and found ways to do them. Sometimes their solutions had important ideas wrapped up within them; at other times they were merely ways to get things done, and their adoption has interfered with progress in later years. For instance, the image-compositing model used in most graphics systems assumes that color values stored in images can be blended linearly. In actual practice, the color values stored in images are nonlinearly related to light intensity; taking linear combinations of these does not correspond to taking linear combinations of intensity. The difference between the two approaches began to be noticed when studios tried to combine real-world and computer-generated imagery; this compositing technology produced unacceptable results. In addition, some early approaches were deeply principled, but the associated programs made assumptions about hardware that were no longer valid a few years later; readers, looking first at the details of implementation, said, "Oh, this is old stuff—it's not relevant to us at all," and missed the still important ideas of the research. All too frequently, too, researchers have simply reinvented things known in other disciplines for years.

We therefore do *not* follow the chronological development of computer graphics. Just as physics courses do not begin with Aristotle's description of dynamics, but instead work directly with Newton's (and the better ones describe the limitations of even *that* system, setting the stage for quantum approaches, etc.), we try to start directly from the best current understanding of issues, while still presenting various older ideas when relevant. We also try to point out sources for ideas that may not be familiar ones: Newell's formula for the normal vector to a polygon in 3-space was known to Grassmann in the 1800s, for instance. Our hope in referencing these sources is to increase the reader's awareness of the variety of already developed ideas that are waiting to be applied to graphics.

Pedagogy

The most striking aspect of graphics in our everyday lives is the 3D imagery being used in video games and special effects in the entertainment industry and advertisements. But our day-to-day interactions with home computers, cell phones, etc., are also based on computer graphics. Perhaps they are less visible in part because they are more successful: The best interfaces are the ones you don't notice. It's tempting to say that "2D graphics" is simpler—that 3D graphics is just a more complicated version of the same thing. But many of the issues in 2D graphics—how best to display images on a screen made of a rectangular grid of light-emitting elements, for instance, or how to construct effective and powerful interfaces—are just as difficult as those found in making pictures of three-dimensional scenes. And the simple models conventionally used in 2D graphics can lead the student into false assumptions about how best to represent things like color or shape. We therefore have largely integrated the presentation of 2D and 3D graphics so as to address simultaneously the subtle issues common to both.

This book is unusual in the level at which we put the "black box." Almost every computer science book has to decide at what level to abstract something about the computers that the reader will be familiar with. In a graphics book, we have to

decide what graphics system the reader will be encountering as well. It's not hard (after writing a first program or two) to believe that some combination of hardware and software inside your computer can make a colored triangle appear on your display when you issue certain instructions. The details of how this happens are not relevant to a surprisingly large part of graphics. For instance, what happens if you ask the graphics system to draw a red triangle that's below the displayable area of your screen? Are the pixel locations that need to be made red computed and then ignored because they're off-screen? Or does the graphics system realize, before computing any pixel values, that the triangle is off-screen and just quit? In some sense, unless you're designing a graphics card, it just doesn't matter all that much; indeed, it's something you, as a user of a graphics system, can't really control. In much of the book, therefore, we treat the graphics system as something that can display certain pixel values, or draw triangles and lines, without worrying too much about the "how" of this part. The details *are* included in the chapters on rasterization and on graphics hardware. But because they are mostly beyond our control, topics like clipping, antialiasing of lines, and rasterization algorithms are all postponed to later chapters.

Another aspect of the book's pedagogy is that we generally try to show *how* ideas or techniques arise. This can lead to long explanations, but helps, we hope, when students need to derive something for themselves: The approaches they've encountered may suggest an approach to their current problem.

We believe that the best way to learn graphics is to first learn the mathematics behind it. The drawback of this approach compared to jumping to applications is that learning the abstract math increases the amount of time it takes to learn your first few techniques. But you only pay that overhead once. By the time you're learning the tenth related technique, your investment will pay off because you'll recognize that the new method combines elements you've already studied.

Of course, you're reading this book because you are motivated to write programs that make pictures. So we try to start many topics by diving straight into a solution before stepping back to deeply consider the broader mathematical issues. Most of this book is concerned with that stepping-back process. Having investigated the mathematics, we'll then close out topics by sketching other related problems and some solutions to them. Because we've focused on the underlying principles, you won't need us to tell you the details for these sketches. From your understanding of the principles, the approach of each solution should be clear, and you'll have enough knowledge to be able to read and understand the original cited publication in its author's own words, rather than relying on us to translate it for you. What we *can* do is present some older ideas in a slightly more modern form so that when you go back to read the original paper, you'll have some idea how its vocabulary matches your own.

Current Practice

Graphics is a hands-on discipline. And since the business of graphics is the presentation of visual information to a viewer, and the subsequent interaction with it, graphical tools can often be used effectively to debug new graphical algorithms. But doing this requires the ability to write graphics programs. There are many alternative ways to produce graphical imagery on today's computers, and for much of the material in this book, one method is as good as another. The conversion between one programming language and its libraries and another is

routine. But for teaching the subject, it seems best to work in a single language so that the student can concentrate on the deeper ideas. Throughout this book, we'll suggest exercises to be written using Windows Presentation Foundation (WPF), a widely available graphics system, for which we've written a basic and easily modified program we call a "test bed" in which the student can work. For situations where WPF is not appropriate, we've often used G3D, a publicly available graphics library maintained by one of the authors. And in many situations, we've written pseudocode. It provides a compact way to express ideas, and for most algorithms, actual code (in the language of your choice) can be downloaded from the Web; it seldom makes sense to include it in the text. The formatting of code varies; in cases where programs are developed from an informal sketch to a nearly complete program in some language, things like syntax highlighting make no sense until quite late versions, and may be omitted entirely. Sometimes it's nice to have the code match the mathematics, leading us to use variables with names like x_R, which get typeset as math rather than code. In general, we italicize pseudocode, and use indentation rather than braces in pseudocode to indicate code blocks. In general, our pseudocode is very informal; we use it to convey the broad ideas rather than the details.

This is *not* a book about writing graphics programs, nor is it about *using* them. Readers will find no hints about the best ways to store images in Adobe's latest image-editing program, for instance. But we hope that, having understood the concepts in this book and being competent programmers already, they will both be able to write graphics programs and understand how to use those that are already written.

Principles

Throughout the book we have identified certain computer graphics principles that will help the reader in future work; we've also included sections on current *practice*—sections that discuss, for example, how to approximate your ideal solution on today's hardware, or how to compute your actual ideal solution more rapidly. Even practices that are tuned to today's hardware can prove useful tomorrow, so although in a decade the practices described may no longer be directly applicable, they show approaches that we believe will still be valuable for years.

Prerequisites

Much of this book assumes little more preparation than what a technically savvy undergraduate student may have: the ability to write object-oriented programs; a working knowledge of calculus; some familiarity with vectors, perhaps from a math class or physics class or even a computer science class; and at least some encounter with linear transformations. We also expect that the typical student has written a program or two containing 2D graphical objects like buttons or checkboxes or icons.

Some parts of this book, however, depend on far more mathematics, and attempting to teach that mathematics within the limits of this text is impossible. Generally, however, this sophisticated mathematics is carefully limited to a few sections, and these sections are more appropriate for a graduate course than an introductory one. Both they and certain mathematically sophisticated exercises are marked with a "math road-sign" symbol thus:  . Correspondingly, topics that

use deeper notions from computer science are marked with a "computer science road-sign,"  .

Some mathematical aspects of the text may seem strange to those who have met vectors in other contexts; the first author, whose Ph.D. is in mathematics, certainly was baffled by some of his first encounters with how graphics researchers do things. We attempt to explain these variations from standard mathematical approaches clearly and thoroughly.

Paths through This Book

This book can be used for a semester-long or yearlong undergraduate course, or as a general reference in a graduate course. In an undergraduate course, the advanced mathematical topics can safely be omitted (e.g., the discussions of analogs to barycentric coordinates, manifold meshes, spherical harmonics, etc.) while concentrating on the basic ideas of creating and displaying geometric models, understanding the mathematics of transformations, camera specifications, and the standard models used in representing light, color, reflectance, etc., along with some hints of the limitations of these models. It should also cover basic graphics applications and the user-interface concerns, design tradeoffs, and compromises necessary to make them efficient, possibly ending with some special topic like creating simple animations, or writing a basic ray tracer. Even this is too much for a single semester, and even a yearlong course will leave many sections of the book untouched, as future reading for interested students.

An aggressive semester-long (14-week) course could cover the following.

1. Introduction and a simple 2D program: Chapters 1, 2, and 3.
2. Introduction to the geometry of rendering, and further 2D and 3D programs: Chapters 3 and 4. Visual perception and the human visual system: Chapter 5.
3. Modeling of geometry in 2D and 3D: meshes, splines, and implicit models. Sections 7.1–7.9, Chapters 8 and 9, Sections 22.1–22.4, 23.1–23.3, and 24.1–24.5.
4. Images, part 1: Chapter 17, Sections 18.1–18.11.
5. Images, part 2: Sections 18.12–18.20, Chapter 19.
6. 2D and 3D transformations: Sections 10.1–10.12, Sections 11.1–11.3, Chapter 12.
7. Viewing, cameras, and post-homogeneous interpolation. Sections 13.1–13.7, 15.6.4.
8. Standard approximations in graphics: Chapter 14, selected sections.
9. Rasterization and ray casting: Chapter 15.
10. Light and reflection: Sections 26.1–26.7 (Section 26.5 optional); Section 26.10.
11. Color: Sections 28.1–28.12.
12. Basic reflectance models, light transport: Sections 27.1–27.5, 29.1–29.2, 29.6, 29.8.
13. Recursive ray-tracing details, texture: Sections 24.9, 31.16, 20.1–20.6.

14. Visible surface determination and acceleration data structures; overview of more advanced rendering techniques: selections from Chapters 31, 36, and 37.

However, not all the material in every section would be appropriate for a first course.

Alternatively, consider the syllabus for a 12-week undergraduate course on physically based rendering that takes first principles from offline to real-time rendering. It could dive into the core mathematics and radiometry behind ray tracing, and then cycle back to pick up the computer science ideas needed for scalability and performance.

1. Introduction: Chapter 1
2. Light: Chapter 26
3. Perception; light transport: Chapters 5 and 29
4. A brief overview of meshes and scene graphs: Sections 6.6, 14.1–5
5. Transformations: Chapters 10 and 13, briefly.
6. Ray casting: Sections 15.1–4, 7.6–9
7. Acceleration data structures: Chapter 37; Sections 36.1–36.3, 36.5–36.6, 36.9
8. Rendering theory: Chapters 30 and 31
9. Rendering practice: Chapter 32
10. Color and material: Sections 14.6–14.11, 28, and 27
11. Rasterization: Sections 15.5–9
12. Shaders and hardware: Sections 16.3–5, Chapters 33 and 38

Note that these paths touch chapters out of numerical order. We've intentionally written this book in a style where most chapters are self-contained, with cross-references instead of prerequisites, to support such traversal.

Differences from the Previous Edition

This edition is almost completely new, although many of the topics covered in the previous edition appear here. With the advent of the GPU, triangles are converted to pixels (or samples) by radically different approaches than the old scan-conversion algorithms. We no longer discuss those. In discussing light, we strongly favor physical units of measurement, which adds some complexity to discussions of older techniques that did not concern themselves with units. Rather than preparing two graphics packages for 2D and 3D graphics, as we did for the previous editions, we've chosen to use widely available systems, and provide tools to help the student get started using them.

Website

Often in this book you'll see references to the book's website. It's at `http://cgpp.net` and contains not only the testbed software and several examples

derived from it, but additional material for many chapters, and the interactive experiments in WPF for Chapters 2 and 6.

Acknowledgments

A book like this is written by the authors, but it's enormously enhanced by the contributions of others.

Support and encouragement from Microsoft, especially from Eric Rudder and S. Somasegur, helped to both initiate and complete this project.

The 3D test bed evolved from code written by Dan Leventhal; we also thank Mike Hodnick at kindohm.com, who graciously agreed to let us use his code as a starting point for an earlier draft, and Jordan Parker and Anthony Hodsdon for assisting with WPF.

Two students from Williams College worked very hard in supporting the book: Guedis Cardenas on the bibliography, and Michael Mara on the G3D Innovation Engine used in several chapters; Corey Taylor of Electronic Arts also helped with G3D.

Nancy Pollard of CMU and Liz Marai of the University of Pittsburgh both used early drafts of several chapters in their graphics courses, and provided excellent feedback.

Jim Arvo served not only as an oracle on everything relating to rendering, but helped to reframe the first author's understanding of the field.

Many others, in addition to some of those just mentioned, read chapter drafts, prepared images or figures, suggested topics or ways to present them, or helped out in other ways. In alphabetical order, they are John Anderson, Jim Arvo, Tom Banchoff, Pascal Barla, Connelly Barnes, Brian Barsky, Ronen Barzel, Melissa Byun, Marie-Paule Cani, Lauren Clarke, Elaine Cohen, Doug DeCarlo, Patrick Doran, Kayvon Fatahalian, Adam Finkelstein, Travis Fischer, Roger Fong, Mike Fredrickson, Yudi Fu, Andrew Glassner, Bernie Gordon, Don Greenberg, Pat Hanrahan, Ben Herila, Alex Hills, Ken Joy, Olga Karpenko, Donnie Kendall, Justin Kim, Philip Klein, Joe LaViola, Kefei Lei, Nong Li, Lisa Manekofsky, Bill Mark, John Montrym, Henry Moreton, Tomer Moscovich, Jacopo Pantaleoni, Jill Pipher, Charles Poynton, Rich Riesenfeld, Alyn Rockwood, Peter Schroeder, François Sillion, David Simons, Alvy Ray Smith, Stephen Spencer, Erik Sudderth, Joelle Thollot, Ken Torrance, Jim Valles, Daniel Wigdor, Dan Wilk, Brian Wyvill, and Silvia Zuffi. Despite our best efforts, we have probably forgotten some people, and apologize to them.

It's a sign of the general goodness of the field that we got a lot of support in writing from authors of competing books. Eric Haines, Greg Humphreys, Steve Marschner, Matt Pharr, and Pete Shirley all contributed to making this a better book. It's wonderful to work in a field with folks like this.

We'd never had managed to produce this book without the support, tolerance, indulgence, and vision of our editor, Peter Gordon. And we all appreciate the enormous support of our families throughout this project.

For the Student

Your professor will probably choose some route through this book, selecting topics that fit well together, perhaps following one of the suggested trails mentioned

earlier. Don't let that constrain you. If you want to know about something, use the index and start reading. Sometimes you'll find yourself lacking background, and you won't be able to make sense of what you read. When that happens, read the background material. It'll be easier than reading it at some other time, because right now you have a *reason* to learn it. If you stall out, search the Web for someone's implementation and download and run it. When you notice it doesn't look quite right, you can start examining the implementation, and trying to reverse-engineer it. Sometimes this is a great way to understand something. Follow the practice-theory-practice model of learning: Try something, see whether you can make it work, and if you can't, read up on how others did it, and then try again. The first attempt may be frustrating, but it sets you up to better understand the theory when you get to it. If you can't bring yourself to follow the practice-theory-practice model, at the very least you should take the time to do the inline exercises for any chapter you read.

Graphics is a young field, so young that undergraduates are routinely coauthors on SIGGRAPH papers. In a year you can learn enough to start contributing new ideas.

Graphics also uses a lot of mathematics. If mathematics has always seemed abstract and theoretical to you, graphics can be really helpful: The uses of mathematics in graphics are practical, and you can often *see* the consequences of a theorem in the pictures you make. If mathematics has always come easily to you, you can gain some enjoyment from trying to take the ideas we present and extend them further. While this book contains a lot of mathematics, it only scratches the surface of what gets used in modern research papers.

Finally, *doubt everything*. We've done our best to tell the truth in this book, as we understand it. We think we've done pretty well, and the great bulk of what we've said is true. In a few places, we've deliberately told partial truths when we introduced a notion, and then amplified these in a later section when we're discussing details. But aside from that, we've surely failed to tell the truth in other places as well. In some cases, we've simply made errors, leaving out a minus sign, or making an off-by-one error in a loop. In other cases, the current understanding of the graphics community is just inadequate, and we've believed what others have said, and will have to adjust our beliefs later. These errors are opportunities for you. Martin Gardner said that the true sound of scientific discovery is not "Aha!" but "Hey, *that's* odd. . . ." So if every now and then something seems odd to you, go ahead and doubt it. Look into it more closely. If it turns out to be true, you'll have cleared some cobwebs from your understanding. If it's false, it's a chance for you to advance the field.

For the Teacher

If you're like us, you probably read the "For the Student" section even though it wasn't for you. (And your students are probably reading this part, too.) You know that we've advised them to graze through the book at random, and to doubt everything.

We recommend to you (aside from the suggestions in the remainder of this preface) two things. The first is that you encourage, or even require, that your students answer the inline exercises in the book. To the student who says, "I've got too much to do! I can't waste time stopping to do some exercise," just say, "We

don't have time to stop for gas . . . we're already late." The second is that you assign your students projects or homeworks that have both a fixed goal and an open-ended component. The steady students will complete the fixed-goal parts and learn the material you want to cover. The others, given the chance to do something fun, may do things with the open-ended exercises that will amaze you. And in doing so, they'll find that they need to learn things that might seem *just* out of reach, until they suddenly master them, and become empowered. Graphics is a terrific medium for this: Successes are instantly visible and rewarding, and this sets up a feedback loop for advancement. The combination of visible feedback with the ideas of scalability that they've encountered elsewhere in computer science can be revelatory.

Discussion and Further Reading

Most chapters of this book contain a "Discussion and Further Reading" section like this one, pointing to either background references or advanced applications of the ideas in the chapter. For this preface, the only suitable further reading is very general: We recommend that you immediately begin to look at the proceedings of ACM SIGGRAPH conferences, and of other graphics conferences like Eurographics and Computer Graphics International, and, depending on your evolving interest, some of the more specialized venues like the Eurographics Symposium on Rendering, I3D, and the Symposium on Computer Animation. While at first the papers in these conferences will seem to rely on a great deal of prior knowledge, you'll find that you rapidly get a sense of what things are possible (if only by looking at the pictures), and what sorts of skills are necessary to achieve them. You'll also rapidly discover ideas that keep reappearing in the areas that most interest you, and this can help guide your further reading as you learn graphics.

About the Authors

John F. Hughes (B.A., Mathematics, Princeton, 1977; Ph.D., Mathematics, U.C. Berkeley, 1982) is a Professor of Computer Science at Brown University. His primary research is in computer graphics, particularly those aspects of graphics involving substantial mathematics. As author or co-author of 19 SIGGRAPH papers, he has done research in geometric modeling, user interfaces for modeling, nonphotorealistic rendering, and animation systems. He's served as an associate editor for *ACM Transaction on Graphics* and the *Journal of Graphics Tools,* and has been on the SIGGRAPH program committee multiple times. He co-organized Implicit Surfaces '99, the 2001 Symposium in Interactive 3D Graphics, and the first Eurographics Workshop on Sketch-Based Interfaces and Modeling, and was the Papers Chair for SIGGRAPH 2002.

Andries van Dam is the Thomas J. Watson, Jr. University Professor of Technology and Education, and Professor of Computer Science at Brown University. He has been a member of Brown's faculty since 1965, was a co-founder of Brown's Computer Science Department and its first Chairman from 1979 to 1985, and was also Brown's first Vice President for Research from 2002–2006. Andy's research includes work on computer graphics, hypermedia systems, post-WIMP user interfaces, including immersive virtual reality and pen- and touch-computing, and educational software. He has been working for over four decades on systems for creating and reading electronic books with interactive illustrations for use in teaching and research. In 1967 Andy co-founded ACM SICGRAPH, the forerunner of SIGGRAPH, and from 1985 through 1987 was Chairman of the Computing Research Association. He is a Fellow of ACM, IEEE, and AAAS, a member of the National Academy of Engineering and the American Academy of Arts & Sciences, and holds four honorary doctorates. He has authored or co-authored over 100 papers and nine books.

Morgan McGuire (B.S., MIT, 2000, M.Eng., MIT 2000, Ph.D., Brown University, 2006) is an Associate Professor of Computer Science at Williams College. He's contributed as an industry consultant to products including the Marvel Ultimate Alliance and Titan Quest video game series, the E Ink display used in the Amazon Kindle, and NVIDIA GPUs. Morgan has published papers on high-performance rendering and computational photography in *SIGGRAPH, High Performance Graphics,* the *Eurographics Symposium on Rendering, Interactive*

3D Graphics and Games, and *Non-Photorealistic Animation and Rendering*. He founded the *Journal of Computer Graphics Techniques,* chaired the Symposium on Interactive 3D Graphics and Games and the Symposium on Non-Photorealistic Animation and Rendering, and is the project manager for the G3D Innovation Engine. He is the co-author of *Creating Games, The Graphics Codex,* and chapters of several *GPU Gems, ShaderX* and *GPU Pro* volumes.

David Sklar (B.S., Southern Methodist University, 1982; M.S., Brown University, 1983) is currently a Visualization Engineer at Vizify.com, working on algorithms for presenting animated infographics on computing devices across a wide range of form factors. Sklar served on the computer science faculty at Brown University in the 1980s, presenting introductory courses and co-authoring several chapters of (and the auxiliary software for) the second edition of this book. Subsequently, Sklar transitioned into the electronic-book industry, with a focus on SGML/XML markup standards, during which time he was a frequent presenter at GCA conferences. Thereafter, Sklar and his wife Siew May Chin co-founded PortCompass, one of the first online retail shore-excursion marketers, which was the first in a long series of entrepreneurial start-up endeavors in a variety of industries ranging from real-estate management to database consulting.

James Foley (B.S.E.E., Lehigh University, 1964; M.S.E.E., University of Michigan 1965; Ph.D., University of Michigan, 1969) holds the Fleming Chair and is Professor of Interactive Computing in the College of Computing at Georgia Institute of Technology. He previously held faculty positions at UNC-Chapel Hill and The George Washington University and management positions at Mitsubishi Electric Research. In 1992 he founded the GVU Center at Georgia Tech and served as director through 1996. During much of that time he also served as editor-in-chief of *ACM Transactions on Graphics*. His research contributions have been to computer graphics, human-computer interaction, and information visualization. He is a co-author of three editions of this book and of its 1980 predecessor, *Fundamentals of Interactive Computer Graphics*. He is a fellow of the ACM, the American Association for the Advancement of Science and IEEE, recipient of lifetime achievement awards from SIGGRAPH (the Coons award) and SIGCHI, and a member of the National Academy of Engineering.

Steven Feiner (A.B., Music, Brown University, 1973; Ph.D., Computer Science, Brown University, 1987) is a Professor of Computer Science at Columbia University, where he directs the Computer Graphics and User Interfaces Lab and co-directs the Columbia Vision and Graphics Center. His research addresses 3D user interfaces, augmented reality, wearable computing, and many topics at the intersection of human-computer interaction and computer graphics. Steve has served as an associate editor of *ACM Transactions on Graphics,* a member of the editorial board of *IEEE Transactions on Visualization and Computer Graphics,* and a member of the editorial advisory board of *Computers & Graphics*. He was elected to the CHI Academy and, together with his students, has received the ACM UIST Lasting Impact Award, and best paper awards from IEEE ISMAR, ACM VRST, ACM CHI, and ACM UIST. Steve has been program chair or co-chair for many conferences, such as IEEE Virtual Reality, ACM Symposium on User Interface Software & Technology, Foundations of Digital Games, ACM Symposium

on Virtual Reality Software & Technology, IEEE International Symposium on Wearable Computers, and ACM Multimedia.

Kurt Akeley (B.E.E., University of Delaware, 1980; M.S.E.E., Stanford University, 1982; Ph.D., Electrical Engineering, Stanford University, 2004) is Vice President of Engineering at Lytro, Inc. Kurt is a co-founder of Silicon Graphics (later SGI), where he led the development of a sequence of high-end graphics systems, including RealityEngine, and also led the design and standardization of the OpenGL graphics system. He is a Fellow of the ACM, a recipient of ACM's SIGGRAPH computer graphics achievement award, and a member of the National Academy of Engineering. Kurt has authored or co-authored papers published in *SIGGRAPH, High Performance Graphics, Journal of Vision,* and *Optics Express*. He has twice chaired the SIGGRAPH technical papers program, first in 2000, and again in 2008 for the inaugural SIGGRAPH Asia conference.

This page intentionally left blank

Chapter 1

Introduction

This chapter introduces computer graphics quite broadly and from several perspectives: its applications, the various fields that are involved in the study of graphics, some of the tools that make the images produced by graphics so effective, some numbers to help you understand the scales at which computer graphics works, and the elementary ideas required to write your first graphics program. We'll discuss many of these topics in more detail elsewhere in the book.

1.1 An Introduction to Computer Graphics

Computer graphics is the science and art of communicating visually via a computer's display and its interaction devices. The visual aspect of the communication is usually in the computer-to-human direction, with the human-to-computer direction being mediated by devices like the mouse, keyboard, joystick, game controller, or touch-sensitive overlay. However, even this is beginning to change: Visual data is starting to flow *back* to the computer, with new interfaces being based on computer vision algorithms applied to video or depth-camera input. But for the computer-to-user direction, the ultimate consumers of the communications are human, and thus the ways that humans perceive imagery are critical in the design of graphics[1] programs—features that humans ignore need not be presented (nor computed!). Computer graphics is a cross-disciplinary field in which physics, mathematics, human perception, human-computer interaction, engineering, graphic design, and art all play important roles. We use physics to model light and to perform simulations for animation. We use mathematics to describe shape. Human perceptual abilities determine our allocation of resources—we don't want to spend time rendering things that will not be noticed. We use engineering in optimizing the allocation of bandwidth, memory, and processor time. Graphic design and art combine with human-computer interaction to make the computer-to-human direction of communication most effective. In this chapter,

1. Throughout this book, when we use the term "graphics" we mean "computer graphics."

we discuss some application areas, how conventional graphics systems work, and how each of these disciplines influences work in computer graphics.

A narrow definition of computer graphics would state that it refers to taking a model of the objects in a scene (a geometric description of the things in the scene and a description of how they reflect light) and a model of the light emitted into the scene (a mathematical description of the sources of light energy, the directions of radiation, the distribution of light wavelengths, etc.), and then producing a representation of a particular *view* of the scene (the light arriving at some imaginary eye or camera in the scene). In this view, one might say that graphics is just glorified multiplication: One multiplies the incoming light by the reflectivities of objects in the scene to compute the light leaving those objects' surfaces and repeats the process (treating the surfaces as new light sources and recursively invoking the light-transport operation), determining all light that eventually reaches the camera. (In practice, this approach is unworkable, but the idea remains.) In contrast, computer vision amounts to *factoring*—given a view of a scene, the computer vision system is charged with determining the illumination and/or the scene's contents (which a graphics system could then "multiply" together to reproduce the same image). In truth, of course, the vision system cannot solve the problem as stated and typically works with assumptions about the scene, or the lighting, or both, and may also have multiple views of the scene from different cameras, or multiple views from a single camera but at different times.

In the field of computer graphics, the word "model" can refer to a geometric model or a mathematical model. A **geometric model** is a model of something we plan to have appear in a picture: We make a model of a car, or a house, or an armadillo. The geometric model is enhanced with various other attributes that describe the color or texture or reflectance of the materials involved in the model. Starting from nothing and creating such a model is called **modeling,** and the geometric-plus-other-information description that is the result is called a **model.**

A **mathematical model** is a model of a physical or computational process. For instance, in Chapter 27 we describe various models of how light reflects from glossy surfaces. We also have models of how objects move and models of things like the image-acquisition process that happens in a digital camera. Such models may be faithful (i.e., may provide a predictive and correct mathematical model of the phenomenon) or not; they may be physically based, derived from first principles, or perhaps empirical or phenomenological, derived from observations or even intuition.

In actual fact, graphics is far richer than the generalized multiplication process of rendering a view, just as vision is richer than factorization. Much of the current research in graphics is in methods for creating geometric models, methods for representing surface reflectance (and subsurface reflectance, and reflectances of participating media such as fog and smoke, etc.), the animation of scenes by physical laws and by approximations of those laws, the control of animation, interaction with virtual objects, the invention of nonphotorealistic representations, and, in recent years, an increasing integration of techniques from computer vision. As a result, the fields of computer graphics and computer vision are growing increasingly closer to each other. For example, consider Raskar's work on a

Figure 1.1: A nonphotorealistic camera can create an artistic rendering of a scene by applying computer vision techniques to multiple flash-photo images and then rerendering the scene using computer graphics techniques. At left is the original scene; at right is the new rendering of the scene. (Courtesy of Ramesh Raskar; ©2004 ACM, Inc. Included here by permission.)

nonphotorealistic camera: The camera takes multiple photos of a single scene, illuminated by differently placed flash units. From these various images, one can use computer vision techniques to determine contours and estimate some basic shape properties for objects in the scene. These, in turn, can be used to create a nonphotorealistic rendering of the scene, as shown in Figure 1.1.

In this book, we emphasize realistic image capture and rendering because this is where the field of computer graphics has had the greatest successes, representing a beautiful application of relatively new computer science to the simulation of relatively old physics models. But there's more to graphics than realistic image capture and rendering. Animation and interaction, for instance, are equally important, and we discuss these disciplines throughout many chapters in this book as well as address them explicitly in their own chapters. Why has success in the nonsimulation areas been so comparatively hard to achieve? Perhaps because these areas are more qualitative in nature and lack existing mathematical models like those provided by physics.

This book is not filled with recipes for implementing lots of ideas in computer graphics; instead, it provides a higher-level view of the subject, with the goal of teaching you ideas that will remain relevant long after particular implementations are no longer important. We believe that by synthesizing decades of research, we can elucidate principles that will help you in your study and use of computer graphics. You'll generally need to write your own implementations or find them elsewhere.

This is not, by any means, because we disparage such information or the books that provide it. We admire such work and learn from it. And we admire those who can synthesize it into a coherent and well-presented whole. With this in mind, we strongly recommend that as you read this book, you keep a copy of Haines, Möller, and Hoffman's book on real-time rendering [AMHH08] next to you. An alternative, but less good, approach is to take any particular topic that interests you and search the Internet for information about it. The mathematician Abel claimed that he managed to succeed in mathematics because he made a practice of reading the works of the masters rather than their students, and we advise that you follow his lead. The aforementioned real-time rendering book is written by masters of the subject, while a random web page may be written by anyone. We believe that

it's far better, if you want to grab something from the Internet, to grab the original paper on the subject.

Having promised principles, we offer two right away, courtesy of Michael Littman:

> ✓ **THE KNOW YOUR PROBLEM PRINCIPLE:** Know what problem you are solving.

> ✓ **THE APPROXIMATE THE SOLUTION PRINCIPLE:** Approximate the solution, not the problem.

Both are good guides for research in general, but for graphics in particular, where there are so many widely used approximations that it's sometimes easy to forget what the approximation is approximating, working with the unapproximated entity may lead to a far clearer path to a solution to your problem.

1.1.1 The World of Computer Graphics

The academic side of computer graphics is dominated by SIGGRAPH, the Association for Computing Machinery's Special Interest Group on Computer Graphics and Interactive Techniques; the annual SIGGRAPH conference is the premier venue for the presentation of new results in computer graphics, as well as a large commercial trade show and several colocated conferences in related areas. The SIGGRAPH proceedings, published by the ACM, are the most important reference works that a practitioner in the field can have. In recent years these have been published as an issue of the ACM Transactions on Graphics.

Computer graphics is also an industry, of course, and it has had an enormous impact in the areas of film, television, advertising, and games. It has also changed the way we look at information in medicine, architecture, industrial process control, network operations, and our day-to-day lives as we see weather maps and other information visualizations. Perhaps most significantly, the graphical user interfaces (GUIs) on our telephones, computers, automobile dashboards, and many home electronics devices are all enabled by computer graphics.

1.1.2 Current and Future Application Areas

Computer graphics has rapidly shifted from a novelty to an everyday phenomenon. Even throwaway devices, like the handheld digital games that parents give to children to keep them occupied on airplane trips, have graphical displays and interfaces. This corresponds to two phenomena: First visual perception is powerful, and visual communication is incredibly rapid, so designers of devices of all kinds want to use it, and second, the cost to manufacture computer-based devices is decreasing rapidly. (Roy Smith [Smi], discussing in the 1980s various claims that a GPS unit was so complex that it could never cost less than $1000, said, "Anything made of silicon will someday cost five dollars." It's a good rule of thumb.)

As graphics has become more prevalent, user expectations have risen. Video games display many millions of polygons per second, and special effects in films are now so good that they're no longer readily distinguishable from

non-computer-generated material. Digital cameras and digital video cameras give us huge streams of **pixels** (the individual items in an array of dots that constitutes the image[2]) to be processed, and the tools for processing them are rapidly evolving. At the same time, the increased power of computers has allowed the possibility of enriched forms of graphics. With the availability of digital photography, sophisticated scanners (Figure 1.2), and other tools, one no longer needs to explicitly create models of every object to be shown: Instead, one can scan the object directly, or even ignore the object altogether and use multiple digital images of it as a proxy for the thing itself. And with the enriched data streams, the possibility of extracting more and more information about the data—using techniques from computer vision, for instance—has begun to influence the possible applications of graphics. As an example, camera-based tracking technology lets body pose or gestures control games and other applications (Figure 1.3).

While graphics has had an enormous impact on the entertainment industry, its influence in other areas—science, engineering (including computer-aided design and manufacturing), medicine, desktop publishing, website design, communication, information handling, and analysis are just a few examples—continues to grow daily. And new interaction settings ranging from large to small form factors—virtual reality, room-size displays (Figure 1.4), wearable displays containing twin LCDs in front of the user's eyes, multitouch devices, including large-scale multitouch tables and walls (Figure 1.5), and smartphones—provide new opportunities for even greater impact.

For most of the remainder of this chapter, when we speak about graphics applications we'll have in mind applications such as video games, in which the most

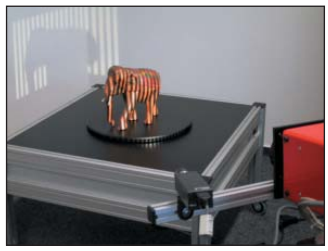


Figure 1.2: A scanner that projects stripes on a model that is slowly rotated on a turntable. The camera records the pattern of stripes in many positions to determine the object's shape. (Courtesy of Polygon Technology, GMBH).

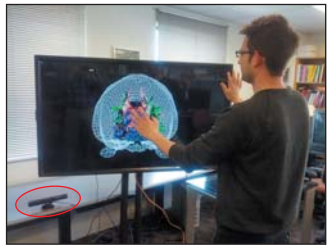


Figure 1.3: Microsoft's Kinect interface can sense the user's position and gestures, allowing a scientist to adjust the view of his data by "body language", without a mouse or keyboard. (Data view courtesy of David Laidlaw; image courtesy of Emanuel Zgraggen.)

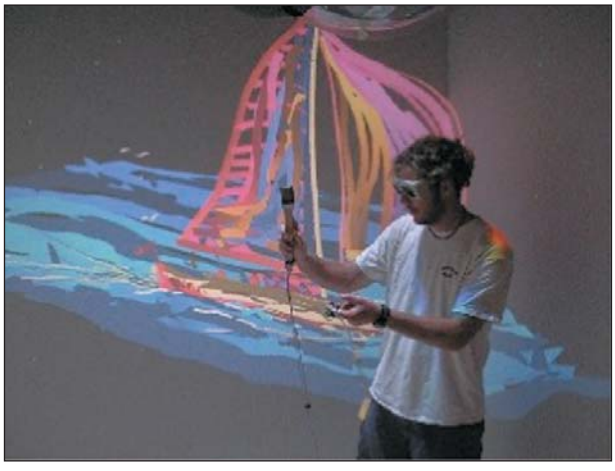


Figure 1.4: An artist stands in a Cave (a room whose walls are displays) and places paint strokes in 3D. The displays are synchronized with stereo glasses to display imagery so that it appears to float in midair in the room. Head-tracking technology allows the software to produce imagery that is correct for the user's position and viewing direction, even as the user shifts his point of view. (Courtesy of Daniel Keefe, University of Minnesota).

Figure 1.5: Two users interact with different portions of a large artwork on a large-scale touch-enabled display and a touch-enabled tablet display. (Courtesy of Brown Graphics Group.)

2. We'll call these **display pixels** to distinguish them from other uses of the term "pixel," which we'll introduce in later chapters.

critical resources are the processor time, memory, and bandwidth associated with **rendering**—causing certain objects or images to appear on the display. There is, however, a wide range of application types, each with its own set of requirements and critical resources (see Section 1.11). A useful measure of performance to keep in mind, therefore, is **primitives per second,** where a **primitive** is some building block appropriate to the application; for an arcade-like video game it might be textured polygons, while for a fluid-flow-visualization system it might be short colored arrows. The number of primitives displayed per second is the product of the number of primitives displayed per frame (i.e., the displayed image) and the number of frames displayed per second. While some applications may choose to display more primitives per frame, to do so they will need to reduce their frame rates; others, aiming at smoothness in the animation, will want higher frame rates, and to achieve them they may need to reduce the number of primitives displayed per frame (or, perhaps, reduce the complexity of each primitive by approximating it in some way).

1.1.3 User-Interface Considerations

The defining change in computer graphics over the past 30 years might appear to be the improvement in visual fidelity of both static and dynamic images, but equally important is the new *interactivity* of everyday computer graphics.[3] No longer do we just look at the pictures—we *interact* with them. Because of this, user interfaces (UIs) are increasingly important.

Indeed, the field of user interfaces has evolved in its own right and can no longer be considered a tiny portion of computer graphics, but the two remain closely integrated. Unfortunately, as of this writing, the state of commercial desktop UIs has not drastically changed from the research systems of a generation ago—input to the computer is still primarily through the keyboard and mouse, and much of what we do with the mouse consists of clicking on buttons, pointing to locations in text or images, or selecting menu items. And even though this point-and-click WIMP (windows, icons, menus, and pointers) interface has dominated for the past 30 years, high-quality and well-designed interfaces are rare, and interface design, at least in the early days, was too often an afterthought. Touch-based interfaces are a step forward, but many of them still mimic the WIMP interface in various ways. With increasing user sophistication and demands, interface design is now a significant part of the development of almost any application.

Two parallel and related trends required the commoditization of graphics hardware as well as enormous advances in software and CPU speed. The first was the quality and speed of image generation, making high-quality imagery part of everyday applications. The second was the development of the GUI, which has made computer applications so intuitive and easy to learn that even preliterate children can use them.

Why are interfaces so important? One reason is economics. In 1960, computers took up large rooms or small buildings; they cost millions of dollars and were shared by multiple users, each with a comparatively small salary. By 2000, computers were small and their costs were a fraction of the salary of the people using them. Figure 1.6 shows the trend of the dimensionless ratio of the salary of a user to the cost of the computer used. While in 1960 it was critical that the computer be used efficiently at all times, and users were obliged to do lots of things to make

3. Early graphics systems used in computer-aided design/computer-aided manufacturing (CAD/CAM) were often interactive at some level, but they were so expensive and complex that ordinary computer users never encountered them.

that happen, by 2000 the situation was entirely reversed: The user had become the precious resource while the computer was a relatively low-cost item. The UI is the place where user time is consumed, even in large and slow-running programs: Once the user sets the program running, he or she can do other things. Hence, we should concentrate more and more effort on interfaces and interaction.

What sorts of issues affect UI design? Many of them are related to psychology, perception, and the general area called human factors. It's one thing to use color in your UI; it's another to make sure the UI works for color-deficient users as well. It's one thing to have all necessary menu items present; it's another to order and group them so that a typical user can find what s/he is looking for quickly and select it easily: The menu items must be organized, and each item must be large enough to make the selection process easy. And it's still another thing to be certain that your UI is appropriate for whatever kind of device you might be using: a desktop machine, a smartphone, a PDA, or a video game controller.

Despite the importance of interfaces, we will not discuss them much; UI research is now its own field, related to graphics but no longer a part of it. In some cases, there are interface elements for which those with experience in graphics can offer particular insight. Chapter 21 discusses some of these as applications of the modeling and transformation technology developed earlier in the book.

From this discussion, it's clear that the goals of computer graphics are not purely based on physics or algorithms, but they depend critically on human beings. We don't merely compute the transfer of light energy in a scene; we must also consider the human perception of the results: Was the extra computation time used in a way that mattered to the viewer? We don't merely create an application program that provides functionality and performance that are appropriate for the some particular endeavor (e.g., playing music from a library or helping a physician maintain notes on patients); we also concern ourselves with whether the interface the program presents makes the program easy to use. Ease of use is obviously closely tied to human perception. We therefore present an introduction to perception in Chapter 5.

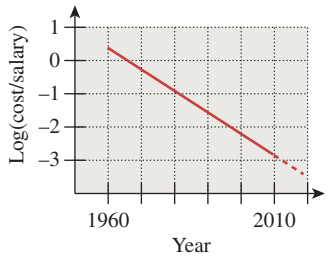


Figure 1.6: The log of the ratio between the cost of a computer and the salary of a person using the computer (roughly amortized for multiuser systems), plotted against the year.

1.2 A Brief History

Graphics research has followed a goal-directed path, but one in which the goal has continued to shift; the first researchers worked in a context of limited processor power, and thus they frequently made choices that got results as quickly and easily as possible. Early efforts were divided between trying to make drawings (e.g., blueprints) and trying to make pictures (e.g., photorealistic images). In each case, many assumptions were made, usually in concession to available processor power and display technologies. When a single display cost as much or more than an engineer's salary, every picture displayed had to have some value. When displaying a few hundred polygons took minutes, approximating curved surfaces with relatively few polygons made a lot of sense. And when processor speeds were measured in MIPS (millions of instructions per second) but images contained 250,000 or 500,000 pixels, one could not afford to perform a lot of computations per pixel. (In the 1960s and early 1970s, many institutions had at most a single graphical display!) Typical simplifying assumptions were that all objects reflected light more or less as flat latex paint does (although some more-sophisticated reflectance models were used in a few systems), that light either illuminated a surface directly or bounced around in the scene so often that it

eventually provided a general ambient light that illuminated things even when they weren't directly lit, and that the colors at the interior points of any triangle could be inferred from the colors computed at the triangle's vertices.

Gradually, richer and richer models—of shape, of light, and of reflectance—were added, but even today the dominant model for describing the light in a scene includes the term **"ambient,"** meaning a certain amount of light that's "all over the place in the scene" without any clear origin, ensuring that any object that's visible in the scene is at least somewhat illuminated. This ad hoc term was added to address aspects of light transport, such as interobject reflections, that could not be directly computed with 1960s computers; but it remains in use today. While many books follow the historical development of light transport, we'll choose a different approach and discuss the ideal (the physical simulation of light transport), how current algorithms approximate that ideal, how some earlier approaches did so as well, and how the vestiges of those approximations remain in common practice. The exception to this is that we'll introduce, in Chapter 6, a reflectance model that represents the scattering of light from a surface as a sum of three terms: **"diffuse,"** corresponding to light that's reflected equally in all directions; **"specular,"**[4] used to model more directional reflection, ranging from things like rough plastic all the way to the nearly perfect reflection of mirrors; and ambient. We will refine this model somewhat in Chapter 14, and then examine it in detail in Chapter 27. The advantage of the early look is that it allows you to experiment with modeling and rendering scenes early, even before you've learned how light is actually reflected.

Graphics displays have improved enormously over the years, with a shift from vector devices to **raster** devices—ones that display an array of small dots, for example, like CRTs or LCD displays—in the 1970s to 1980s, and with steadily but slowly increasing **resolution** (the smallness of the individual dots), size (the physical dimensions of the displays), and **dynamic range** (the ratio of the brightest to the dimmest possible pixel values) over the past 25 years. The performance of graphics processors has also progressed in accordance with Moore's Law (the rate of exponential improvement has been greater for graphics processors than for CPUs). Graphics processor architecture is also increasingly parallel; how far this can go is a matter of some speculation.

In both processors and displays, there have also been important leaps along with steady progress: The switch from vector devices to raster displays, and their rapid infiltration of the minicomputer and workstation market, was one of these. Another was the introduction of commodity graphics cards (and their associated software), which made it possible to write programs that ran on a wide variety of machines. At about the same time as raster displays became widely adopted another major change took place: the adoption of Xerox PARC's WIMP GUIs. This is when graphics moved from being a laboratory research instrument to being an unspoken component of everyday interaction with the computer.

One last leap is worth noting: the introduction of the *programmable* graphics card. Instead of sending polygons or images to a graphics card, an application could now send certain small *programs* describing how subsequent polygons and images were to be processed on their way to the display. These so-called "shaders"

4. The word "specular" has multiple meanings in graphics, from "mirrorlike" to "anywhere from sort of glossy to a perfect mirror." Aside from its use in Chapter 6 we'll use "specular" as a synonym for mirror reflection, and "glossy" for things that are shiny but not exactly mirrorlike.

opened up whole new realms of effects that could be generated without any additional CPU cycles (although the GPU—the Graphics Processing Unit—was working very hard!). We can anticipate further large leaps in graphics power in the next few decades.

1.3 An Illuminating Example

Let's now look at a simple scene and ask ourselves how we can make a picture of it.

A 100 W pinpoint lamp hangs above a table that's painted with gray latex paint at a height of 1 m, in an otherwise dark room. We look at the table from above, from 2 m away. What do we see? Regardless of the visible-light output of the lamp and the exact reflectivity of the surface, the pattern of illumination in the scene—brighter just beneath the lamp, dimmer as we move away—is determined by physics. We can do a thought experiment and imagine an ideal "picture" of this scene. And we can hope that a computer graphics system, asked to render a picture of this scene, would produce a result that would be a good approximation of this picture.

Nonetheless, it's difficult to write a conventional program with a standard graphics package to even display the general pattern of illumination. Most standard packages have no notion of units like "meters" or "grams" or "joules"; even their descriptions of light omit any mention of wavelength. Furthermore, conventional graphics packages compute the brightness of incoming light in a way that varies with the distance from the source. However, it does not vary as $1/d^2$, as we know it must from physics, but rather according to a different rule. To be fair, one *can* make the conventional package have a quadratic falloff, but the resultant picture still looks wrong.[5] That's in part because of nonlinearities in displays and the use of a small range of values (typically 0 to 255) to represent light amounts, together with the limited dynamic range of many displays (one cannot display very brightly lit or very dimly lit things faithfully). Using a linear falloff (often with a small quadratic term mixed in) partly compensates for these and results in a better-looking picture. But it's really just an ad hoc solution to a collection of other problems.

To correctly make a picture of the simple scene described above, it's probably best to model the physics directly and only then worry about the display of the resultant data. By the end of Chapter 32 you'll be able to do so.

In asking for a physically correct result in this example, we're examining a particular area of graphics—that of *realism*. It's remarkable that the quest for realism should have gone so very well in the early years of graphics, given the lack of any physical basis for most of the computations. This can be attributed to the remarkable robustness of the human visual system (HVS): When we present to the eyes anything that remotely resembles a physically realistic image, our visual system somehow makes sense of it. More recent trends in which captured imagery (e.g., digital photographs) are combined with graphics imagery have shown how important it can be to get things right: A mismatch in the brightness of real and synthetic objects is instantly noticeable.

5. The wrongness is not from the unfamiliarity of the point-light source; even if we made a graphical model of a larger-area light source, the results would be wrong.

But often in graphics we seek not a physical simulation but a way to present information visually (like a book or newspaper layout). In these cases, the typical viewing situation is a well-lit room, with light of approximately constant intensity arriving from all directions, and with the reflectance of the items on the page varying by a factor of perhaps 10^3. Simply setting the intensities of screen pixels to reasonable values that vary over a similar range works well, and there's no reason to do a physical simulation of the reflecting page. However, there may be a reason to be sure that what's displayed is faithful to the original (i.e., that the colors *you* see on your display are the same ones *I* see on mine); displays of fashion items or paint colors need to be accurate for users to understand how they really look.

Indeed, such a situation is a good opportunity for **abstraction,** which is a key element in visual communication in general: Because the physical characteristics of the document will not have a large impact on the viewer's experience as s/he encounters it, one can instead discuss the document in more abstract terms of shape and color and form. It's imperative, of course, that these abstractions capture what's important and leave out what's unimportant about whatever is being discussed; this is a key characteristic of the process of modeling, which we will return to frequently throughout this book.

1.4 Goals, Resources, and Appropriate Abstractions

The lightbulb example gives us another principle: In any simulation, first understand the underlying physical or mathematical processes (to the degree they're known), and then determine which approximations will best provide the results we need (our *goals*), given the constraints of time, processor power, and similar factors (our *resources*).

This approach applies both to 2D display graphics—the kinds of graphical objects found in the interface to your web browser, for instance, like the buttons that help you navigate and the display of the successive lines of text—and to 3D renderings used for special effects. In the former case, the dominant phenomena may not be those of physics but of perception and design, but they must still be understood. In addition to choosing a rich-enough abstraction, part of modeling wisely is choosing the right *representation* in which to work: To represent a real-valued function on a plane, you might use a rectangular array of values; divide the plane into triangular regions of various shapes and sizes, with values stored at the triangle vertices (this is common when making models of things like fluid flow); or use a data structure that stores the rectangular value array in such a way that whenever adjacent values agree they are merged into a larger "cell" so that detail is only present in the areas where the function is changing rapidly.

We summarize the preceding discussion in a principle:

✓ **THE WISE MODELING PRINCIPLE:** When modeling a phenomenon, understand the phenomenon you're modeling and your goal in modeling it, *then* choose a rich-enough abstraction, and *then* choose adequate representations to capture your abstraction within the bounds of your resources. Once this is done, *test* to verify that your abstraction was appropriate.

The testing will vary with the situation: If the design abstracts something about human perception, then the test may involve user studies; if the design abstracts something physical ("We can safely model small ocean waves with sinusoids"), then the test may be quantitative.

Barzel [Bar92] argues that most physical models for computer graphics come in three parts: the physical model itself, a mathematical model, and a numerical model. (As an example, the physical model might be that ocean waves are represented by vertical displacements of the water's surface, and their motion is governed solely by the forces arising from differences in nearby heights [rather than by wind, for example]; the mathematical model might be that these displacements are represented by time-dependent functions defined at integer points in some coordinate grid on the ocean's surface, with intermediate values being interpolated; and the computational model might be that the water's state one moment in the future can be determined from its state now by approximating all derivatives with "finite differences" and then solving a linearized version of the resultant equations.) Including this separation in your programs can help you debug them. This means, however, that during debugging, you must remember your model and its level of abstraction and the limitations these impose on your intended results. (In our example, the physical model itself says that you cannot hope to see breaking waves, while the mathematical model says that you cannot hope to see details of the water's surface at a scale smaller than the coordinate grid.) Within computer science, this is very unusual: In most other areas of computer science, you've got either a computational model or a machine model, and this single model provides your foundation. In graphics, we have physical, mathematical, numerical, computational, and perceptual models, all interacting with one another.

In both 2D and 3D graphics, it's critically important to consider the eventual goal of your work, which is usually *communication* in some form, and usually communication to a human. This end goal influences many things that we do, and should influence everything. (This is just a restatement of the "form follows function" dictum, as valuable in graphics as it is anywhere else.) As a simple example, consider how we treat light, which is just a kind of electromagnetic radiation: Because humans can only detect certain frequencies of light with their eyes, we usually don't worry about simulating radio waves or X-rays in graphics, even though the light emitted by conventional lamps (and the sun) includes many energies outside the visible spectrum. Hence, a limitation of the visual system becomes a computational savings for our programs. Similarly, because the eye's sensitivity to light energy is approximately logarithmic, we build our display hardware (to a first approximation) so that equal differences in pixel values correspond to equal *ratios* of displayed light energy.

✓ **THE VISUAL SYSTEM IMPACT PRINCIPLE:** Consider the impact of the human visual system on your problem and its models.

As another example, even in 2D display graphics there are perceptual issues to consider: The limits of human visual acuity tell us that the things we display must be of a certain size to be perceptually meaningful; at the same time, the limits of human motor control tell us that interaction must be designed in ways that fit those limits. We cannot ask a user to click on a sequence of pixels in a

1280×1024-pixel, 17-inch display with an ordinary mouse—clicking on a particular pixel is virtually impossible.

We don't mean to suggest that perception should influence every decision made in graphics; in Chapter 28 we'll see the risks that arise from treating light throughout the rendering process in a way that captures only our three-dimensional perception of color rather than the full spectral representation. However, in many situations where the range of brightness is small, the logarithmic nature of the eye's sensitivity is not particularly important, and common practice therefore often involves such things as averaging pixel values that represent log brightnesses; such techniques often serve their purposes admirably.

1.4.1 Deep Understanding versus Common Practice

Because computer graphics is actually in use all around us, we have to make concessions to common practice, which has generally evolved because it produced good-enough results at the time it was developed. But after a discussion of common practice, we'll often have a stand-back-and-look critique of it as well so that the reader can begin to understand the limitations of various approaches to graphics problems.

1.5 Some Numbers and Orders of Magnitude in Graphics

Because we will start our study of graphics with a discussion of light, it's useful to have a few rough figures characterizing the light encountered in ordinary scenes. Visible light, for instance, has a wavelength between approximately 400 and 700 nanometers (a nanometer is 10×10^{-9} m). A human hair has a diameter of about 10×10^{-4} m, so it's about 100 to 200 wavelengths thick, which helps give a human scale to the phenomena we're discussing.

1.5.1 Light Energy and Photon Arrival Rates

A single photon (the indivisible unit of light) has an energy E that varies with the wavelength λ according to

$$E = hc/\lambda, \tag{1.1}$$

where $h \approx 6.6 \times 10^{-34}\,\mathrm{J\,sec}$ is **Planck's constant** and $c \approx 3 \times 10^{8}\,\mathrm{m/sec}$ is the speed of light; multiplying, we get

$$E \approx \frac{1.98 \times 10^{-25}\,\mathrm{J\,m}}{\lambda}. \tag{1.2}$$

Using 650 nm as a typical photon wavelength, we get

$$E \approx \frac{1.98 \times 10^{-25}\,\mathrm{J\,m}}{650 \times 10^{-9}\,\mathrm{m}} \approx 3 \times 10^{-19}\,\mathrm{J} \tag{1.3}$$

as the energy of a typical photon.

An ordinary 100 Watt incandescent bulb consumes 100 W, or $100\,\mathrm{J/sec}$, but only a small fraction of that—perhaps 2% to 4% for the least efficient bulbs—is

converted to visible light. Dividing $2\,\mathrm{J/sec}$ by $3 \times 10^{-19}\,\mathrm{J}$, we see that such a bulb emits about 6.6×10^{18} visible photons per second. An office—say, $4\,\mathrm{m} \times 4\,\mathrm{m} \times 2.5\,\mathrm{m}$—together with some furniture has a surface area of very roughly $100\,\mathrm{m}^2 = 1 \times 10^6\,\mathrm{cm}^2$; thus, in such an office illuminated by a single 100 W bulb on the order of 10^{12} photons we arrive at a typical square centimeter of surface each second.

By contrast, direct sunlight provides roughly 1000 times this arrival rate; a bedroom illuminated by a small night-light has perhaps $1/100$ the arrival rate. Thus, the range of energies that reach the eye varies over many orders of magnitude. There is some evidence that the dark-adapted eye can detect a single photon (or perhaps a few photons). At any rate, the ratio between the daytime and nighttime energies of the light reaching the eye may approach 10^{10}.

1.5.2 Display Characteristics and Resolution of the Eye

Because we also work with computer displays, and the computers driving these displays typically draw polygons on the screen, it's valuable to have some numbers describing these. A typical 2010 display had between 1 million and 1.5 million pixels (individually controllable parts of the display[6])—which will soon grow to 4 million pixels; with displays that are 37 cm (about 15 inches) wide, the diagonal distance between pixel centers is on the order of 0.25 mm. The dynamic range of a typical monitor is about 500:1 (i.e., the brightest pixels emit 500 times the energy emitted by the darkest pixels). The display on a well-equipped 2010 desktop subtended an angle of about 25° at the viewer's eye.

The human eye has an angular resolution of about one minute of arc; this corresponds to about 300 mm at a 1 km distance, or (more practical for viewing computer screens) about 0.3 mm at a 1 m distance. When pixels get about half as large as they are now, it will be nearly impossible for the eye to distinguish them.[7] A one-pixel shift in a single character's position on a line of text may be completely unnoticeable. Furthermore, the eye's resolution far from the center of the view is much less, so pixel density at the edge of the display screen may well be wasted much of the time. On the other hand, the eye is very sensitive to motion, so two adjacent pixels in a gray region that alternately flash white may give an illusion of motion that's easily detectable, which might be useful for attracting the user's attention.

1.5.3 Digital Camera Characteristics

The lens of a modern consumer-grade digital camera has an area of about $0.1\,\mathrm{cm}^2$; suppose that we use it to photograph a typical 100 W incandescent bulb, filling the frame with the image of the bulb. To do so, we place the lens 10 cm from the

6. Each display part may actually consist of several pieces, as in a typical LCD display in which the red, green, and blue parts are three parallel vertical strips that make up a rectangle, or may be the result of a combination of multiple things, like the light emitted by the red, green, and blue phosphors of each triad of phosphors on a CRT screen.
7. This doesn't mean it won't be worth further reducing their size; 300 dot-per-inch (dpi) printers use dots that are about 0.1 mm, and their quality is noticeably poorer than that of 1200 dpi printers, even when viewed at a distance of a half-meter. Distinguishing between adjacent pixels and detecting the smoothness of an overall image are evidently rather different tasks.

Figure 1.7: The standard teapot, created by Martin Newell, a model that's been used thousands of times in graphics.

bulb. The surface area of a sphere with radius 10 cm is about 1200 cm^2; our lens therefore receives about 1/10,000 of the light emitted by the bulb, or 6.6×10^{14} photons per second. If we take the picture with a 0.01 sec exposure and we have an approximately 1-million-pixel sensor, then each sensor pixel receives about 10^6 photons. Photographing a dark piece of carpet in our imaginary office above might result in each sensor pixel receiving only 100 photons.

1.5.4 Processing Demands of Complex Applications

Computer games are some of the most demanding applications at present; to make objects appear on the user's screen, these applications send polygons to a graphics processor. These polygons have various attributes (like color, texture, and transparency) and are displayed with various technologies (antialiasing, smooth shading, and others, all of which we'll discuss in detail later). For a polygon to be displayed, certain pixels must be colored in certain ways. Thus, **polygon rate** (the number of polygons displayed per second) and **fill rate** (the number of pixels colored per second) are both used to measure performance. The numbers are constantly changing, and there's a huge difference between a textured, antialiased, transparent polygon covering 500 pixels and a flat-shaded 10-pixel triangle, so comparisons are difficult. But complex scenes for interactive display can easily contain 1 million polygons, of which maybe 100,000 are visible (the others being hidden by things in front of them or outside the field of view), each occupying perhaps 10 pixels on average. In many cases, a single polygon occupies less than a single pixel. This happens in part because complex shapes are often modeled with polygonal meshes (see Figure 1.7). For high-quality, noninteractive, special-effects production, the resolution of the final image may be considerably higher, but at the same time, scenes can contain many millions of polygons; the "polygon is smaller than a pixel" rule of thumb continues to apply.

1.6 The Graphics Pipeline

The functioning of a standard graphics system is typically described by an abstraction called the **graphics pipeline.** The term "pipeline" is used because the transformation from mathematical model to pixels on the screen involves multiple steps, and in a typical architecture, these are performed in sequence; the results

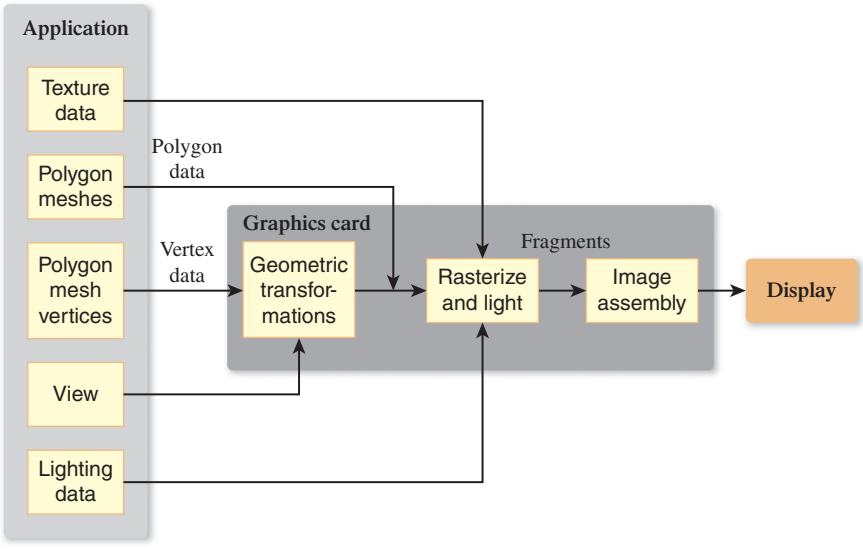


Figure 1.8: The graphics pipeline, version 1.

of one stage are pushed on to the next stage so that the first stage can begin processing the next polygon immediately.

Figure 1.8 shows a simplified view of this pipeline: Data about the scene being displayed enters at various points to produce output pixels.

For many purposes, the exact details of the pipeline do not matter; one can regard the pipeline as a black box that transforms a geometric model of a scene and produces a pixel-based perspective drawing of those polygons. (Parallel-projection drawings are also possible, but we'll ignore these for the moment.) On the other hand, some understanding of the nature of the processing is valuable, especially in cases where efficiency is important. The details of the boxes in the pipeline will be revealed throughout the book.

Even with this simple black box you can write a great many useful programs, ignoring all physical considerations and treating the transformation from model to image as being defined by the black box rather than by physics (like the non-quadratic light-intensity falloff mentioned above).

The past decade has, to some degree, made the pipeline shown above obsolete. While graphics application programming interfaces (APIs) of the past provided useful ways to adjust the parameters of each stage of the pipeline, this fixed-function pipeline model is rapidly being superseded in many contexts. Instead, the stages of the pipeline, and in some cases the entire pipeline, are being replaced by programs called shaders. It's easy to write a small shader that mimics what the fixed-function pipeline used to do, but modern shaders have grown increasingly complex, and they do many things that were impossible to do on the graphics card previously. Nonetheless, the fixed-function pipeline makes a good conceptual framework onto which to add variations, which is how many shaders are in fact created.

1.6.1 Texture Mapping and Approximation

One standard component of the black box is the **texture map.** With texture mapping, we take a polygon (or a collection of polygons) and assign a color to each point via a lookup in a texture image; the technique is a little like applying a stencil

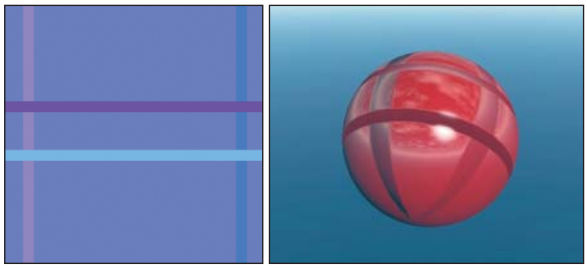


Figure 1.9: (a) The image at left depicts a normal map. Each image point has x- and y-coordinates that correspond to the latitude and longitude of a point on the sphere. The RGB color triple stored at each point determines how much to tilt the normal vector at the corresponding point of the sphere. The pale purple color indicates no tilt, while the four stripes tilt the normal vector up and down or left and right. (b) The resultant shape, which looks bumpy; you can tell it's actually smooth by looking at the silhouette. Note that it has also been "color textured" with a reflected sky.

to a surface or gluing a decal onto an object. You can think of the texture image, which can be a piece of artwork scanned into the system, a photo taken with a digital camera, or an image created in a paint program, for instance, as a rubber sheet with a picture on it. The texture coordinates describe how this sheet is stretched and deformed to cover some part of the object.

The idea of using a texture to modify the color characteristics of each point of an image is only one of many applications of texture mapping. The central ideas of texture mapping have been generalized and applied to many surface properties. The appearance of a surface, for instance, depends in part on the surface's **normal vector** (or **normal**), which is the vector that's perpendicular to the surface at each point. This normal vector is used to compute how light reflects from the surface. Since the surface is typically represented by a mesh of polygons, these surface normal vectors are usually computed at the polygon vertices and then interpolated over the interior of the polygon to give a smooth (rather than faceted) appearance to the shape.

If instead of using the *true* normal to a surface (or its approximation by interpolation as above) we use a substantially different one at different points of each polygon, the surface will have a different appearance at different points, appearing to tilt more toward or away from us, for instance. If we apply this idea across a whole surface we can generate what seems to be a lumpy surface (see Figure 1.9), while the underlying shape is actually nearly smooth.

The surface appears to have lots of geometric variation even though it's actually spherical. Unfortunately, near the silhouette of the surface the unvarying nature is evident; this is a common limitation of such mapping tricks. On the other hand, being able to draw just a few normal-mapped polygons instead of thousands of individual ones can be enough of an advantage to make this choice appropriate. This kind of choice is commonplace in graphics—one must decide between physical correctness (which might require huge models) and approximately correct imagery made with smaller models. If model size and processing time constitute a significant portion of your engineering budget, these are the sorts of tradeoffs you have to make.

1.6.2 The More Detailed Graphics Pipeline

As we said above, a pipeline architecture lets us process many things simultaneously: Each stage of the pipeline performs some task on a piece of data and hands the result to the next stage; the original pipeline stage can then begin performing

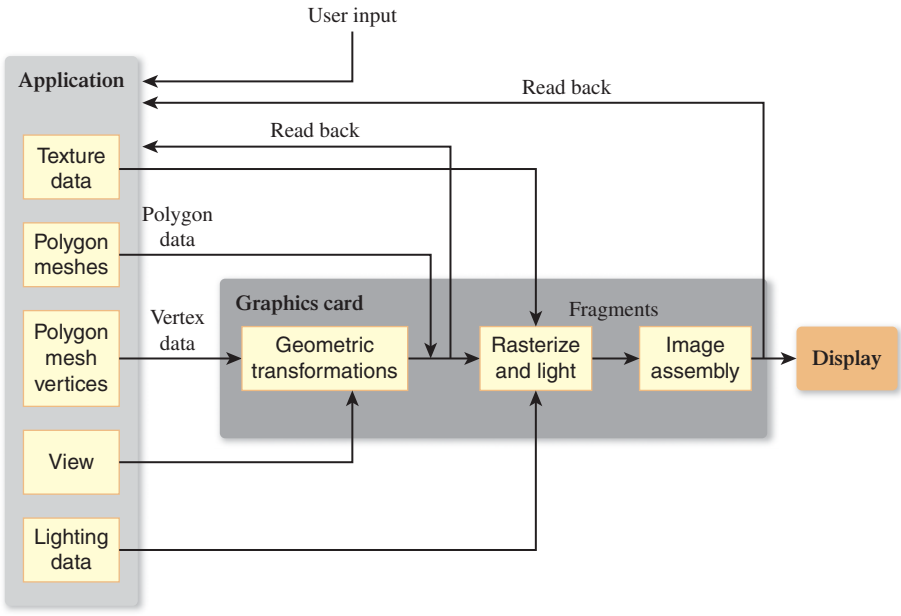


Figure 1.10: In this depiction of a larger graphics pipeline, the application program performs some work (e.g., animation) to determine the geometry to be displayed; this geometric description is handed to the graphics pipelines; the resultant image is displayed. At the same time, user input, in response to the displayed image, may affect the next operations of the application program, as may data read back from the graphics pipeline itself.

the task on the next piece of data. When such a pipeline is properly designed this can result in improved throughput, although as stages are added, the total amount of time it takes for an input datum to produce a result continues to increase. In systems where interactive performance is critical, this **lag** or **latency** can be important.

The graphics pipeline consists of four main parts: vertex geometry processing and transformation, triangle processing (through rasterization) and fragment generation, texturing and lighting, and fragment-combination operations for assembling the final image, all of which we'll summarize presently (and which are covered in more detail in Chapters 15 and 38). You can think of this pipeline as part of a larger pipeline that captures the structure of a typical program (see Figure 1.10, in which the vertex processing portion is labeled "Geometric transformation"; the fragment generation, texturing, and lighting are collected into a single box; and the portion representing the final processing of fragments is labeled "Image assembly").

In this larger pipeline, an application program generates data to be displayed, and the graphics pipeline displays it. But there may be user input (possibly in response to the displayed images) that controls the application, as well as information read back from the graphics pipeline and also used in the computation of the next image to be shown.

Each part of the graphics pipeline may involve several tasks, all performed sequentially. The exact implementations of these tasks may vary, but the user of such a system can still regard them as sequential; graphics programmers should have this abstraction in mind while creating an application. This **programmer's model** is the one provided by most APIs that are used to control the graphics

pipeline. In actual practice (see Chapter 38), the exact order of the tasks within the parts (or even the parts to which they are allocated) may be altered, but a graphics system is required to produce results *as if* they were processed in the order described. Thus, the pipeline is an abstraction—a way to think about the work being done; regardless of the underlying implementation, the pipeline allows us to know what the results will be.

The vertex geometry part of the pipeline is responsible for taking a geometric description of an object, typically expressed in terms of the locations of certain vertices of a polygonal mesh (which you can think of informally as an arrangement of polygons sharing vertices and edges to cover an object, i.e., to approximate its surface), together with certain transformations to be applied to these vertices, and computing the actual positions of the vertices after they've been transformed. The polygons of the mesh, which are defined in terms of the vertices, are thus implicitly transformed as well.

The triangle-processing stage takes the polygons of the mesh—most often triangles—and a specification for a virtual camera whose view we are rendering, and processes the polygons one by one in a process called **rasterization,** to convert them from a continuous-geometry representation (triangle) into the discrete geometry of the pixelized display (the collection of pixels [or portions of pixels] that this triangle contains).

The resultant **fragments** (pixels or portions of pixels that belong to the triangle and may eventually appear on the display if they're not obscured by some other fragment) are then assigned colors based on the lighting in the scene, the textures (e.g., a leopard's spots) that have been assigned to the mesh, etc.

If several fragments are associated to the same pixel location, the frontmost fragment (the one closest to the viewer) is generally chosen to be drawn, although other operations can be performed on a per-pixel basis (e.g., transparency computations, or "masking" so that only certain fragments get "drawn," while others that are masked are left unchanged).[8]

In modern systems, all of this work is usually done on one or more Graphics Processing Units (GPUs), often residing on a separate graphics card that's plugged into the computer's communication bus. These GPUs have a somewhat idiosyncratic architecture, specially designed to support rapid and deep pipelining of the graphics pipeline; they have also become so powerful that some programmers have started treating them as coprocessors and using them to perform computations unrelated to graphics. This idea—having a separate graphics unit that eventually becomes so powerful that it gets used as a (nongraphics) coprocessor—is an old one and has been reinvented multiple times since the 1960s. In early generations, this coprocessor was typically moved closer and closer to the CPU (e.g., sharing memory with the CPU) and grew increasingly powerful until it became so much a part of the CPU that designers began creating a *new* graphics processor that was closely associated to the display; this was called the **wheel of reincarnation** in a historically important paper by Myer and Sutherland [MS68]. The notion may be slightly misleading, however, as observed by Whitted [Whi10]: "We

8. Note that the choice of a representation by a raster grid implies something about the final results: The information in the result is limited! You cannot "zoom in" to see more detail in a single pixel. But sometimes in computing, what should be displayed in a single pixel requires working with subpixel accuracy to get a satisfactory result. We'll frequently encounter this tension between the "natural" resolution at which to work (the pixel) and the need to sometimes do subpixel computations.

sometimes forget that the famous 'wheel of reincarnation' translates as it rotates, transporting us to unfamiliar technological territory even if we recognize historical similarities."

1.7 Relationship of Graphics to Art, Design, and Perception

The simple lamp at the top of Figure 1.11 conveys both a shape and a design style in just a few strokes. Henri Matisse's "Face of a Woman," shown at the bottom of Figure 1.11, contains no more than 13 pen strokes but is nonetheless able to convey an enormous amount to a human viewer; it's far more recognizable as a face than many of the best contemporary face renderings in graphics. This is partly because of the **uncanny valley**—an idea from robotics [Mor70] that states that as robots got increasingly humanlike, a viewer's sense of familiarity would increase to a point, but then it would drop precipitously until the robot was *very* humanlike, at which point the familiarity would rise rapidly above its previous level. The uncanny valley is the region in which familiarity is low but human resemblance is high. In the same way, graphics images of humans that are "almost right" are often described as "creepy" or "weird." But ignoring this for a moment, there's another important difference: Matisse's drawing is simple, whereas an enormous amount of computational effort is expended in making a realistic face rendering. This is because artists and designers have reverse-engineered the human visual system to get the greatest effect for the least amount of "drawing budget." Looking at their work helps us understand that the goal of all graphics is *communication,* and that sometimes this is best achieved not with realism but with other means. Auto-repair manuals, for instance, can be illustrated with photos, but the top-quality manuals are instead illustrated with drawings (see Figure 1.12) that emphasize important details and elide other details. Which details are important? That depends on the intent of the person creating the image and on the human visual system. We know, for instance, that the human visual system is sensitive to sharp transitions in brightness and is somewhat more sensitive to vertical and horizontal lines than to diagonal ones; this partly explains why line drawings are effective, and why one can afford to leave out diagonal lines preferentially over verticals and horizontals.

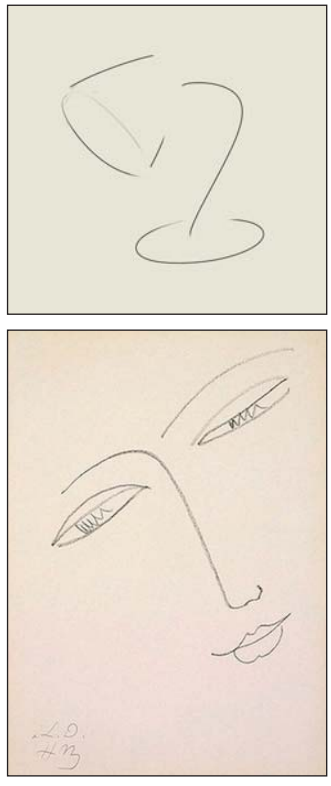


Figure 1.11: The lamp, courtesy of Jack Hughes, has just five strokes. Matisse's "Face of a Woman" depicts both shape and mood in just 13 strokes.

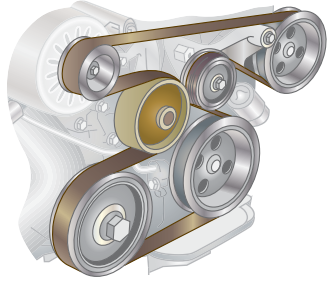


Figure 1.12: A repair manual shows details where needed, but omits unnecessary material.

In every engineering problem, there's a budget; graphics is no different. You are limited in graphics by things like the number of polygons you can send to the pipeline before you have to draw the next frame to display, the number of pixels that can be filled, and the amount of computation you can afford to do in the CPU to decide what polygons you want to draw in the first place. Artists who are drawing something have a similar budget: the amount of effort spent in placing marks on a page, the time before the scene being rendered changes (you can't paint a sunset-in-progress at midnight), etc. They've developed techniques that allow them to convey a scene on a low budget: For instance, contour drawings work well, and flat fill-in color adds contrast that helps separate individual objects, etc. We can learn from the artists' reverse engineering of the human visual system and use their techniques to render more efficiently. And because most computer-generated images are intended to be viewed by a human, the human brain is the ultimate measurement tool for what's satisfactory. There's another budget to consider as well: the viewer's attention. Graphics is also limited by the time and effort

Figure 1.13: Each strip is a single color, but the left side of each strip looks a little brighter and the right side looks a little darker, which has the effect of accentuating the dividing line between the strips; this effect is known as Mach banding.

the human viewer can be expected to spend to understand what is being communicated. Of course, our standard for satisfaction varies with time: The images produced in the 1960s and 1970s seemed amazing at the time, but are completely unsatisfactory by today's standards.

On the other hand, sometimes the nature of the visual system lets us make very effective but simple approximations of reality that are entirely convincing; early cloud models [Gar85] used extremely simple approximations of cloud shapes very effectively, because the eye is not terribly sensitive to the geometry of a cumulus cloud, as long as it looks fluffy. But all too often, such simplifications fail badly. For instance, we could attempt to make a mesh appear to have a smoothly changing color by filling each triangle with its own color (**flat shading**) and then making the individual triangles small so that the changes from one triangle to the next are tiny. Unfortunately, unless the triangles are *very* tiny, this leads to something called **Mach banding** (see Figure 1.13), which is extremely distracting to the eye.

1.8 Basic Graphics Systems

A modern graphics system consists of a few interaction devices (keyboard, mouse, perhaps a tablet or touch screen), a CPU, a GPU, and a display. Today's displays are either liquid-crystal displays (LCDs) or cathode-ray tube (CRT) displays, although new technologies like plasma displays and OLEDs (organic light-emitting diodes) are constantly changing the landscape. Each displays a rectangular array of pixels, or regions that can be lit to varying degrees in varying colors by the control of three colored parts, typically red, green, and blue. In the case of a CRT, when a single pixel is turned on it produces a glowing, approximately circular area containing an RGB triad of phosphors on the screen, an area that is bright in the center and rapidly fades at the edges so that the bright areas of adjacent pixels overlap only a little. In the case of an LCD, there is a backlight behind the screen, and each pixel is a set of three small rectangles that allow some amount of the backlight in the red, green, or blue spectrum to pass through to the viewer. There is a very small space between the pixels (like the grout on a tile floor), but for most purposes we can treat the LCD pixels as completely covering the screen. The brightness of each pixel (on either type of display) can be controlled by a program; we can also assume, except in the most rigorous situations, that all pixels are capable of displaying the same brightnesses, and that there is no

substantial variation of their apparent brightness with position (i.e., pixels at the display's edge look just as bright as those at the center when they're "turned on" to the same degree).

A typical graphics program runs on the CPU, processing input from the UI devices and sending instructions to the GPU describing what should be displayed; this, in turn, prompts further user interaction, and the cycle continues. In almost all cases, this structure is provided by a graphics platform that serves as an intermediary between a graphics application and the hardware, but for now, let's consider the simple case where we're building a basic graphics program from scratch. Frequently the display is steadily changing (e.g., it is being updated every 1/30 of a second), and user input may come only occasionally. The simplest model for the application program is to issue for each redisplay cycle new display instructions to the GPU, often resulting in a frame rate that's typically 15 to 75 frames per second. Too-low frame rates can severely degrade the quality of interaction, as can too-great latency (the time between an action—be it a user-initiated click, or the initiation of a frame redisplay—and its effect), so this simple model must be used with caution.

1.8.1 Graphics Data

Typically graphical models are created in some convenient coordinate system; a cube that is to be used as one of a pair of dice might be modeled as a unit cube, centered at the origin in 3-space, with all x-, y-, and z-coordinates between -0.5 and 0.5. This coordinate system is called **modeling space** or **object space.**

This cube is then placed in a **scene**—a model of a collection of objects and light sources. Perhaps the dice are on a table that's six units tall in y; in the scene description, they're moved there by applying some transformation to the coordinates of all the vertices (the corners) of the cube. In the case of the die, perhaps all six vertices have 6.5 added to their y-coordinates so that the bottom of the die sits on the top of the table. The resultant coordinates are said to be in **world space** (see Figure 1.14). (Chapter 2 describes an example of this modeling process in great detail.)

The location and direction of a virtual camera is also given in world space, as are the positions and physical characteristics of virtual lights. Consider a set of coordinate axes (see Figure 1.15) whose origin is at the center of the virtual

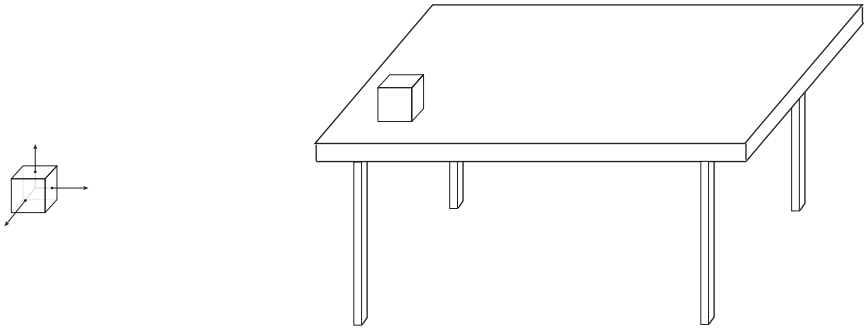


Figure 1.14: On the left, a die is centered on its own axes in modeling coordinates. The same die is placed in the world (on the right) by adding 6.5 to each y-coordinate (the y-direction points "up") to get world coordinates for the die.

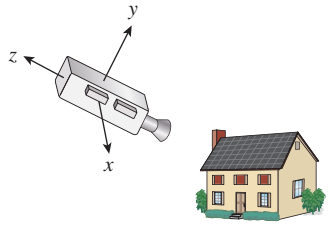


Figure 1.15: The virtual camera looks at a scene from a specified location, and with some orientation or attitude. We can create a coordinate system whose origin is at the center of the camera, whose z-axis points opposite the view direction, and whose x- and y-axes point to the right and to the top of the camera, respectively. The coordinates of points in this coordinate system are called camera coordinates.

camera, whose x-axis goes to the right side of the camera (as seen from the back), whose y-axis points up along the back of the camera, and whose negative z-axis points along the camera view. All objects in world space have coordinates in this coordinate system as well; these coordinates are called **camera-space coordinates** or simply **camera coordinates.** Computing these camera-space coordinates from world coordinates is relatively simple (Chapter 13) and is one of the services typically provided by a graphics platform.

These camera coordinates are transformed into **normalized device coordinates,** in which the visible objects have floating-point xy-values between -1 and 1, and whose z-coordinate is nonpositive. (Objects with xy-values outside this range are outside the camera's field of view; objects with $z > 0$ are behind the camera rather than in front of it.) Finally, the visible fragments are transformed to **pixel coordinates,** which are integers (with $(0, 0)$ being the upper-left corner of the display and $(1280, 1024)$ being the lower-right corner of the display) by scaling and rounding the xy-coordinates. These resultant numbers are sometimes said to be coordinates in **image space.** Returning to the cube that's to be used as one of a pair of dice, we want each side of the cube to look like the side of a die. To do this, we might use a texture map containing a picture of each side of a die. The vertices[9] of each face of the cube will then also be given texture coordinates indicating what portion of the texture should be applied to them (see Figure 1.16).

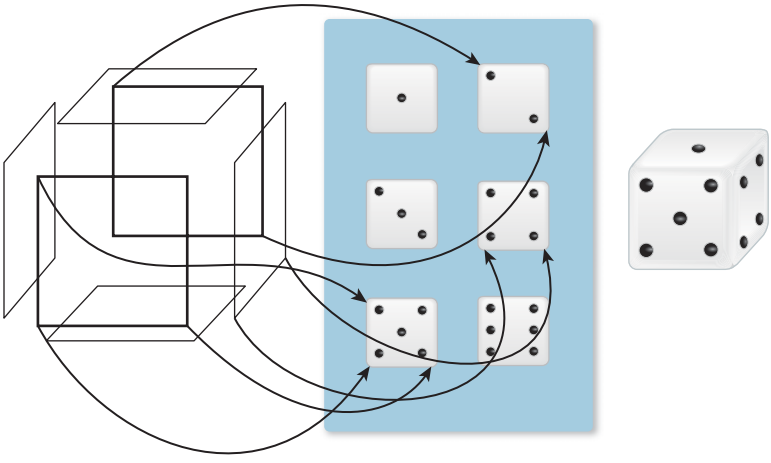


Figure 1.16: The vertices of each of the six faces of the die (shown in an exploded view) are assigned texture coordinates (a few are indicated by the arrows in the diagram); the texture image is then used to determine the appearance of each face of the die, as if the texture were a rubber sheet stretched onto the face. Note that a single 3D location may have many texture coordinates associated to it, because it is part of many different faces. In the case of the die, this is moot, because all instances of the 3D point get the same texture color assigned. Chapter 20 discusses topics like this at greater length. The resultant textured die is shown on the right.

9. The word "vertices" (VERT-uh-sees) is the plural of "vertex," although "vertexes" is sometimes used. Occasionally our students mistakenly back-construct the singular "vertice." Please avoid this. Other similarly formed plurals are index–indices and simplex–simplices.

The various conversions from the continuous geometry of Euclidean space to the rasterized geometry of the screen (with rasterized textures being used along the way) involve many subtleties, to be discussed in Chapter 18.

1.9 Polygon Drawing As a Black Box

Given the difficulties of carrying out the steps in the pipeline (especially those that involve the transformation from continuous to discrete geometry), we can, for the time being, treat polygon drawing as a black box: We have a graphics system which, when told to draw a polygon, somehow makes the right pixels on the display be illuminated with the right colors. This black-box approach will let us experiment with interaction, color, and coordinate systems. We'll then return to the details in later chapters.

1.10 Interaction in Graphics Systems

Graphics programs that display images in some form typically feature some level of user interaction as well. For example, in many programs the user clicks on things with the mouse, selects menu items, and types at the keyboard. However, the level of interaction in some programs (indeed, in many 3D games) is far more complex.

Graphics programs typically support such interaction by having two parallel threads of execution; one thread handles the main program and the other handles the GUI. Each component of the GUI—button, checkbox, slider, etc.—is associated with a **callback procedure** in the main program. For instance, when the user clicks a button the GUI thread calls the button's callback procedure. That procedure in turn may alter some data, and may also ask the GUI to change something.

As an example, imagine a trivial game in which a user has to guess a number that the computer has chosen—either one, two, or three. To do so, the user clicks on one of three buttons. If the user clicks on the correct button, the display reads "You win!"; if not, it reads "Try again." In this scenario, when the user clicks button 2 but the secret number is 1, the button-2 callback does the following.

1. It checks to see whether 2 was the secret number.
2. Because 2 is not the secret number, it asks the GUI to display the "try again" message.
3. It asks the GUI to gray out (disable) the "2" button so that the user is not able to guess the same wrong answer more than once.

Of course, the button-1 and button-3 callbacks would be very similar, and in each case, if the guess was correct the button would ask the GUI to display the fact that the user had won.

For more complex programs, the structure of the callbacks can be far more complex, of course, but the general idea is this simple one. One speaks of the code in the callback as the button's "behavior"; thus interaction components have both *appearance* and *behavior*. Not surprisingly, many successful interfaces correlate the two—the behavior of a component can, to some extent, be inferred by the user

who is confronted with its appearance. (The simplest example of this is that of a button with text on it. A Quit button should, when clicked, cause the program [or some action] to quit!)

The entire matter of scheduling the GUI thread and the application thread is typically handled by a graphics framework, via the operating system, in a way that's usually completely transparent to the programmer.

1.11 Different Kinds of Graphics Applications

A wide variety of applications use computer graphics, and many different characteristics determine the overall characteristics of these applications. With the current explosion of applications and application areas, it's impossible to classify them all. Instead, we will examine how these applications differ.

The following are some of the relevant criteria.

- Is the display changing on every refresh cycle (typical of many computer games) or changing fairly rarely (typical of word processors)?
- Are the coordinates used by the program described by an abstraction in which they're treated as floating-point numbers in programs (as in many games), or are individual pixel coordinates the defining way to measure positions (as in certain early paint programs)?
- Usually a model of the data is being displayed; is the transformation from this view to the display described in terms of a camera model (typical of 3D games) or something different (like the viewable portion of a text document that one sees in a word processing program)? In each case, there's a need to **clip** (not display) the part of the data that lies outside some rectangle on the display.
- Are objects being displayed with associated behaviors? The buttons and menus on a GUI are such objects; the pictures of the "bad guys" in a video game typically are not. (Clicking on a bad guy has no effect. Shooting a gun at the bad guy may kill him, but this is a separate kind of interaction, based on the game logic rather than on the interaction behavior of displayed objects.)
- Is the display trying to present a physically realistic representation of an object, or is it presenting an abstract representation of the object? A tool for creating schematic diagrams of electronic circuits does not aim to show how those diagrams, if printed on paper and viewed in a sunlit office, would appear. Instead, it presents an abstract view of the diagrams, in which all lines are equally dark and all parts of the background are equally light, and the lightness/darkness of each is a user-determined property rather than a result of some physical simulation. By contrast, the displays in 3D computer games often aim for photorealism, although some now aim for deliberately nonphotorealistic effects to convey mood.

Less critical, but still important, are the following.

- Do the abstract floating-point coordinates have units (feet, centimeters, etc.), or are they simply numbers? One advantage of having units is that a single program can adapt itself by determining what sort of display is being used—a 19-inch desktop display or a 1.5-inch cellphone display. The

desktop display of, say, driving directions might show the entire route, while the cellphone display might show a scrollable and zoomable small portion. Since display pixel sizes vary widely, physical units make more sense than pixel counts in many cases.

- Does the graphics platform handle updates via a changing model? If the platform has you update a model of what is to be displayed and then automatically updates the display whenever necessary, querying that model as needed, the programming demands are relatively simple but the way in which updates are handled may be beyond your control. A system that does *not* provide such updating would, for example, require the application to do "damage repair" when movement of overlapped windows reveals new areas to be displayed. Programs in which screen display can be very expensive (some image-editing programs are like this) prefer to handle damage repair themselves so that when a user moves a window in which an image is displayed, the newly revealed parts are only filled in occasionally during the move, since constantly filling in the parts could make the move too slow for comfortable use.

Many 2D graphics fall into the category in which there is little physical realism, most of the objects displayed have associated behaviors, and the display is updated relatively infrequently. Much of 2.5D graphics applications, in which one works with multiple 2D objects that are "stacked one on top of the other" (the layers in many image-editing programs fit this model), also produce imagery that is far from realistic. The cost of updating the display may become a critical resource in some of these programs. By contrast, many 3D graphics applications rely on simulation and realism, and objects in 3D scenes tend to have less "behavior" in the sense of "reactions to interactions with devices like the mouse or keyboard," although this is rapidly changing.

Not surprisingly, the different requirements of 2D, 2.5D, and 3D programs means that there is no one best answer to many questions in graphics. The circuit-design program doesn't need physically realistic rendering capability, just as the twitch game doesn't typically need much of an interaction-component hierarchy.

1.12 Different Kinds of Graphics Packages

The programmer who sets out to write a graphics program has a wide choice of starting points. Because graphics cards—the hardware that generates data to be displayed on a screen—or their equivalent chipsets vary widely from one machine to the next, it's typical to use some kind of software abstraction of the capabilities of the graphics card. This abstraction is known as an **application programming interface** or **API.** A graphics API could be as simple as a single function that lets you set the colors of individual pixels on the display (although in practice this functionality is usually included as a tiny part of a more general API), or it could be as complex as a system in which the programmer describes a scene consisting of high-level objects and their properties, light sources and their properties, and cameras and their properties via the API, and the objects in the scene are then rendered as if they were illuminated by the light sources and seen from the particular cameras. Often such high-level APIs are just a part of a larger system for application development, such as modern game engines, which may also provide

features like physical simulation, artificial intelligence for characters, and systems for adapting display quality to maintain frame-rates.

A range of software systems are available to assist graphics programs, from simple APIs that give fairly direct access to the hardware all the way to more complex systems that handle all interaction, display refresh, and model representation. These can reasonably be called "graphics platforms," a term we've been using somewhat vaguely until now. The variety of systems and their features are the subject of Chapter 16.

1.13 Building Blocks for Realistic Rendering: A Brief Overview

When you want to go from models of reality to the creation, in the user's mind, of the illusion of seeing something in particular, you have to have the following:

- An understanding of the physics of light
- A model for the materials with which light interacts, and for the process of interaction
- A model for the way we capture light (with either a real or a virtual camera, or with the human eye) to create an image
- An understanding of how modern display technology produces light
- An understanding of the human visual system and how it perceives incoming light
- And an understanding of a substantial amount of mathematics used in the description of many of these things

The difficulty with a bottom-up approach to this material is that you have to learn a great deal before you make your first picture; many reasonable students will ask, "Why don't I just grab something from the Web, run it, and then start tinkering until I get what I want?" (The answer is "You can do that, but it will probably take longer for you to get to the end result than if you try to have some understanding first.") As authors, we have to contend with this tension. Our approach is to tell you a few basic things about each of the items above—enough so that you know, as you start making your first pictures, which things you're doing are approximations and which are correct—and then take you through some very effective approximate approaches to making pictures. Only then do we return to the higher-level goal of understanding the ideal and how we might approach it.

1.13.1 Light

Chapter 26 describes the physics of light in considerable detail. Right now, we rely on your intuitive understanding of light and lay out some basic principles that we'll refine in later chapters.

- Light propagates along straight-line rays in empty space, stopping when it meets a surface.
- Light bounces like a billiard ball from any shiny surface that it meets, following an "angle of incidence equals angle of reflection" model, or is absorbed by the surface, or some combination of the two (e.g., 40% absorbed, 60% reflected).

- Most apparently smooth surfaces, like the surface of a piece of chalk, are microscopically rough. These behave as if they were made of many tiny, smooth facets, each following the previous rule; as a result, light hitting such a surface scatters in many directions (or is absorbed, as in the mirror-reflection case mentioned in the preceding bulleted item).
- A pinhole in a flat sheet of material admits a bundle of light rays, all of which pass through or very near to the center of the pinhole.
- A pixel of a camera, or one of the cells in the eye that detects light, sums up (by integration) all the light that arrives at a small area over a small period of time. The value of the integral is the sensor response that corresponds to how much total light, based on the number of incident photons, the pixel (or cell) "saw."
- A pixel of a display can be adjusted to emit light of a specified intensity and color.

That's it! This is enough of a model of light to produce very realistic pictures. Each of the bulleted items above is only approximately correct, but each is correct enough for a great many purposes. With them in hand, three big challenges remain. First, we need some data structures for representing the surfaces, camera, and lights in a scene. Second, we need an algorithm for evaluating all of the light bounces and integration. Third, and most important, both the data structures and the algorithm have to be efficient. Nature uses about 10^{21} photons per square meter per second to produce images of scenes lit by the sun; even if computers were a billion times more powerful than they are today, we still couldn't afford to write loops or data structures that actually simulate the motion of every single photon.

1.13.2 Objects and Materials

Our initial assumption about objects is that they are composed of materials that either reflect or absorb light (or do both, in varying amounts) at their surfaces. We assume that air does neither—light simply passes through it. And we ignore, for the time being, materials that transmit light, like water and glass, and to some degree, materials like skin.

Because we assume that light only interacts with the surfaces of materials, we represent objects by their surfaces, which are in turn generally represented by polyhedra with triangular faces. Because the edges between faces have no surface area, we ignore them and treat all light–object interactions as happening at the interior of triangles. Each triangular facet T of a polyhedron lies in some plane, and there's a unit vector $\mathbf{n}$ perpendicular to this plane that points away from the object and into the air (or empty space); we call this the **normal vector** to the triangle T. If the polygonal object approximates the original surface well, then this normal vector approximates (and is often treated as) the **surface normal** to the original surface, or a vector perpendicular to the surface at some particular point (see Figure 1.17).

For a perfectly reflective surface like a mirror (a **specular** surface), light that arrives[10] in a direction ℓ and hits the triangle T is reflected in the $\ell\mathbf{n}$-plane, with

10. There are two possible choices for describing the light arriving at a surface: Either record the direction of travel of the photons (the transport-centered view), or record the direction from the surface point to the light (the reflection-centered view). For now, we'll use ℓ for the former; many papers use L for the latter.

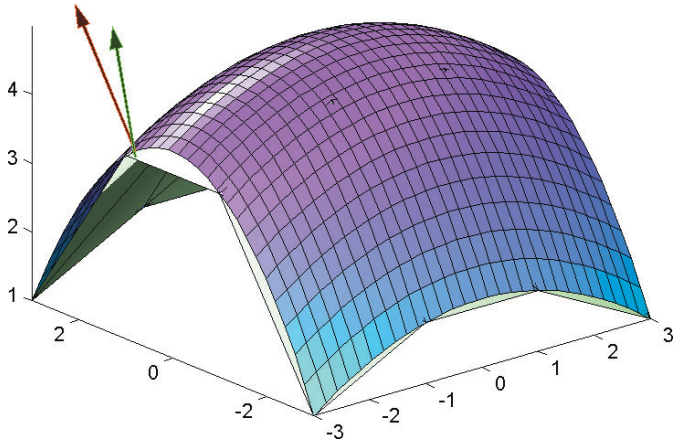


Figure 1.17: A semitransparent smooth surface and a normal vector to it (red), with a polygonal approximation (white) within, and a normal vector to the polygonal mesh at a corresponding point (green).

the angle between the outgoing vector and $\mathbf{n}$ being the same as the angle between $\boldsymbol{\ell}$ and $\mathbf{n}$.

For other surfaces, incoming light scatters in many directions.

For completely diffuse surfaces, light scatters in every direction (Figure 1.18) but the brightness of the reflected light varies in proportion to the absolute value of the dot product[11] $|\boldsymbol{\ell} \cdot \mathbf{n}|$, that is, the cosine of the angle between the surface normal and the incoming light direction.[12] So a surface that faces the light appears bright from wherever one sees it, while a surface that's tilted a bit away from the light appears dimmer. This kind of scattering was described by Lambert long before the development of computer graphics. As a precondition for scattering, the surface must be facing the light, that is, $\boldsymbol{\ell} \cdot \mathbf{n} < 0$. (This **Lambertian reflectance** model is discussed further in Chapters 6 and 27.)

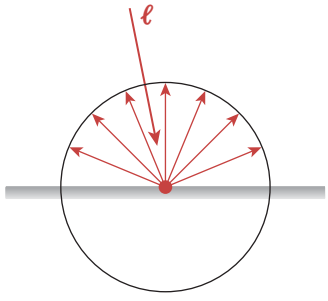


Figure 1.18: Light arrives traveling in direction $\boldsymbol{\ell}$; it's reflected in all directions. For a source of fixed brightness, the intensity of the reflected light is greatest when $\boldsymbol{\ell}$ is perpendicular to the surface.

For somewhat shiny surfaces, the appearance of the surface depends on your viewpoint; if you look at a surface in a well-lit room and move your head back and forth, you may see highlights move on the surface. This can be modeled, with an empirically decent fit, by saying that the reflected light is in proportion to $(\mathbf{n} \cdot \mathbf{h})^k$ for some exponent k, where $\mathbf{h}$ is computed from the average of the vector $-\boldsymbol{\ell}$ from the surface to the light and the vector $\mathbf{e}$ from the surface to the eye, by adjusting that vector to have unit length, that is:

$$\mathbf{h} = \frac{\mathbf{e} - \boldsymbol{\ell}}{\|\mathbf{e} - \boldsymbol{\ell}\|}. \tag{1.4}$$

This model of scattering is due to Phong [Pho75] and Blinn [Bli77], and has been widely used in graphics.

For surfaces in general, the reflected light is a combination of the diffuse, somewhat shiny, and specular cases.

11. The dot product is reviewed in Section 7.6.4.
12. This description is so vague that it's almost meaningless; to make it sensible, we need to discuss how we measure light brightness (a term we're using informally here), which is quite subtle. For now, you should just imagine that brightness is measured from zero to one in some unstated units.

1.13.3 Light Capture

The sensor in a camera and the human eye both respond to light in a similar way: They accumulate light energy for some period of time and then report that accumulated energy. In the case of the sensor, the time period is determined by the shutter opening; in the case of the cell in the eye, the cell sends a signal when the accumulated light reaches some level, so the frequency of signals is proportional to the arriving light intensity. Simulating such a sensor (or cell) therefore involves computing an integral of the incoming light over the area of the sensor. Writing down an analytic solution to this integral for any but the simplest scenes is impractical. For more interesting scenes, we have to perform numerical integration. That necessarily introduces some error, but it also opens up a vast array of computational options for trading quality against time and space. We must approximate the integral by **numerical integration,** which involves evaluating the integrand at several places (this is called **sampling)** and then combining these samples to estimate the overall value. The simplest possible version of this approach is to evaluate the incoming light at the sensor center *only,* and multiply this sample by the area of the sensor to estimate the overall incoming light integral. If the incoming light intensity changes slowly as a function of position, this works quite well; if it changes rapidly, the single-sample approximation introduces many kinds of errors.

1.13.4 Image Display

Modern displays typically are divided into small squares called **pixels;** each small square[13] is individually addressable and can be told to send out a mix of red, green, and blue (RGB) light by specifying a triple of numbers (r, g, b), each between 0 and 255. The amount of light emitted from the square is *not* directly proportional to the numbers; instead, it follows a relation so that equal differences in numbers correspond approximately to equal differences in perceived brightness. You've probably encountered the use of RGB triples in some photo-editing program; typically the RGB values each occupy a single byte, and hence are represented by numbers between 0 and 255, and sometimes are written as two-digit hexadecimals. Thus, a color expressed as 0xFF00CC can be read as "Red is FF, which is 255, there's no green at all, and there's CC worth of blue, which is 204 decimal; that means it's a somewhat reddish purple." It isn't obvious what any given color triple or set of hexadecimal values will yield as a hue; color specification is discussed in Chapter 28.

1.13.5 The Human Visual System

Our eyes respond to light that arrives at the lens, passes through the pupil, and reaches the cornea. While direct sunlight is almost 10^{10} times as bright as the faint light in a dark bedroom, our eyes can detect and process both, but not at the same time. In fact, our eyes adapt to the general illumination around us, and *once adapted* we can distinguish light intensities that range over a factor of about 1000: The faintest thing we can distinguish from black is presenting light to our eyes that's about $1/1000$ the intensity of the thing we perceive as being "as bright

13. The term "pixel" is also used to denote one of the values stored in an image, or a small physical portion of a sensor. There are subtle differences in these denotations, and you should not say "a pixel is a little square" [Smi95].

as possible." Our perception of brightness is not linear, however: If you print thin black stripes on a piece of white paper so that only 20% of the white paper remains visible, it will reflect only 20% of the light that falls on it. But if you place that piece of paper next to a blank piece of the same type of white paper and view both from a distance great enough that the stripes are not visible, the printed paper will appear about half as bright as the unprinted paper. Roughly speaking, for an eye adapted to a given level of light, reducing the incoming light intensity by 80% will make the light seem half as bright.

Our visual system organizes the patterns of light and darkness that arrive at the eye and attempts to make sense of them. The visual system is extremely well adapted to bad input: We can take a black-and-white photograph and add noise (grayscale variation) to it and still be able to recognize the objects in it. We can recognize our homes even in a rainstorm. We can recognize a friend in bright sunlight or in a dark room. In fact, our visual systems are so well tuned to seeing shapes that even when we are watching the static patterns on an old analog television, we occasionally believe we see recognizable patterns. One consequence of this adaptation to bad input is that any stimulus that triggers approximately the right responses leads to recognition: A photograph of a pair of dice, a pencil drawing of them, and a bad computer graphics rendering of them *all* generate in our brains the perception that we are seeing a pair of dice. This has proved to be a blessing and a curse for the field of computer graphics; it means that even very bad approximations of reality make images that we recognize, so it's easy to get started in graphics. On the other hand, it's also easy to believe that the bad approximations are correct because they "look good," and this can impede progress in the field. The adaptability of the visual system has two effects. First, hacking away at graphics can be very satisfying, because even initial results look good enough, on account of adaptability, to make you believe you're getting somewhere. And second, results that appear visually very close to perfect may in fact be generated by programs that are not at all correct, because your visual system is hiding errors from you. Hacking away is really fun (and we encourage you to do it at every opportunity), but it may lead you away from your real goal. We have therefore structured this book so that you get the satisfaction of making fairly good pictures right away, but you also learn, as you do so, the limitations of the techniques that you're learning so that you're better prepared for the more advanced techniques you'll encounter later. If you find yourself asking, "But won't that look wrong in such-and-such a case?" the answer is almost surely "Yes!," and the later chapters will help you understand how to address these limitations.

To return to the importance of perception, in resource-critical applications an understanding of the perception process lets us make informed decisions about what kinds of approximations we can make while still retaining visual fidelity.

1.13.6 Mathematics

Rather than trying to briefly introduce all the mathematics involved in computer graphics, we'll introduce the ideas as they arise; most of them are not directly relevant to a basic understanding of graphics, but rather to the efficient representation or approximation of things we use in graphics. However, after giving you a first taste of 2D and 3D graphics in Chapters 2 and 6, we will review in Chapter 7 some of the mathematics that we assume is familiar to our readers, in part to establish the notational conventions that we'll follow throughout the book. You *can* write graphics programs with only a knowledge of arithmetic and algebra, but

to really work with things in a reasonable way, you'll want to be familiar with the following:

- Trigonometry
- Operations on small vectors and matrices (which we already discussed in this chapter)
- Integrals and derivatives
- And some geometric and topological notions, like continuity, the geometry of surfaces in three dimensions, and curvature

All of this is made easier by a working knowledge of basic linear algebra, which we assume throughout the book.

1.13.7 Integration and Sampling

The most fully developed area of computer graphics is **photorealistic rendering**—producing an image from some model of a scene and the lights in it. Each pixel of a rendered image can be thought of as representing a *measurement* of the light passing along certain rays in the scene, just as each pixel of a digital photograph is a measurement of all the light that hit one small region of the photo sensor in the camera. This can be seen as an integral of the incoming light energy over that region. Since it's impractical to evaluate most such integrals exactly, we end up using approximations (e.g., the rule we mentioned earlier that states that the "integral is approximately the value at the center of the region, multiplied by the area of the region"). In doing this, we've replaced the desired value by a value computed from a single sample; we could have used more samples, but in practice, we'll always be using a finite number of samples, and using these to estimate some integral. Thus, the process of sampling, and of approximating integrals through samples, is central to rendering.


Every measurement in science is an act of statistics: Our measuring device may function differently from day to day; the thing we measure may be just one of many possible, nearly equivalent, measurements (think of measuring the temperature in a beaker of water; you only really measure it in one part of the beaker). In the case of rendering, the statistic is some integral; the random variable is the set of samples that we use to evaluate it, and the result is that a given rendering of a scene usually depends on some random number generator: Multiple renderings of the same scene with the same software will produce different values for any particular pixel. This distribution of values will typically cluster around some mean value, which one hopes is correct, and will have some variance. If the variance of adjacent pixels is uncorrelated, it may appear in the output as speckle, or visual noise. If it's correlated, it may appear as **jaggies**—a staircaselike representation of what should be a smooth diagonal line. This means that assessing the quality of an algorithm also entails statistical measurements.

1.14 Learning Computer Graphics

The subject matter of computer graphics is no longer linearizable in any reasonable way. Each topic ends up so intertwined with all others that there's no way to decide which one to discuss first, and any presentation ends up with successive disclosures: a first description, a later correction, a further improvement, etc. Readers, naturally, like books to be *organized;* when you want to review

something about polygon meshes, you hope there will be a chapter that has all the polygon-mesh information in it, for instance. Then again, a book that treated each subject in its entirety before moving on to any other would have you make your first pictures at the end of an entire semester of study, at the earliest!

We have compromised: In this introductory chapter, we've given you some very informal information about light, perception, the representation of shapes, and the interaction of light with shapes so that you can understand how to make some pictures right away. When you *do* make pictures with these sloppy models of things, the pictures won't be very good. Sure, your picture of a boxy robot will be recognizable as a boxy robot, but in looking at it critically, you'll soon realize that there's no way you could make a *real* robot shape (from cardboard, tape, and paint, say) and photograph it with a *real* camera and end up with anything like the picture you've made. It's not photorealistic. But making those first pictures will give you experience with creating models, with certain applications of linear algebra, with polygonal meshes, and with some key ideas for rendering, all of which will make you better able to understand and experiment with richer or more accurate models of light, reflection, objects, etc., as you encounter them.

The next few chapters introduce Microsoft's Windows Presentation Foundation (WPF), a framework for writing graphics programs, some basic ideas from rendering, an introduction to visual perception, and quite a lot of mathematics that's useful throughout graphics.

Chapter 2 introduces the 2D aspects of WPF to get you familiar with drawing simple 2D shapes. WPF uses a declarative specification of graphics—in contrast to more traditional APIs—which is valuable both because it provides a higher level of abstraction and because its interpretive nature makes it very useful for rapid prototyping. Modeling in WPF is based on a hierarchical representation of shapes that's widely used in almost all graphics APIs.

Chapter 3 describes a program for making very simple pictures of very simple 3D shapes so that you can understand from the start how simple graphics can be. Chapter 4 describes two WPF programs that you'll use when conducting experiments in graphics throughout much of the remainder of the book.

Chapter 5, which covers perception, describes some of the most pertinent aspects of the human visual system.

In Chapter 6 we present an introduction to the 3D aspects of WPF, which also informally introduces the geometric tools used for shape modeling, and the application of the simple models of how light and objects interact that we have described in this chapter. It also continues the description of hierarchical models of compound shapes introduced in Chapter 2.

With the experience of using both the 2D and 3D versions of WPF, you will be prepared for Chapter 7's review of mathematical essentials for graphics. Chapters 8 through 13 introduce the linear algebra that lies at the core of a great deal of computer graphics, together with certain data structures that represent the topology and geometry from which we make images.

Following this, we again discuss (in Chapter 14) the conventional approximations to reality—the models—that are used in many basic graphics systems. We describe models of light, of shape, of material, and of how light is transported in a scene, in each case with more detail than in this chapter. This rather long chapter not only prepares you for the later material on rendering, shape representation, and material representation, but also introduces many topics that are essential for understanding legacy programs.

With an understanding of basic models of light and reflection, we can make preliminary versions of two renderers: a ray tracer and a rasterizer (Chapter 15). Doing so introduces the key ideas and challenges of each kind of rendering; because our preliminary versions are so basic, Chapter 15 also introduces some of the problems of each, and of the conventional approximations to reality.

In Chapter 16 we discuss various graphics systems, comparing and contrasting them with WPF; by the end of that chapter, you will have encountered much of what was traditionally taught in computer graphics.

The remaining chapters in the book discuss images and signal processing, light, color, materials, texturing, and rendering; cover some interaction techniques, geometric algorithms and data structures that support rendering as well as many interaction methods, and various approaches to modeling shapes; and introduce animation and graphics hardware. These chapters are less sequential and more interdependent than the chapters preceding them. You can skip forward and read about splines and subdivision surfaces if you want to learn how to create interesting shapes, but you'll find references to the ideas of convolution and filtering that were introduced in Chapters 17 through 19. You can read about some of the best available rendering algorithms in Chapter 32, but you'll find that the discussion relies heavily on the discussion of rendering theory in Chapter 31. This should not prevent you from taking this approach; for many students, it's the desire to make the practical algorithms work that motivates them to learn more about the theory, and if you read Chapter 31 with particular questions in mind, you may find the material easier to absorb.

This page intentionally left blank

Chapter 2

Introduction to 2D Graphics Using WPF

2.1 Introduction

Having presented a broad overview of computer graphics, we now introduce a more immediately practical topic: application programming using a commercial graphics platform. After an overview of the history of 2D platforms, we examine a specific one, Microsoft Windows Presentation Foundation (WPF).

We chose WPF because it is one of the few modern graphics platforms that support both 2D and 3D applications, providing user-interface as well as rendering functionality using a consistent programmer's model. In addition, it is an excellent rapid-prototyping platform for experimenting with the principles of 2D and 3D graphics. Its Extensible Application Markup Language (XAML) is a declarative language (in the style of HTML) that provides a concise way to construct scenes, and XAML interpreters provide virtually instantaneous testing/debugging cycles. This allows us to introduce you rapidly to a number of fundamental concepts in 2D and 3D graphics and to let you experiment almost immediately without a time-consuming learning curve.

Of course, declarative languages have their limitations, particularly in support for conditionality and flow of control, so WPF developers can extend XAML with procedural code written in an imperative programming language such as C#. This hybrid strategy is simplified by WPF's cross-language consistency; for example, each XAML element type corresponds to a WPF class, and the element's properties correspond to data members of that WPF class.

Our dedication of a chapter to 2D may surprise you. First, we feel that many 3D concepts—such as specification and transformation of geometric shapes, hierarchical modeling, and animation—are easier to understand when initially presented in a 2D context, free of complex 3D-related requirements such as simulating the interaction between lights and materials. Second, we note the dominance of 2D graphics in applications across all platforms from smartphone to

tablet to desktop, and the common integration of 3D renderings together with 2D user interfaces and visualizations such as maps, schematics, data grids/charts, etc.

This chapter and its 3D continuation (Chapter 6) form a sequence, so a good understanding of this material and comfort with XAML are a prerequisite to Chapter 6. Thus, we strongly suggest that you perform the associated exercises using the accompanying lab software, which presents small XAML programs inside an integrated editor/interpreter providing instant feedback, thus reducing the learning curve and making experimentation easy and stimulating.

2.2 Overview of the 2D Graphics Pipeline

In Chapter 1, we saw that a graphics platform is an intermediary between the application and the display hardware, providing functionality related to both output (instructing the GPU to display information) and input (invoking callback functions in the application to respond to user interaction). To prepare for a discussion of the various types of graphics platforms, let's take a high-level view of a 2D graphics application, shown in Figure 2.1.

It is rare that an application's purpose is only to paint pixels. Usually some data—which we call the **application model (AM)**—is being represented by the rendered image and manipulated via user interaction with the application. In a typical desktop/laptop environment, the application is running in conjunction with a window manager, which determines the area of the screen allocated to each application and takes care of the display of and interaction with the **window chrome** (i.e., the title bar, resize handles, close/minimize buttons, etc., shown in pale green in Figure 2.1). The application's focus is on drawing inside the

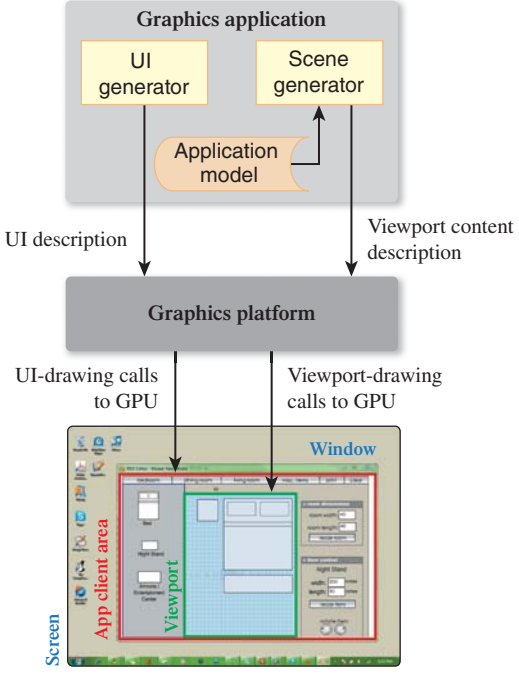


Figure 2.1: Graphics platform as an intermediary between a 2D application and display-device screen-space resources allocated by the window manager.

client area (red in Figure 2.1) that is the interior of the window, by making calls to the graphics platform API. The platform responds to those calls by driving the GPU to produce the desired rendering.

Typically the application uses the client area for two purposes: Some portion of the area is devoted to the application's user-interface (UI) controls, and the remaining area contains the **viewport** that is used to display the rendering of the **scene,** which is extracted or derived from the AM by the application's **scene generator** module. As you can see in the diagram, the **UI generator** module that generates the user interface is distinct from, and operates very differently from, the scene generator, even though they both use the underlying 2D platform to drive the display.

Our use of the terms "scene" and "viewport" for 2D may surprise those with experience in 3D graphics, in which those terms have 3D-centric usages. In the 2D domain, we use the term "scene" analogously to mean the collection of 2D shapes that will be rendered to create a particular view of the AM. Note that the 2D scene generator corresponds directly to the scene generator for 3D applications that feeds a 3D platform to produce a rendering. Similarly, our 2D use of the term "viewport"—to mean an area in which the scene's rendering will appear—is consistent with 3D usage.

Consider an interior-design application that displays and enables editing of a furniture layout. The application model records all data associated with a given furniture layout, including nongraphical data such as manufacturer, model number, pricing, weight, and other physical characteristics. Some of this information is needed to produce a graphical view of the model, and some is used only for nongraphical functionality (e.g., purchasing). It is the task of the application's scene generator to traverse the application model, extract or compute the geometric information relevant to the desired scene, and invoke the graphics platform API to specify the scene for rendering.

The scene may contain a visualization of all the geometry described in the application model or it may represent a subset (e.g., showing only one room of the house being designed). Moreover, analogous to multiple views of databases, the application may be able to provide multiple views using different presentation styles for the same geometric information (e.g., showing furniture either as outlines or as shapes filled with textures simulating fabric or wood).

In the above example, the application model is inherently geometric. However, in other applications the AM may contain no geometric data at all, as is typical in **information visualization** applications. For example, consider a database storing population and GDP statistics for a set of countries. In this case, the scene will often be a chart or graph, derived from the AM by the scene generator and designed to present these statistics in an intuitive visualization. Other examples of data visualization applications include organizational charts, weather data, and voting patterns superimposed on a map background.

2.3 The Evolution of 2D Graphics Platforms

Graphics platforms have experienced the same low- to high-level evolution (depicted in Figure 2.2) that has taken place in programming languages and software development platforms. Each new generation of raster graphics platform has offered an increasingly higher level of abstraction, absorbing common tasks that previously were the responsibility of the application.

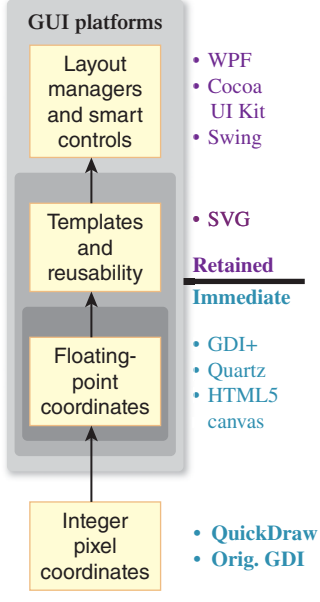


Figure 2.2: Evolution in level of abstraction in commercial 2D graphics platforms—from immediate mode to retained mode.

2.3.1 From Integer to Floating-Point Coordinates

We'll start with the state of the art of raster graphics in the 1980s and early 1990s. The typical popular 2D raster graphics platform (e.g., Apple's original QuickDraw and Microsoft's original GDI) provided the ability to paint pixels on a rectangular canvas, using an integer coordinate system. Instead of painting individual pixels, the application painted a scene by calling procedures that drew **primitives** that were either geometric shapes (such as polygons and ellipses) or preloaded rectangular images (often called bitmaps or pixmaps, used to display photos, icons, static backgrounds, text characters extracted from font glyph sets, etc.). Additionally, the appearance of each geometric primitive was controlled via specification of attributes; in Microsoft APIs, the **brush** attribute specified how the interior of a primitive should appear, and the **pen** attribute controlled how the primitive's outline should appear.

For example, the simple clock image shown in Figure 2.3 is composed of four primitives: an ellipse filled using a solid-gray brush, two polygons filled using a solid-navy brush for the clock's hour and minute hands, and a red-pen line segment for the second hand.

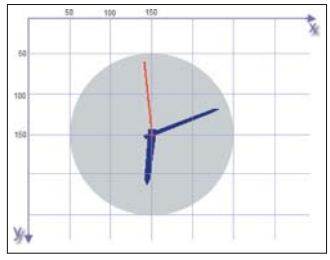


Figure 2.3: Clock scene with GDI coordinate-system overlay.

In the original GDI's simplest-usage scenario, the application uses integer coordinates that map directly (one-to-one) to screen pixels, with the origin (0,0) located in the upper-left corner of the canvas, and with x values increasing toward the right and y values increasing toward the bottom.

The application specifies each primitive via a function (e.g., `FillEllipse`) that receives the integer geometry specifications along with appearance attributes. (The GDI source code for this example application is available as part of the online material for this chapter.) The specification is reminiscent of plotting on graph paper; for example, the geometry of the gray circular clock face is passed to the `FillEllipse` function via this data pair:

> Center point: (150,150)
>
> Bounding box (smallest axis-aligned enclosing rectangle): upper left at (50,50), dimensions of 200 × 200

How large will this clock face appear when rendered onto the output device? There's no definitive answer to that question. The displayed size depends on the resolution[1] (e.g., dots per inch, or dpi) of the output device. Suppose our clock application was originally designed for a 72dpi display screen. If the application were tested on a higher-resolution device (e.g., a 300dpi printer or screen), the clock's image would be smaller and possibly illegible. Conversely, if the target display were changed to the small, low-resolution screen of an early-generation smartphone, the image might become too big, with only a small portion of it visible.

The raster graphics community solved this problem of **resolution dependence** by borrowing ideas long present in vector graphics, using floating point to support alternative coordinate systems that insulate geometry specification from device-specific characteristics. In Section 2.4, we'll introduce and compare two such coordinate systems: **physical** (based on actual units of measurement like

1. This use of the term "resolution" contrasts with another common usage, the total number of display pixels (e.g., "LCD monitor with 2560 × 1440 resolution").

millimeters and typographic points) and **abstract** (with application-determined semantics).

2.3.2 Immediate-Mode versus Retained-Mode Platforms

The evolution from integer-based specification to floating point was shared by all major 2D graphics platforms, but eventually a "split" occurred, creating two architectures with different goals and functionality: **immediate mode (IM)** and **retained mode (RM).**

The former category includes platforms (like Java's `awt.Graphics2D`, Apple's Quartz, and the second-generation GDI+) that are thin layers providing efficient access to graphics output devices. These lean platforms do not retain any record of the primitives specified by the application. For example, when the `FillEllipse` function of GDI+ is invoked, it immediately (thus the term "immediate mode") performs its task—mapping the ellipse's coordinates into **device coordinates** and painting the appropriate pixels in the display buffer—and then returns control to the application. At its most basic, the programmer's model for working in IM is straightforward: To effect any change in the rendered image, the scene generator traverses the application model to regenerate the set of primitives representing the scene.

The lean nature of IM platforms makes them attractive to application developers who want to program as close to the graphics hardware as possible for maximum performance, or whose products must keep as small a resource footprint as possible.

But other application developers look for platforms that offload as many development tasks as possible. To satisfy these developers, RM platforms retain a representation of the scene to be viewed/rendered in a special-purpose database that we call a **scene graph** (discussed further in Chapters 6 and 16). As shown in Figure 2.4, the application's UI and scene generators use the RM platform's API to create the scene graph, and can specify changes incrementally by simply editing the scene graph. Any incremental change causes the RM platform's display synchronizer to automatically update the rendering in the client area. Because it retains the entire scene, the RM platform can take on many common tasks concerning not only the display, but also user interaction (e.g., **pick correlation,** the determination of which object is the target of a user click/tap, as described in Section 16.2.10).

All RM packages can be traced back to Sketchpad [Sut63], Ivan Sutherland's pioneering project from the early 1960s, which launched the field of interactive graphics. Sketchpad supported the creation of **master** templates, which could be **instantiated** one or more times onto the canvas to construct a scene. Each template was a group of primitives and possibly instances of subordinate templates, bundled to compose a single unified graphics object. Each instance could be geometrically transformed—that is, positioned, oriented, and scaled—but in all other respects, the instance retained the appearance of its master, and changes to the master would immediately be reflected in all instances.

These key ideas from Sketchpad survive in all modern RM packages, making these platforms excellent foundations for creating user interfaces. **UI controls** (also known as widgets) are templated objects that, as an integrated collection, have an inherent, consistent **look and feel.** In this commonly used phrase,

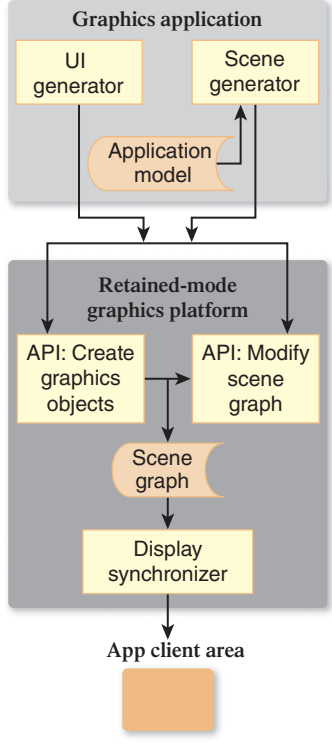


Figure 2.4: Schematic of a graphics application atop a retained-mode platform storing a scene graph.

the word "look" refers to the graphical design/appearance (size, shape, font, coloring, drop shadow, etc.). The word "feel" pertains to the controls' dynamic behaviors, typically in response to user interaction, which can be subdivided into built-in automated feedback behaviors and semantic/application behaviors. Examples of built-in feedback include the graying-out of a control that is currently disabled, the glow highlighting of a button when the pointer enters its region, and the display of a blinking cursor when the user is typing into a control's text box. These feedback behaviors often include nice-looking animations, performed by the platform with no application involvement. Of course, the application must get involved when the user initiates an application action (e.g., clicks a button to submit a form for processing). To spark such activity, the RM simply invokes the application callback function attached to the manipulated control.

Most RM UI platforms also include layout managers that spatially arrange controls in a pleasing and organized way, with consistent dimensions and spacing, and that provide for automatic revision of the layout in reaction to programmatic or user-initiated changes in the size or shape of the UI region.

A well-designed set of UI controls requires significant work by a team with expertise in graphic and UI design; it is no small feat to construct a pleasing and intuitive UI framework. Rendering and laying out the UI, and handling user interaction, make up a large portion of the work involved in building an interactive application, so it should be no surprise that the use of RM UI platforms, which offload many tasks as described above, is pervasive. Indeed, it would be hard to find a modern 2D application that does not use an RM UI platform to handle virtually all of its needs for interaction through components such as menus, buttons, scroll bars, status bars, dialog boxes, and gauges/dials.

In contrast with retained mode's high popularity in the 2D domain, its use in 3D is less pervasive. Even though 3D RM platforms offer powerful features—such as simplifying hierarchical modeling and rigid-body animation—these features carry a high resource cost. We address this topic in greater detail in Chapter 16.

2.3.3 Procedural versus Declarative Specification

Traditionally, each graphics platform has provided one of the following techniques to developers for the purpose of specification of user interfaces and/or scenes:

- **Procedural code** written in an imperative programming language (typically, but not necessarily, object-oriented), interfacing with the display devices via any of dozens of graphics APIs, such as Java Swing, Mac OS X Cocoa, Microsoft WPF or DirectX, Linux Qt or GTK, etc.
- **Declarative specification** expressed in a markup language, such as SVG or XAML

One of WPF's distinguishing characteristics is that it offers developers a choice of specification techniques, as shown in Figure 2.5 and described below.

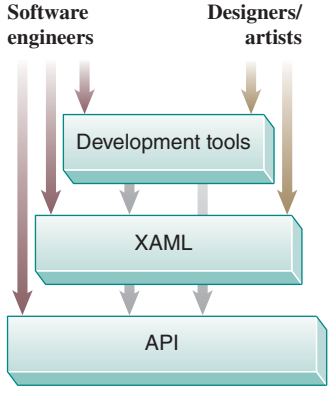


Figure 2.5: WPF application/ developer interface layers.

2.3.3.1 Lowest Layer: Object-Oriented API

The core layer is a set of classes providing all WPF functionality. Programmers can use any of the Microsoft .NET languages (e.g., C# or Visual Basic) or Dynamic Language Runtime languages (such as IronRuby) to specify application appearance and behavior at this level. A WPF application can be created via this layer alone, but the other two layers provide improvements in developer efficiency and convenience, and the ability to include designers and implementers in less technical roles.

2.3.3.2 Middle Layer: XAML

The middle layer provides an alternative way to specify a large subset of the functionality of the API, via the declarative language XAML, whose syntax is readily understandable by anyone familiar with HTML or XML. Its declarative nature facilitates support for rapid prototyping via interpreted execution, and it is more conducive to use by nonprogrammers (in the same way that HTML is more approachable than PostScript).

2.3.3.3 Highest Layer: Tools

As with any language, there is a learning curve associated with adopting XAML. The highest layer of the WPF application/developer interface comprises the utilities that designers and engineers can use to generate XAML, including tools for drawing graphics (e.g., Microsoft Expression Design or Adobe Illustrator), building 3D geometric models (e.g., ZAM 3D), and creating sophisticated user interfaces (e.g., Microsoft Expression Blend or ComponentArt Data Visualization).

2.4 Specifying a 2D Scene Using WPF

As explained earlier, WPF provides for both the construction of user-interface regions and the specification of what we call "2D scenes." The former is beyond the scope of this textbook, so our focus here is on the specification of 2D scenes.

2.4.1 The Structure of an XAML Application

Throughout Section 2.4, we'll be building a simple XAML application that displays the analog clock shown in Figure 2.6.

If you are familiar with HTML syntax, XAML should be instantly accessible. An HTML file specifies a multimedia web page by creating a hierarchy of **elements**—with the root being `<HTML>`, its children being `<HEAD>` and `<BODY>`, all the way down to paragraph and "text-span" elements for formatting, such as `<B>` for boldface and `<I>` for italics. Other elements provide support for media presentation and script execution.

An XAML program similarly specifies a hierarchy of elements. However, the set of element types is distinct to XAML, and includes layout panels (e.g., a StackPanel for arranging tightly packed controls/menus, and a Grid for creating spreadsheetlike layouts), user-interface controls (e.g., buttons and text-entry boxes), and a rectangular "blank slate" scene-drawing area called the `Canvas`.

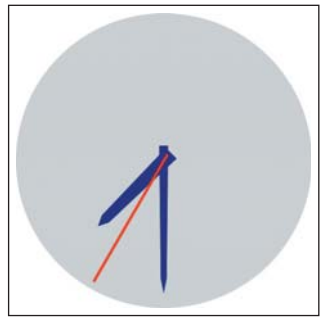


Figure 2.6: WPF-based clock application.

In a fully formed application, like the one shown in Figure 2.1, the application's appearance is specified via a hierarchy of layout panels, UI controls, and a `Canvas` element acting as the viewport displaying the application's scene; however, for our first simple XAML example, let's just create a standalone `Canvas`:

```
<Canvas
   xmlns=
     "http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x=
     "http://schemas.microsoft.com/winfx/2006/xaml"
   ClipToBounds="True"
>
</Canvas>
```

The setting of `ClipToBounds` to `True` is almost always desired; it simply ensures that the canvas is bounded, that is, it does not display any data outside its assigned rectangular area.

We have not specified the size of the canvas, so its size will be controlled by the application in which it appears. For example, the lab software for Sections 2.4 and 2.5 (provided as part of the online materials) includes a "split-screen" layout with the WPF canvas in one pane, vertically stacked on top of a second pane displaying the XAML source code. The lab uses WPF layout managers to allocate space between the two panes, and uses a draggable-separator control to allow the user to exert some control over that allocation.

You will note that XAML has syntactic idiosyncrasies (such as the strange `xmlns` properties in the `Canvas` tag shown above), but they rarely obscure the semantics of the tags and properties, which are well named for the most part. If you choose to investigate the more cryptic parts of the syntax, just use the lab software: Click on any maroon-highlighted XAML code to request a brief explanation.

2.4.2 Specifying the Scene via an Abstract Coordinate System

Our sample application's scene—the simple clock—is a composite of several objects: the face and three individual hands. The face object is a single ellipse primitive filled with a solid-gray color. Two of the clock hands (minute and hour) are navy-filled polygons, similar in shape but differing in size. Finally, there is the red line forming the second hand.

Note that thus far in our simple scene graph, all the components are primitives, but in a more complex scenario (introduced in Section 2.4.6), there may be a hierarchy in which components may be composed of lower-level subcomponents.

With our list of components in hand, we now refine our specification by detailing the precise geometry of each primitive.

Take a blank sheet of graph paper, choose and mark the (0,0) origin, and draw the x-axis and y-axis—the result is the 2D Cartesian coordinate system. One of its characteristics is that any two real numbers form an (x, y) coordinate pair that uniquely identifies exactly one point on the plane.

But there's a limit to the lack of ambiguity of a graph-paper coordinate system. People asked to draw a 4×4 square on graph paper will all produce a square shape encompassing 16 grid squares, but these shapes will not have an identical areas in terms of physical units (e.g., cm^2), because there is no single standard grid/ruling size for graph paper.

Indeed, a sheet of graph paper is, by itself, an **abstract coordinate system** in that it does not describe positions or sizes in the physical world. Using an abstract system for geometry specification is perfectly fine—in fact, we are about

to construct our clock using one. But the "real world" must be reckoned with when it's time to display such a scene, and at that point the abstract system must be mapped to the display's physical coordinate system. We'll describe that mapping shortly, but first let's start the process of geometric specification. We will use the abstract coordinate system shown in Figure 2.7.

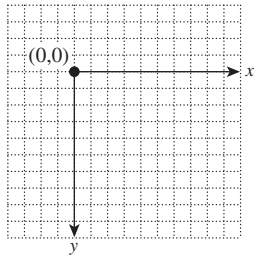


Figure 2.7: Abstract coordinate system.

Which primitive should we draw first? By default, the order of specification does matter, so an element E, constructed after element D, will (partially) occlude D if they overlap.[2] The term **"two-and-a-half dimensional"** is sometimes used to describe this stacking effect.

Thus, we should work from back (farthest from the viewer) to front (closest to the viewer), so let's start with the circular clock face.

Figure 2.8 shows a simple single-circle design for the face. We've arbitrarily chosen a radius of ten graph-paper units, because that size is convenient on this particular style of graph paper. This decision is truly arbitrary; there is no one correct diameter for this clock, since the coordinate system is abstract.

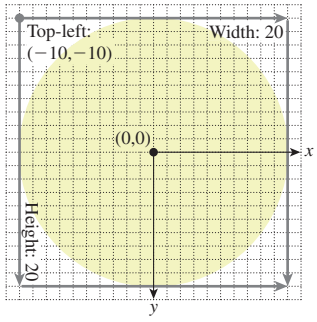


Figure 2.8: Defining the clock face's geometry using our abstract coordinate system.

The syntax for specifying a solid-color-filled circular ellipse is:

```
<Ellipse
    Canvas.Left=...  Canvas.Top=...
    Width=...        Height=...
    Fill=...
/>
```

where `Canvas.Left` and `Canvas.Top` specify the *x*- and *y*-coordinates for the upper-left corner of the primitive's bounding box, and `Fill` is either a standard HTML/CSS color name or an RGB value in hexadecimal notation (#RRGGBB—e.g., #00FF00 being full-intensity green).

We now are ready to construct a WPF application that places this primitive on a canvas:

```
<Canvas ... >
  <Ellipse
    Canvas.Left="-10.0" Canvas.Top="-10.0"
    Width="20.0" Height="20.0"
    Fill="lightgray" />
</Canvas>
```

(Note: In this and the remaining XAML code displays in this chapter, we highlight the new or modified portion for your convenience.)

Although this specification is unambiguous, it's not obvious what this ellipse will look like when displayed. What is the on-screen size of a circle of diameter 20 units, where our unit of measurement was determined by an arbitrary piece of graph paper?

We suggest you run the lab software (available in the online resources), and select V.01 to see the result of the execution of the above XAML. A screenshot of the rendered result (shown in Figure 2.9, along with a mouse cursor for scale) shows that the result is not acceptable for two reasons: The gray circle is too small to act as a usable clock face, and we are only seeing one quadrant.

Figure 2.9: Rendered result of revision V.01 of our XAML clock application, exhibiting problems with both image size and positioning.

The schematic view shown in Figure 2.10 depicts this ellipse specification, with the left side (light-pink box) representing the abstract geometric data that

2. This default order-dependent stacking order can be overridden by the optional attribute `Canvas.ZIndex`.

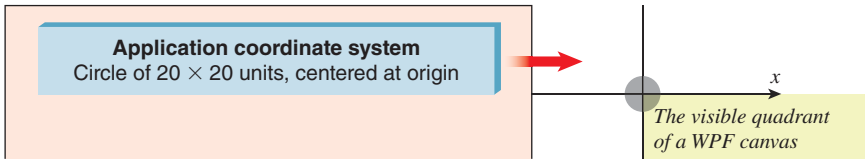


Figure 2.10: Schematic view of our application's initial specification of the clock-face ellipse.

lives in the application with no physical representation, and the red arrow representing the rendering process that produces the displayed image.

Here, we are seeing the effect of delivering abstract coordinates directly to the graphics platform. It is OK to use abstract coordinates to design your scene, but when it's time to worry about how it will appear on the display, we must consider (1) characteristics of the display device, such as size, resolution, and aspect ratio; (2) how we want the rendered image to be sized and positioned in consideration of the form factor's constraints; and (3) how to specify the geometry to the graphics platform in order to achieve the desired result.

In Section 2.6, we will discuss these considerations in the scope of full-featured applications. Here, for our simple clock application, let's assume that the canvas will be displayed on a laptop screen, that the clock should be an "icon-sized" 1 inch in diameter, and that the clock should appear in the upper-left corner of the canvas. How can we revise our application to achieve this desired appearance?

2.4.3 The Spectrum of Coordinate-System Choices

We now have a specific physical size (one inch in diameter) in mind for our clock scene. So, should we reconsider our decision to design the geometry using an abstract coordinate system? To answer this question, let's consider two alternative coordinate systems we might choose for scene description.

We could consider using an integer-pixel-based coordinate system like that described in Section 2.3.1, but that is not appropriate given our need to control the displayed size of our scene, independent of screen resolution.

Alternatively, we can consider designing our scene using the WPF canvas coordinate system, which is "physical"—the unit of measurement is $1/96$ of an inch—and *not* resolution dependent. For example, an application can draw a rectangle of size $1/8 \times 1/4$ inches by specifying a width of 12 units and a height of 24 units.[3] Thus, we can create a circle that is 1 inch in diameter by specifying the diameter as 96 units.

Although direct use of the WPF coordinate system does provide resolution independence, we do not recommend that strategy, for there are two other kinds of independence worthy of pursuit.

- **Software-platform independence:** By using the coordinate system of a specific graphics platform, we are unnecessarily tying portions of our application code to that platform, potentially increasing the work that would be necessary to later "port" the application to other platforms.

3. There *are* limitations to physical coordinate systems. Perfect accuracy in the sizes of displayed shapes cannot be guaranteed due to dependencies on disparate parts, including the device driver and the screen hardware.

- **Display-form-factor independence:** The display screens on today's devices come in a huge variety of sizes and aspect ratios (also known as form factors). To ensure compatibility with a large variety of form factors, from phone to tablet to desktop, a developer should keep scene geometry as abstract as possible and nail down the geometry at runtime using logic that considers the current situation (form factor, window size, etc.). For example, in deciding on a 1-inch diameter for our clock, we were thinking about icons on a laptop form factor; we might well choose a different optimal size for a smartphone device. An abstract coordinate system allows for runtime decision making on actual physical sizing. For further discussion on this important topic, see Section 2.6.

We now see that the use of an abstract coordinate system is advantageous in several ways, so let's continue with that strategy.

There's a further advantage to the abstract coordinate system: It's often easier to specify a shape using small numbers—for example, to say, "I want a disk that goes from -1 to 1 in x and y, and then I want to move it to be centered at $(37, 12)$," rather than saying, "I want a disk that goes from 36 to 38 in x and from 11 to 13 in y." In the former specification, it's easy to see that the radius of the disk is 1, and that it's a *circular* disk rather than an elliptical one. This idea—that it's easier to work in some coordinate systems than in others—will arise again and again, and we embody it in a principle:

✓ **THE COORDINATE-SYSTEM/BASIS PRINCIPLE:** Always choose a coordinate system or basis in which your work is most convenient, and use transformations to relate different coordinate systems or bases.

2.4.4 The WPF Canvas Coordinate System

At this point, you have been informed of only one characteristic of WPF canvas coordinates. Figure 2.11 demonstrates the other important features: The origin (0,0) lies at the upper-left corner of the canvas, the positive x-axis extends to the right, the positive y-axis extends downward, and the canvas is "bounded" on all four sides (represented by the light-blue rectangle in the figure). That is to say, each WPF canvas has a definitive width and height (usually controlled by layout logic as described previously). In the common case of `ClipToBounds="True"`, these bounds are strictly enforced, so any visual information lying outside the bounds is invisible.[4]

With this information, we can now return to developing our clock application. Let's prepare by reviewing the sequence of "spaces" through which the scene's geometry travels from abstract to physical to device, shown in Figure 2.12. We already discussed the application and WPF canvas coordinate systems, and in Section 2.4.5 we show you how the former system is mapped to the latter. So here we'll briefly address the final transition shown in the sequence, the mapping of the WPF canvas to actual pixels on the display device. This part of the pipeline is not

4. As we implement this application throughout Section 2.4, we'll assume the canvas is large enough to show the entire clock, but Inline Exercise 2.5 will invite you to investigate what happens when it's not.

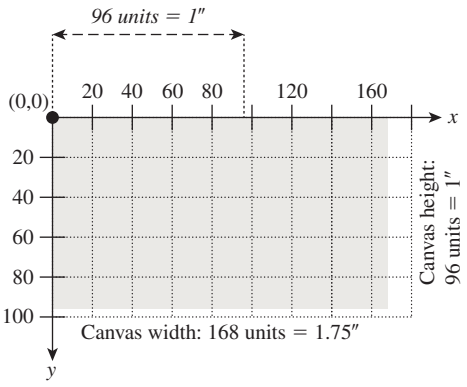


Figure 2.11: A WPF canvas of size 168 × 96 *units, with* `ClipToBounds=True`*. Note the hardwired location of the origin and the hardwired semantics of* 96 *units to the physical inch. On the display device, when rendered by an accurate device driver, this canvas will appear with a size of* 1.75 × 1 *inches. The canvas is bounded on all four sides and displays only the visual information lying within the bounds.*

fully under application control; rather, it is performed by a collaboration between a number of modules: the WPF layout managers (created and configured by the application to control the location and size of all components in the client area, including the canvas), the window manager (controlling the location and size of the application's client area), and the low-level rasterization pipeline (composed of a sequence of modules such as an immediate-mode package like DirectX or OpenGL, a low-level device driver, and the graphics hardware itself).

> In the mid-1980s, the standard device-independent unit (DIU) for both Mac and Windows was 1/72 of an inch, corresponding to the dpi of typical display monitors at the time. But research by Microsoft revealed that the typical computer user sits one-third farther away from a display screen than from a printed page. Thus, to ensure that text rendered at a given point size appears roughly equivalent on screen and on paper, the DIU for GDI was scaled up by 33% to 96dpi.
>
> Keep in mind that graphics software platforms do not have control over the accuracy of display hardware, so the DIU is only an approximation. A line of length 96 units on a WPF canvas will appear to be one inch long on an "ideal device," but not necessarily on an actual display screen.

2.4.5 Using Display Transformations

At last, we now understand why our clock face, as defined in our abstract system, produces the unacceptable result of Figure 2.9.

- The circle has a radius of 20 units. We now know that 20 units on the WPF canvas is less than 1/4 inch, unacceptably small.
- The circle is specified with its center at the origin. We now know that the WPF canvas shows only data in the $(+x, +y)$ quadrant, so most of our circle is hidden.

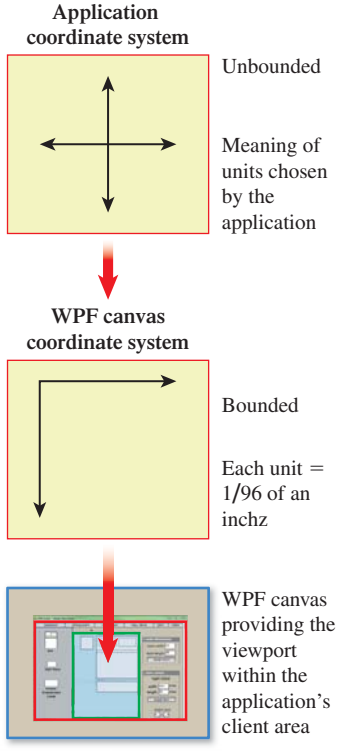


Figure 2.12: Progression from an application's abstract coordinate system, then into the WPF canvas coordinate system, and finally onto the display device as a part of the application window's client area.

To repair our application, we will set up a **display transformation** to mathematically adjust the entire scene's geometry from the abstract application coordinate system to the WPF canvas system in a way that (a) makes the clock completely visible and (b) makes it the right size.

First, consider the need to resize. Our clock's diameter is 20 units in our abstract system. We would like that to map to 1 inch on the WPF canvas; thus, we want it to map to 96 WPF units. Consequently, we want to multiply each graph-paper coordinate by 96/20, or 4.8, on both axes. To request that the canvas perform this scale transformation, we attach a `RenderTransform`[5] to the canvas to specify a geometric operation that we want to be applied to all objects in the scene:

```
<Canvas ... >

  <!- THE SCENE ->
  <Ellipse ... />


  <!- DISPLAY TRANSFORMATION ->
  <Canvas.RenderTransform>
   <!- The content of a RenderTransform is a TransformGroup
       acting as a container for ordered transform elements. ->
    <TransformGroup>
     <!- Use floating-point scale factors:
         1.0 to represent 100%, 0.5 to represent 50%, etc. ->
     <ScaleTransform ScaleX="4.8" ScaleY="4.8"
                     CenterX="0" CenterY="0"/>
    </TransformGroup>
  </Canvas.RenderTransform>

</Canvas>
```

Note that when you specify a 2D scale operation, you must specify the center point, which is the point on the plane that is stationary—all other points move away from (or toward) the center point as a result of the scale. Here, we use the origin (0, 0) as the center point.

The effect of our new revision (V.02 in the laboratory) is depicted in Figure 2.13. Clearly, we have solved the size problem, but still only one quadrant of the circle is present on the visible portion of the canvas.

Thus, we want to add another transform to our canvas, to move our scene to ensure full visibility. We will use a translate transformation:

```
<TranslateTransform X="..." Y="..."/>
```

How many units do we need to translate? Since our scale transform has ensured that our circle has a 1-inch diameter on the WPF canvas, and we're seeing only one-half of the circle on each dimension, we need to move the circle a half-inch down and a half-inch toward the right (i.e., 48 canvas units on each axis) to ensure full visibility.

5. WPF's use of the term "RenderTransform" for a transformation is somewhat misleading since it implies it is used only to control display. A better name would be "GeometricTransform" since this element type performs 2D geometric transformations to achieve a wide variety of purposes, for both modeling and display control, as is demonstrated throughout this chapter.

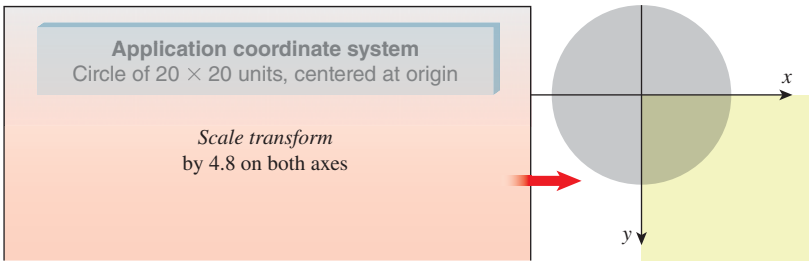


Figure 2.13: Schematic view of our application now enhanced with a scale transform.

Here is the revised XAML (V.03 in the lab); its effect is depicted in Figure 2.14.

```
1  <Canvas ... >
2    <!- THE SCENE ->
3    <Ellipse ... />
4
5    <!- THE DISPLAY TRANSFORM ->
6    <Canvas.RenderTransform>
7      <TransformGroup>
8       <ScaleTransform  ScaleX="4.8"  ScaleY="4.8" ... />
9       <TranslateTransform X="48" Y="48" />
10     </TransformGroup>
11   </Canvas.RenderTransform>
12 </Canvas>
```

NOTE: Animated versions of all of the application schematic views in this chapter are provided as part of the online material.

To review: We have used a sequence of transforms, attached to the canvas, to perform what we call a display transformation to execute the geometric adaptations necessary to make our scene have the desired spatial appearance on the display device. The display transformation maps our application coordinate system to WPF's canvas coordinate system; we indicate this goal state by highlighting the coordinate system's representation with a drop shadow.

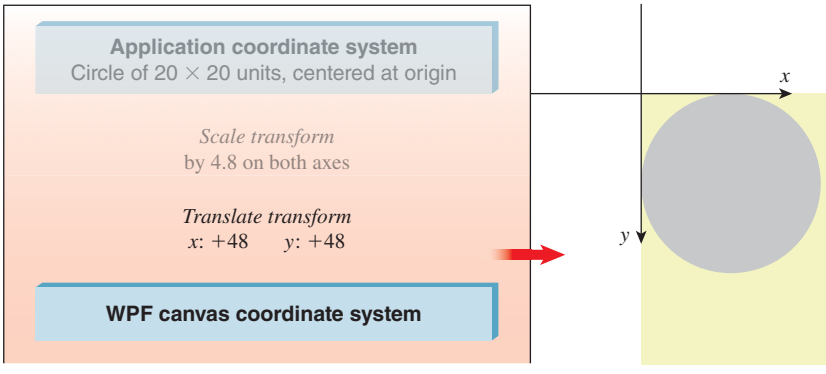


Figure 2.14: Schematic view of our application now enhanced with a two-step display-transform sequence (scale and translate).

Because the display transformation is attached to the canvas, it operates on the entire scene, no matter how large or complex. At this point our scene is a single primitive, but as we continue developing this application and the scene becomes more complex, the value of this display transformation will be more apparent.

Inline Exercise 2.1: Performing the scale *before* the translate is one way to accomplish this display transformation; however, the reverse order will work as well, with different values for the numeric properties. Using the laboratory, visit V.03 and edit the XAML code to reverse the order of the two transforms. First, change the order without adjusting the numeric properties, notice how the rendered scene changes, and then change the properties as needed to restore the desired target rendering.

Inline Exercise 2.2: Note that the circle is "hugging" the top and left side of the canvas. Edit V.03 to move the circle $1/8$ of an inch to the right and $1/8$ of an inch down, to give it some "breathing room." Here again, the correct numeric values will depend on the order of the transforms.

Inline Exercise 2.3: Edit V.03 to add a small blue dot to act as the 12:00 marker.

In Inline Exercise 2.1, you noted the order dependency of a transformation sequence: Scale followed by translate doesn't yield the same results as translate followed by scale. The reason for the order dependency is based on laws of linear algebra. As you will discover in Chapter 12, each transformation, like rotation and translation, is represented internally by a matrix. Sequencing a number of transformations is implemented via matrix multiplication, a noncommutative operation. Thus, it should be no surprise that the order of sequential transformations is important.

2.4.6 Creating and Using Modular Templates

These same transformation utilities are also used for the purpose of constructing a scene by positioning and adjusting copies of reusable stencils called **control templates.**[6] Unlike physical templates that cannot change their size, graphics templates can be rotated, translated, and scaled.

Consider how we might approach defining the hour and minute clock hands. We would like both to share a similar shape, but we'd like the hour hand to be shorter and stouter, a variation that can be achieved via a nonuniform scaling of the same polygon that generates the minute hand. So let's consider how we might construct and place those two hands by defining and using the template shown in Figure 2.15.

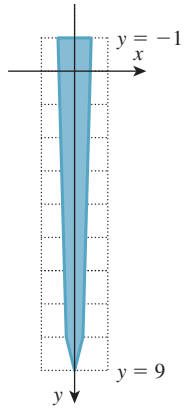


Figure 2.15: Geometry of our clock-hand template.

6. WPF's use of the word "control" in its template nomenclature refers to a typical use of this kind of template: the construction of reusable custom GUI controls.

The WPF element type `Polygon` is used to create an outlined or filled polygon via a sequential (either clockwise or counterclockwise) specification of the vertices. Here is the XAML specification of our canonical clock hand, to be filled with a navy color; note the use of spaces to separate coordinate pairs. Also notice that we define the clock hand using our application's abstract coordinate system.

```
<Polygon
   Points="-0.3, -1   -0.2, 8   0, 9   0.2, 8   0.3, -1"
   Fill="Navy" />
```

We want this `Polygon` element to be a reusable template, defined once and then **instantiated** (added to the scene) any number of times. A control template is specified in the resource section of the root element (in this case, the `Canvas` element). Each template must be given a unique name (using the `x:Key` attribute) so that it can be referenced for the purpose of instantiation.

```
<Canvas ... >

  <!- First, we define reusable resources,
     giving each a unique key: ->
  <Canvas.Resources>
   <ControlTemplate x:Key="ClockHandTemplate">
     <Polygon ... />
   </ControlTemplate>
  </Canvas.Resources>


  <!- THE SCENE ->
  <Ellipse ... />

  <!- THE DISPLAY TRANSFORM ->
  <Canvas.RenderTransform> ... </Canvas.RenderTransform>
</Canvas>
```

If we were to execute our application now, with this new template specification, we would not detect any change. Still, only the gray clock face would be visible. We must instantiate this template to actually change the displayed scene.

To do so, we add a `Control` element—which instantiates by reference to the `ClockHandTemplate` resource—to our scene, resulting in revision V.04:

```
  <!- THE SCENE ->

  <!- The clock face ->
  <Ellipse ... />

  <!- The minute hand: ->
  <Control Name="MinuteHand"
           Template="{StaticResource ClockHandTemplate}"/>
```

How will this new revision of our application look on the screen? Because the display-transformation sequence is attached to the entire canvas, the minute-hand polygon will be subjected to the entire display sequence—in essence, it will "tag along" with the circle through the transformation sequence, as shown in Figures 2.16 through 2.18.

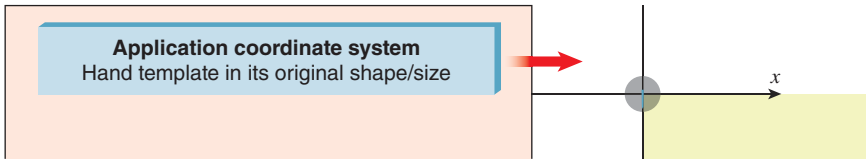


Figure 2.16: Minute hand subjected to the display-transform sequence (1 of 3).

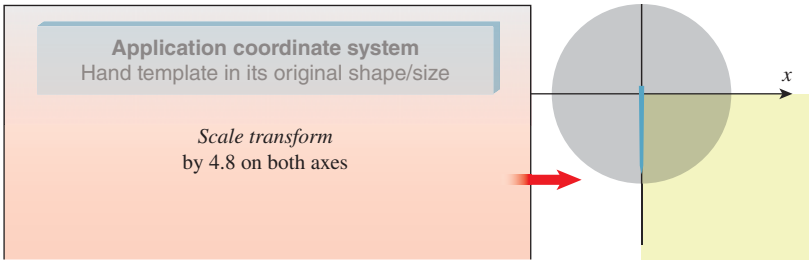


Figure 2.17: Minute hand subjected to the display-transform sequence (2 of 3).

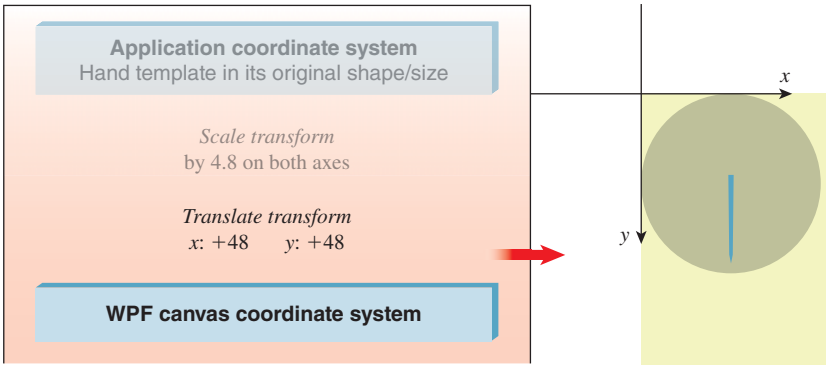


Figure 2.18: Minute hand subjected to the display-transform sequence (3 of 3).

Thus far, it may seem that our use of a template did nothing but make the XAML more complicated. After all, we could have simply specified a `Polygon` after specifying the `Ellipse`. But now, as we build the hour hand—and later, when you perform the suggested exercises—you will appreciate this template strategy.

We will now construct the hour hand via the same approach—instantiating the template—but we are going to make two adjustments.

First, we want to adjust its shape to distinguish it from the minute hand. To do so, we attach a scale transformation to the instance. Although we used scale transforms earlier in this chapter, here we pursue a different goal. Whereas our previous use of a transform sequence was to control our scene's size and placement on the output device—a display transformation—here we are using scaling to construct a component of the scene—what we call a **modeling transformation.** This distinction between two uses of transformations has meaning to us as developers, but is unknown to the underlying platform since the same mechanism—`RenderTransform`—is used for both. (It's a "what-for" distinction, not a "how-to" distinction.)

Second, to make it easy to distinguish between the two clock hands when the full scene is composed, we want to adjust the scene so that they don't both lie on

the *y*-axis overlapping each other. Thus, we are going to rotate the hour hand 45° clockwise, so the clock will show the time of 7:30. To do so, we need the third WPF transformation type, `RotateTransform`:

```
<RotateTransform Angle=... CenterX=... CenterY=... />
```

To instantiate the hour hand, we use the same `Control` tag we used for the minute hand; however, we attach a `RenderTransform` to this instantiation to perform our modeling transformation sequence. This results in the code shown in revision V.05 in the lab.

```
<!- The hour hand: ->
<Control Name="HourHand" Template="{StaticResource ClockHandTemplate}">
   <Control.RenderTransform>
     <TransformGroup>
       <ScaleTransform ScaleX="1.7" ScaleY="0.7" CenterX="0" CenterY="0"/>
       <RotateTransform Angle="45" CenterX="0" CenterY="0"/>
     </TransformGroup>
   </Control.RenderTransform>
</Control>
```

Note that to specify a rotation, you must provide not only the amount of rotation (clockwise, in degrees), but also the center of rotation, which is the point around which the rotation is to occur. One of the nice features of our custom coordinate system is that (0,0) represents the center of the clock, so the origin conveniently serves as the center of rotation for the clock hands (and also as the center point for the scaling operation).

Our scene's XAML specification now has two uses of `RenderTransform` elements: one acting as a modeling transformation (built from two basic transformations) to "construct" the hour hand, and one acting as the display transformation that maps the entire scene to the canvas for display.

```
<Canvas ... >
  <!- RESOURCES ATTACHED TO THE CANVAS ->
  <Canvas.Resources>
    <ControlTemplate x:Key="ClockHandTemplate">
      <Polygon ... />
    </ControlTemplate>
  </Canvas.Resources>

  <!- THE SCENE ->
  <!- The clock face: ->
  <Ellipse ... />
  <!- The minute hand: ->
  <Control Name="MinuteHand"
           Template="{StaticResource ClockHandTemplate}"/>
  <!- The hour hand: ->
  <Control Name="HourHand"
           Template="{StaticResource ClockHandTemplate}">
      <Control.RenderTransform>
            The modeling transform for the hour hand should be here.
      </Control.RenderTransform>
  </Control>
  <!- THE DISPLAY TRANSFORM ->
  <Canvas.RenderTransform>
        The display transform for the scene should be here.
  </Canvas.RenderTransform>

</Canvas>
```

Let's watch the hour hand's progress through the modeling transformation. In Figure 2.19, we see the hand template instantiated with its original geometry; the hand's image is tiny, since this is prior to display transformation, so our schematic includes a magnification callout for clarity.

The first modeling transform is a nonuniform scale that produces the desired shorter, wider shape. The effect of this transformation is the desired hour-hand shape, as shown in Figure 2.20.

The second modeling transformation rotates it into the desired 7:30 location, as shown in Figure 2.21.

The hour hand is now ready to be exposed to the display transformation. It effectively "tags along" with the other members of the scene (clock face and

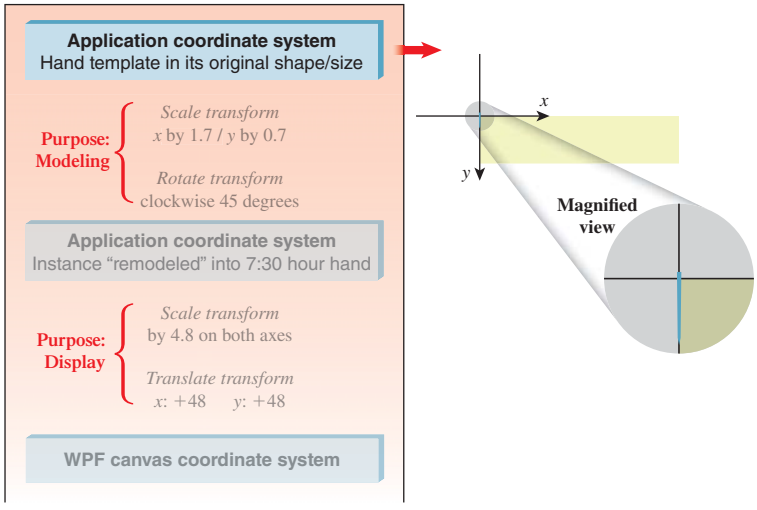


Figure 2.19: Instance of hand template, prior to modeling transform.

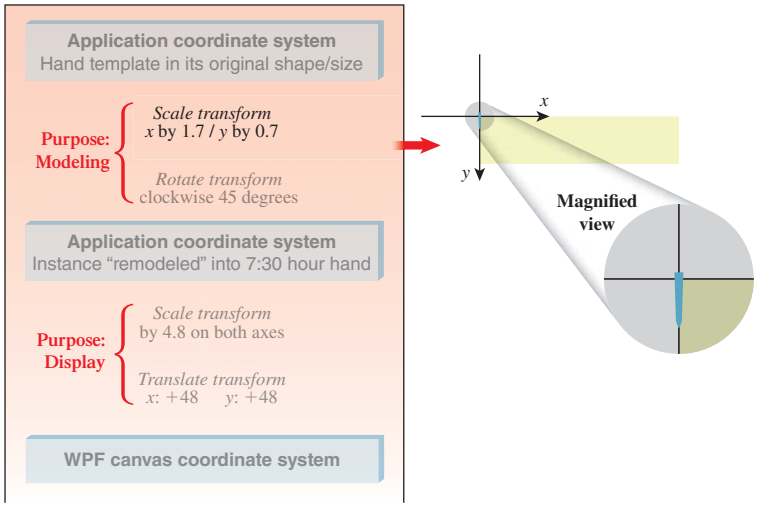


Figure 2.20: Instance of hand template, transformed into hour-hand shape.

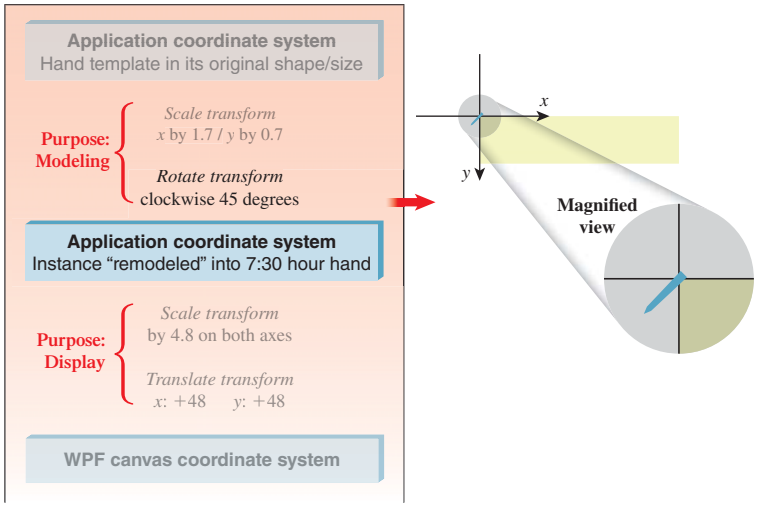


Figure 2.21: Final result of modeling transform that constructs an hour hand at the 7:30 position.

minute hand) through the display-transformation sequence. The result is the full clock image, depicting the time of 7:30. Note that this chapter's online resources include an animation showing the complete operation of this sequence of modeling and display transformations.

Inline Exercise 2.4: To ensure your complete understanding of how we've built the entire static clock scene, launch an XAML development environment and start with just a blank canvas. Add all the XAML code necessary to build a clock scene showing the time of 1:45. Add a 12:00 dot if you wish.

Inline Exercise 2.5: When using the WPF canvas in the recommended manner (`ClipToBounds="True"`), visual information that would lie outside the canvas's bounds is hidden; that is, the image is "clipped" to the canvas boundary.
(a) To see what happens when the canvas is too small to show the entire clock image, use your window manager to radically reduce the size of the window in which the lab software runs.
(b) Jump ahead to read Section 2.6, which describes a couple of the ways a full-featured application might adapt to situations in which the canvas is forced to be too small to show the entire scene. Think about how an application could use WPF display transformations to implement either the zoom-out or the pan/scroll solutions presented in that section.

Inline Exercise 2.6: Construct a thin, red-colored second hand by creating a new resource template with its own distinct polygonal shape. Instantiate it on top of your solution to Inline Exercise 2.4 to test your work. Our solution is in the lab (V.06).

Inline Exercise 2.7: The more complex a template is, the more value its reusability provides. Add some additional visual elements to the clock-hand template (e.g., a thin line bisecting it longitudinally), or give it a more complex shape … and watch how its instances automatically adapt to show the template's new definition.

Hint: The `ControlTemplate` will complain if you put more than one primitive inside it, so you'll need to wrap its contents in a `Canvas` element. (Indeed, `Canvas` is used for multiple purposes, including acting as a general-purpose wrapper around multiple primitives.) Don't put any attributes in the `Canvas` start tag for this usage.

Our use of this simple clock-hand template is a very basic, single-level example of **hierarchical modeling,** which is a sophisticated technique for constructing highly complex objects and scenes. See Chapter 6 for a proper introduction to, and example of, this technique.

2.5 Dynamics in 2D Graphics Using WPF

A retained-mode architecture supports the implementation of simple dynamics; the application focuses on maintaining the scene graph (including keeping it in sync with the application model if one is present), and the platform carries the burden of keeping the displayed image in sync with the scene graph. In this section, we examine two types of dynamics available to WPF applications:

- Automated, noninteractive dynamics in which 2D shapes are manipulated by animation objects specified in XAML
- And traditional user-interface dynamics, in which methods written in procedural code (callbacks) are activated by user manipulation of GUI controls, such as buttons, list boxes, and text-entry areas

2.5.1 Dynamics via Declarative Animation

WPF provides the ability to specify simple animations without procedural code, via XAML **animation elements** that can force object properties to change using interpolation over time. The application creates an animation element, connects it to the property to be manipulated, and specifies the animation's characteristics: starting value, ending value, speed of interpolation, and desired behavior at termination (e.g., that the animation should be repeated when the ending value is reached). Lastly, the application specifies what event should trigger the start of the animation. Once set up, the animation element works automatically, without the need for intervention by the application.

Virtually every XAML element property can be the target of an animation. Examples include the following.

- The origin point of a shape (e.g., the upper-left corner of an ellipse) can be manipulated by an animation element to make the shape appear to vibrate.
- The fill-color, edge-color, and edge-thickness properties of a shape primitive can be manipulated by an animation element to perform feedback animations, such as glowing or pulsing.

- The angle property of a rotation transform can be manipulated by an animation element to make the affected objects rotate.

That last example is of interest to us as clock builders. We can use three animation elements, one for each hand, to provide for the clock's movement.

Let's look at the current status of our hour hand's modeling transform, as designed previously.

```
1   <Control.RenderTransform>
2     <TransformGroup>
3         <ScaleTransform ScaleX="1.7" ScaleY="0.7" />
4         <RotateTransform Angle="45"/> <!- for 7:30 ->
5     </TransformGroup>
6   </Control.RenderTransform>
```

The instance transform already contains a `RotateTransform` placing the hand into a 7:30 position. It would be best to have 12:00 be the hour hand's "normalized" default position, now that we're looking into adding time semantics to our application, so let's change that rotate transform:

```
1 <!- Rotate into 12 o'clock default position ->
2 <RotateTransform Angle="180"/>
```

Additionally, to prepare for automated manipulation of the hand's position, let's add another `RotateTransform` and give it a tag (`ActualTimeHour`) to allow its manipulation by an animator. With these two changes, the `TransformGroup` becomes this:

```
1 <TransformGroup>
2    <ScaleTransform ScaleX="1.7" ScaleY="0.6" />
3    <!- Rotate into 12 o'clock default position ->
4    <RotateTransform Angle="180"/>
5    <!- Additional rotation for animation to show actual time: ->
6    <RotateTransform x:Name="ActualTimeHour" Angle="0"/>
7 </TransformGroup>
```

Now, let's look at the declaration of the animation element that will rotate the hour hand. WPF provides one animation element type for each data type that one might want to automate. To control a rotation's angle, which is a double-precision floating-point number, use the element type `DoubleAnimation`:

```
1 <DoubleAnimation
2        Storyboard.TargetName="ActualTimeHour"
3        Storyboard.TargetProperty="Angle"
4        From="0.0" To="360.0" Duration="1:00:00.0"
5        RepeatBehavior="Forever"
6 />
7
```

The animation is connected to the hour hand through the setting of the `TargetName` and `TargetProperty` attributes, which point to the `Angle` property of the target `RotateTransform` element. The `From` and `To` attributes determine the range and direction of the rotation, and `Duration` controls how long it should take to move the property's value through that range. `Duration` is specified using this convention:

```
1 Hours : Minutes : Seconds . FractionalSeconds
```

The `RepeatBehavior="Forever"` setting ensures that the clock hand will continue moving as long as the application is running; as soon as the value reaches the "To" destination, it "wraps around" to the "From" value and continues the animation.[7]

> You may well wonder about the accuracy of the actual animation, considering how precise the specifications are. Will the animation operate if the CPU is under heavy load or is insufficiently powered?
>
> Although the smoothness of the animation may suffer (become "jumpy") when the CPU is stressed, the image will keep up with where it needs to be at any given time. The animation engine works in an absolute way—newly calculating where the property values should be at the present time—instead of in a relative way based on accumulating deltas. Thus, even if the application has been denied adequate CPU time for a long period, the image will jump to the correct state when the application next receives sufficient CPU cycles for image refreshing.

The final step is to install the animation in the XAML code. We want the animation to start as soon as the clock face is made visible. Thus, we create an `EventTrigger` and use it to set the `Triggers` property of the canvas. A trigger must specify what type of event launches it (in this case, as soon as the canvas's content has been fully loaded) and what action it performs (in this case, a set of three simultaneous animation elements, encapsulated into what WPF calls a `Storyboard`):

```
1  <Canvas ... >
2
3      The specification of the clock scene should be located here.
4
5     <Canvas.Triggers>
6        <EventTrigger RoutedEvent="FrameworkElement.Loaded">
7             <BeginStoryboard>
8                <Storyboard>
9                 <DoubleAnimation
10                  Storyboard.TargetName="ActualTimeHour"
11                  Storyboard.TargetProperty="Angle"
12                  From="0.0" To="360.0"
13                  Duration="01:00:00.00" RepeatBehavior="Forever" />
14
15                  Two more DoubleAnimation elements should be located
16                  here to animate the other clock hands.
17
18               </Storyboard>
19            </BeginStoryboard>
20         </EventTrigger>
21      </Canvas.Triggers>
22
23 </Canvas>
```

7. Other available behavior types include reverse motion (i.e., "bouncing back") and simply stopping (for "one-shot" motions).

Revision V.07 of the laboratory shows this animated clock, but its XAML has been modified to make the hour hand move unnaturally fast to make it easier to notice and test the animation's movement. We recommend the following exercises to those who want to test their understanding of this section.

Inline Exercise 2.8: Study the XAML code of revision V.07, and then do the following.

a. The minute hand is currently instantiated into the scene with no modeling transformation. Add the necessary tagging to attach a transform group to it and install the two rotation transforms (one to place it in the default 12:00 position, the other to facilitate animation). Add the necessary tagging to the storyboard to animate it to rotate once per minute.
b. Set up animation of the second hand similarly.
c. Repair the animation of the hour hand so that it is accurate.
d. If you'd like to demo the clock to a friend, manually change the "default position" rotate transformations to initialize the hands to better approximate the actual time at your location, and then commence execution and watch the clock keep accurate time.
e. The ultimate solution for the clock-initialization problem is to use procedural code to initialize the clock. If you have access to Visual Studio software and tutorials, take this XAML prototype and "productize" it by adding initialization logic to create a fully functional WPF clock application that shows the correct local time.

Inline Exercise 2.9: For a more thorough exercise in template-based modeling and animation, visit the online resources to download instructions for the "Covered Wagon" programming exercise.

2.5.2 Dynamics via Procedural Code

Obviously, there is a limit to the richness of an application built using XAML alone. Procedural code is necessary for the performance of processing, logic, database access, and sophisticated interactivity. WPF developers use XAML for what it's best suited (scene initialization, resource repository, simple animations, etc.), and use procedural code to complete the specification of the application's behavior. For example, in a real clock application, procedural code would be used to determine the correct local time, to support alarm features, to respond to user interaction, etc.

2.6 Supporting a Variety of Form Factors

The wide variety of raster display devices—ranging from small smartphone screens to wall-size LCD monitors—poses a challenge to applications. These problems are similar to the ones faced by a desktop application when its window's

size is decreased beyond a certain reasonable limit. In both cases, the application needs to adapt to changes in form factor.

A well-designed application uses logic to examine its current form factor and adapt its appearance as needed. Let's look at adaptation strategies for both of the key areas in a typical application: its UI area and its scene-display area.

When the screen area allocated for a set of UI controls becomes limited, it is rarely wise to adapt by zooming out (scaling down) the controls. The usability of UI controls, and the user's reliance on "spatial memory" for quick access to commonly used controls, are adversely affected by such a technique. An application can instead respond by elision (e.g., hiding controls that are less often used) or by rearrangement of the layout.

An example of the latter is shown in the three-part Figure 2.22. Part (a) shows the menu bar and toolbar in their optimal layout. If the window's width is reduced significantly, the bars are clipped at the right side, as shown in part (b), and "expansion" buttons labeled "≫" are revealed to provide access to the menus and controls that had to be hidden. Part (c) shows the result of the use of an expansion button to reveal the remainder of the toolbar.

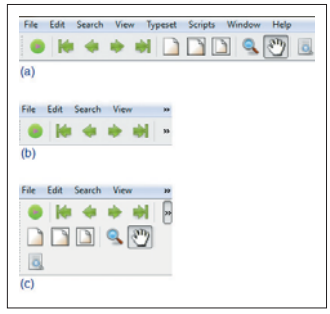


Figure 2.22: Example of automated UI layout adaptation in a Windows application.

A different set of strategies should be considered for scene display when the viewport's size is restricted. Potential solutions include the following.

- The application can zoom out (scale down) the rendering to make more of (or all of) the scene fit within the viewport.
- The application can clip the rendering to the viewport's boundaries and provide an interface supporting panning (scrolling) to access any part of the scene.

These choices are by no means mutually exclusive, and applications often employ a combination of both scaling and clipping; an example, the Adobe Reader thumbnail pane, is shown in Figure 2.23. In this example, the selected document is a long PDF document; think of it as a very tall and narrow scene, one standard page width in width, and 136 standard page heights in height. This application uses a different approach for handling height versus width. For the former, it "clips" the scene and shows only a few pages at a time, providing scrolling features for navigation. For the latter, it uses a zoom-out/scale-down strategy to adapt the scene so that its width exactly matches the width of the pane. The user can choose to widen the thumbnail pane, thus increasing the thumbnails' width, and reducing the intensity of the downscaling, making the thumbnail more "readable".

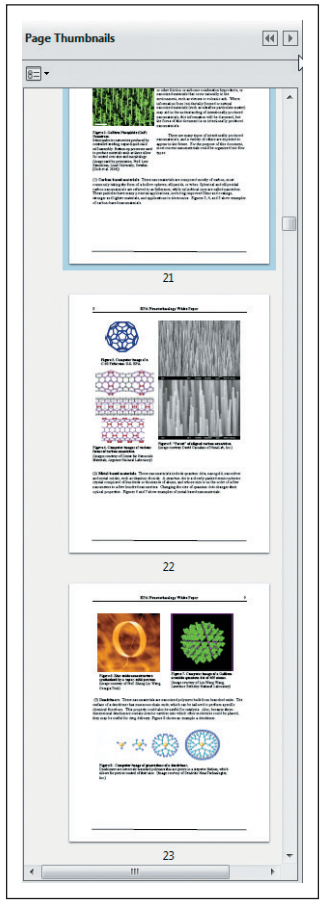


Figure 2.23: Example of automated scene adaptation in a popular PDF viewer.

In both of these example adaptations, the application is responsible for the logic to determine the *policy* to use for a particular form factor/screen size, but the WPF platform provides much of the *mechanism* needed to implement the policy. For example, WPF's UI layout tools simplify UI adaptations like the one described above. And, for scene adaptations, transformations are of great service: Scaling facilitates zooming in/out, and translation facilitates scrolling/panning effects.

2.7 Discussion and Further Reading

In this chapter we've seen how to create collections of primitives in the abstract application coordinate system that is our 2D world, and how to reuse primitives as instances of defined templates. Although we haven't shown templates composed of simpler templates, we will treat that common form of geometric modeling in

our discussion of WPF 3D in Chapter 6. We have made use of transforms, both for modeling and for mapping from abstract application coordinates to WPF canvas coordinates and then on to physical device coordinates; we will discuss the underlying mathematics of transformations in subsequent chapters. Finally, we have seen the advantages of a retained-mode graphics platform for factoring out a number of tasks that are common to many applications, including simple animations. Most importantly, we've introduced the basic features of declarative programming useful for rapid prototyping that can be conveniently extended to do geometric modeling and rendering in WPF 3D.

We have not discussed UI callbacks in response to user interactions like button presses. If such interactions change the thing to be drawn, we must redraw it, just as we must change the application model's state if they change that. Such a callback response is now fairly standard for any program that has a UI, which means most modern programs, and we do not discuss it further.

There's also a second kind of interaction to consider: interaction *within the viewport,* that is, the clicks and drags that the user may perform on the displayed scene. To respond appropriately to these, we often need to know things like which object in the scene the user clicked on, and where the drag started and ended. Determining which object was clicked is known as **pick correlation,** and we discuss this in the context of 3D in Chapter 6. Responding to click-and-drag operations in a 3D scene often requires careful work with the modeling transform hierarchy; we discuss some examples in Chapter 21.

There are a great many other graphics packages in the world, and you're likely to encounter at least a few as you work with computer graphics. We encourage you to browse the Web and read about OpenGL, for instance, and Swing, to get a sense of the variety of features provided by various packages and some of the commonality and differences among them.

Chapter 3

An Ancient Renderer Made Modern

3.1 A Dürer Woodcut

In 1525, Albrecht Dürer made a woodcut demonstrating a method by which one could create a perspective drawing of almost any shape (see Figure 3.1); in the woodcut, two men are making a drawing of a lute. In this chapter, we'll develop a software analog of the method depicted by Dürer.

The apparatus consists of just a few parts. First, there is a long string that starts at the tip of a small pointer, passes through a screw eye attached to a wall, and ends at a small weight that maintains tension in the string. The pointer can be moved around by one person to touch various spots on an object to be drawn.

Second, there is a rectangular frame, with a board, which we'll call the shutter, attached to it by a hinge so that the board can be moved aside (as shown in the woodcut) or rotated to cover the opening, as a shutter covers a window. On the board is mounted a piece of paper on which the drawing is to be made. In the woodcut, you can see a drawing of a lute partially made on the paper. The first man has moved the pointer to a new location on the lute itself. The string passes through the frame, and at the point where it does so, the second man holds a pencil. The string is pushed aside, the shutter is closed, and the pencil makes a new mark on the paper. This process is repeated until the whole drawing (in the form of many pencil marks) emerges. The man holding the pencil must hold it very steady for this to work, of course!

The result is a drawing of the lute consisting of many pencil marks on the paper, which can be connected together to make a more complete drawing. The drawing is a perspective view of the lute, showing the way the lute would look to a viewer whose eye was at the exact point where the string passes through the screw eye on the wall. Note that the height of the screw eye on the wall and the position

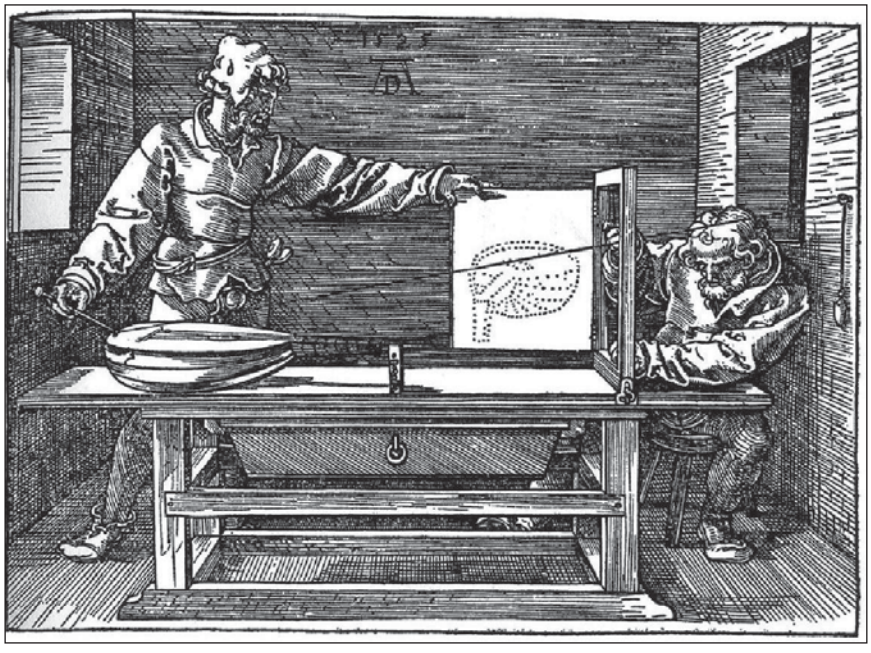


Figure 3.1: Two people using an early "rendering engine" to make a picture of a lute.

of the table can both be changed, so the relative positions of the viewpoint and the drawing should be regarded as parameters of this scene-rendering engine.

The drawing's fidelity can be attributed to three main factors. First, light travels along straight lines, so the stretched string represents a light path from the lute to the eye. Second, the drawing of the lute sits in the scene, and when the shutter is closed, it too sends light to the eye, from a corresponding direction. The points of high contrast in the scene are represented by marks in the drawing, which are themselves points of high contrast. Third, our visual system seems to "understand" a scene largely in terms of high-contrast edges in the scene, so marks on the paper and the real-world scene tend to provoke related responses in our visual systems.

Notice that the string always passes through the frame. If the first man moves the pointer to a place that cannot be seen from the screw eye through the frame, the string will touch the frame itself and bend around it. In this case, no mark is made on the paper.

We'll now make our description of this "rendering" process slightly more formal, as shown in Listing 3.1.

Listing 3.1: Pseudocode for the Dürer perspective rendering algorithm.

```
1  Input: a scene containing some objects, location of eye-point
2  Output: a drawing of the objects
3
4  initialize drawing to be blank
5  foreach object o
6    foreach visible point P of o
7       Open shutter
8       Place pointer at P
```

```
9       if string from P to eye-point touches boundary of frame
10        Do nothing
11      else
12        Hold a pencil at point where string passes through frame
13        Hold string aside
14        Close shutter to make pencil-mark on paper
15        Release string
```

Three aspects of this algorithm deserve note, all within the loop that iterates over all points. First, the iteration is over *visible* points, so determining visibility will be important. Second, there may be an infinite number of visible points. Third, we've said what to do in the event that the string hits the frame rather than passing through the open area bounded by the frame. This is sure to happen if the object is so large that only a part of it is visible from the screw eye through the frame.

We'll discuss the first issue presently; the second is addressed by agreeing to make an *approximate* rendering, which we do by selecting only a finite number of points, but choosing them so that the marks on the paper end up representing the shape well. The process of choosing a finite number of computations to perform, in order to best approximate a result that in theory requires an infinite number of computations, is critical, and will arise in many places in this book. In this chapter, we'll render an object that is so simple that we can set this problem aside for the time being.

The third issue—avoiding drawing points that are outside the view—is also representative of a common operation in graphics. Generally, the process of avoiding wasting time on things outside the **view region** (the part of the world that the eye or camera can see) is called **clipping.** Clipping may be as simple as observing that a point (or a whole object) is outside the view region, or it may involve more complex operations, like taking a triangle that's partly outside the view region and trimming it down until it's a polygon completely inside the view region. For now, we'll use a very simple version of clipping that's appropriate for points only—we'll just ignore points that are outside the view region.

We've made one useful simplification: To determine whether the tip of the pointer is inside the view region (a 3D volume), we check whether the place where the string passes through the frame is within the paper area (rather than the string touching the frame). The two tests are equivalent, but when it comes to implementing them, doing a point-in-rectangle test is easier than doing a point-in-3D-volume test.

Inline Exercise 3.1: Imagine that you can move the lute in Dürer's woodcut.

a. How could you move it to ensure that "touches boundary of frame" is true for each iteration of the inner loop, thus resulting in almost all the work of the algorithm (aside from setup costs) falling into that clause?

b. How would you move it to ensure that no work fell into that clause?

(Your answers should be of the form "Move it closer to the frame and lift it up a little," i.e., they should describe motions of the lute within the room.)

Figure 3.2: A different Dürer rendering approach.

Inline Exercise 3.2: Imagine that, instead of moving the pointer at the end of the string to points on the lute and then seeing where they pass through the paper, the two men start with a piece of graph paper. For each square on the graph paper, the man with the pencil holds it at the center of the square; the shutter is then opened, and the man with the pointer moves it so that the string passes by the pencil point *and* so that the end of the string is touching either the lute, or the table, or the wall. The object being touched is noted, and when the shutter is closed again, the man with the pencil fills in the grid square with some amount of pencil-shading, corresponding to how dim or bright the touched point looks: If it appears dark, the grid square is filled in completely. If it's light, the grid square is left empty. If it's somewhere between light and dark, the grid square is shaded a light grey tone. Think for yourself for a moment about what kind of picture this produces. This approach (working "per pixel" and finding out what's seen through that pixel) is the essence of ray tracing, as discussed in Chapter 14. A slightly different version of this approach, also developed by Dürer, is shown in Figure 3.2: The graph paper is on the table; the corresponding grid in the shutter consists of wires stretched across the shutter, or semitransparent graph paper, and the string and pointer are replaced by the line of sight through a small hole in front of the artist.

The renderings produced by the string-and-pointer method and the graph-paper method are quite different. The first produces an outline of the shapes in the scene (provided the first man chooses his points wisely). The second ignores the scene completely in choosing what to draw, and merely assigns a grayscale value to each grid square. In the event that the scene is very simple—that there are few outlines in the scene—the first method is quite fast. If the scene is very complex (imagine it consists of a bowl full of spaghetti!), then the second method, with its fixed number of grid squares, is faster. Of course, it's only faster because we can, in the physical world, determine which point is visible instantaneously, which we'll discuss further in the next section. In general, though, many techniques in graphics involve tradeoffs that are determined by scene complexity; this is just the first we've encountered.

Now that we have an understanding of this basic rendering process, how can we modernize it so that it becomes useful in computer graphics? We'll focus on creating the drawing, or, more accurately, we'll create a program that models this

method of rendering. Through this, we hope you'll gain insight into the ways we represent the real world in computer graphics. We start with a discussion of visibility.

3.2 Visibility

The matter of selecting only visible points might not even have occurred to Dürer, but it is important for us. One way for Dürer to determine whether a point P is visible is to place the pointer on P and see whether the string runs in a straight line to the screw eye, or has to bend around some other part of the lute or other object to get there. In making our simplified model of the process, we'll disregard the issue of visibility determination, for the time being, not because it's unimportant—Chapter 36 is entirely about data structures for improving visibility computations—but because it is both tricky in general and unnecessary for the model we'll be drawing in this simple renderer.

3.3 Implementation

To mimic the woodcut's algorithm, we'll use algebra and geometry, and a very simple description of a very simple shape: We'll describe a cube by giving the locations of its six corners (or **vertices),** and by noting which pairs of vertices are connected by an edge. Thus, our model of a cube can be considered a **wire-frame model,** in which an object is created by connecting pieces of wire together.

To make the geometry simple, we'll establish a particularly nice way of measuring coordinates in the room. The origin of our coordinate system (see Figure 3.3) will be at the screw eye on the wall, and will be denoted E (for "eye"). The drawing frame will be in the $z = 1$ plane (i.e., we measure the distance from the screw eye to the plane of the drawing frame, and choose a system of units in which this distance is 1). The point on this plane that's closest to the screw eye will be called T; its coordinates are $(0, 0, 1)$. We'll make the y-axis vertical and the x-axis horizontal, as shown.

The frame, within the $z = 1$ plane, will extend from the point $(x_{\min}, y_{\min}, 1)$ to the point $(x_{\max}, y_{\max}, 1)$, where $x_{\min} < x_{\max}$ and $y_{\min} < y_{\max}$, as the names suggest, and where, for simplicity and consistency with the drawing (in which the frame appears fairly square[1]), we'll assume that the width, $x_{\max} - x_{\min}$, is the same as the height, $y_{\max} - y_{\min}$. To be precise, two opposite corners of the frame are points whose 3D coordinates are $(x_{\min}, y_{\min}, 1)$ and $(x_{\max}, y_{\max}, 1)$.

Inline Exercise 3.3: What are the coordinates of the other two corners of the frame?

We'll also imagine that the paper that fills the frame (when the shutter is closed) is actually graph paper, whose lower-left corner is labeled $(x_{\min}, y_{\min})$ and whose upper-right corner is labeled $(x_{\max}, y_{\max})$, therefore providing us with

1. The panel that fills the frame does *not* appear square, however; we'll stick with squareness for simplicity.

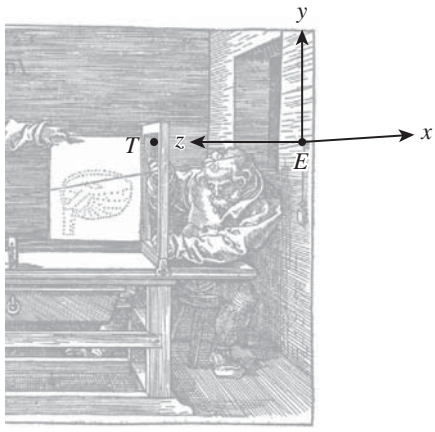


Figure 3.3: The coordinate system for the Dürer woodcut: The origin is at the screw eye, labeled E, and the y- and z-coordinate axes are shown there. The picture frame lies in the plane $z = 1$, parallel to the plane of the wall, $z = 0$. The x-coordinate arrow is horizontal, lying in the plane of the wall, approximately in the direction of the shading lines on the wall, while the z-coordinate arrow is horizontal and perpendicular to the wall. Due to the effects of perspective, the x-direction and z-direction appear almost parallel, but pointing in opposite directions, at the screw eye. The point T is the point in the frame of the drawing plane ($z = 1$) closest to the screw eye. The z-direction points from the screw eye toward T, making the xyz-coordinates of T be $(0, 0, 1)$.

coordinates within the plane of the paper. Thus, to every 3D point $(x, y, 1)$ whose last coordinate is 1, we have associated graph-paper coordinates (x, y).

Now let's suppose that we observe a point $P = (x, y, z)$ of our object, as shown in Figure 3.4; the line from P to E (the string) passes through the frame at a point[2] $P' = (x', y', z')$. We need to determine the coordinates (x', y', z') from the known coordinates x, y, and z.

In Figure 3.5, we've drawn two similar triangles in the $x = 0$ plane. The vertices of the red triangle are (1) the point $E = (0, 0, 0)$, (2) the projection of P' onto the $x = 0$ plane, which is $(0, y', 1)$, and (3) the point $(0, y', 0)$, just below E. The vertices of the blue triangle are (1) the point E, (2) the projection of P onto the $x = 0$ plane, which is $(0, y, z)$, and (3) the point $(0, y, 0)$ well below E.

Similarity tells us that the ratio of the vertical to the horizontal side in the two triangles must be equal, that is, $y'/1 = y/z$. A similar argument, using triangles in the $y = 0$ plane (best visualized by imagining a bird's-eye view of the scene), shows that $x'/1 = x/z$, which can be simplified to give

$$x' = \frac{x}{z}, \tag{3.1}$$

$$y' = \frac{y}{z}. \tag{3.2}$$

2. The use of P and P' to denote a point and another point associated to it can be very helpful in keeping the association in mind; unfortunately, most programming languages disallow the use of primes and other such marks in variable names, so in our code, we must use a different convention.

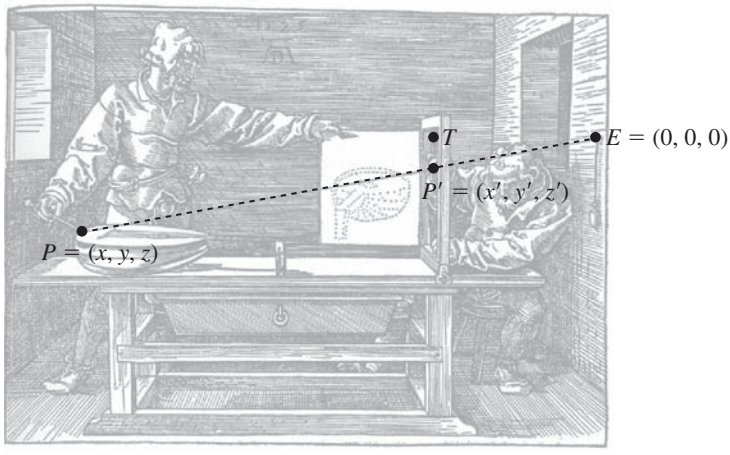


Figure 3.4: The point $P = (x, y, z)$ is on our object. The string from P to the eye, E, will pass through the window frame at some location $P' = (x', y', z')$. Note that $z' = 1$, because we chose our coordinates to make that happen.

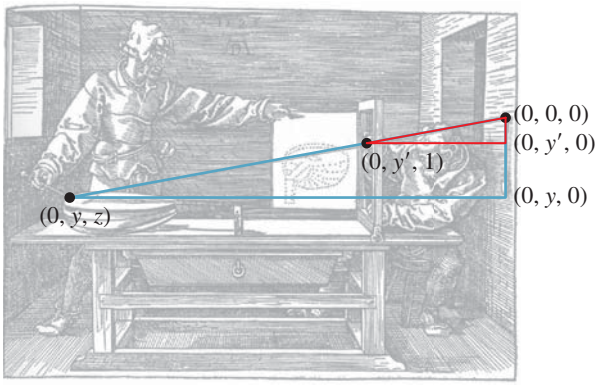


Figure 3.5: Two similar triangles overlaid on the picture in the $x = 0$ plane. The vertical edges of the small red and large blue triangles have lengths y' and y, respectively. What are the lengths of their horizontal edges?

We now know how to find the coordinates of P' from those of P in general! So we can describe our revised algorithm as shown in Listing 3.2.

Listing 3.2: Simple-implementation version of the Dürer rendering algorithm.

```
1  Input: a scene containing some objects
2  Output: a drawing of the objects
3
4  initialize drawing to be blank
5  foreach object o
6    foreach visible point P = (x,y,z) of o
7    if x_min ≤ (x/z) ≤ x_max and y_min ≤ (y/z) ≤ y_max
8       make a point on the drawing at location (x/z, y/z)
```

Let's pause for a moment to examine this: We've now got the x- and y-coordinates of the pencil marks in the picture plane. But if we are to view the eventual picture from the location of the screw eye in the wall, the direction of

increasing x will point to our *left*. (The direction of increasing y will still point up, as expected.) We could choose to plot our points on a piece of graph paper in which we decide that x increases to the left, or we can flip the sign on x to make it increase to the right. We'll do the latter, because it's consistent with the more general approach we'll take later. So we revise the last part of our algorithm to the code shown in Listing 3.3.

Listing 3.3: Minor alteration to the Dürer rendering algorithm.

```
1  if x_min ≤ (x/z) ≤ x_max and y_min ≤ (y/z) ≤ y_max
2     make a point on the drawing at location (-x/z, y/z)
```

To complete our modern-day implementation, we need (1) a scene, (2) a model of the objects in the scene, and (3) a method for drawing things.

For simplicity, our scene will consist of a single cube. Our model of the cube will initially consist of a set containing the cube's eight vertices. A basic cube can be described by the following table:

Index	Coordinates
0	(−0.5, −0.5, −0.5)
1	(−0.5, 0.5, −0.5)
2	(0.5, 0.5, −0.5)
3	(0.5, −0.5, −0.5)
4	(−0.5, −0.5, 0.5)
5	(−0.5, 0.5, 0.5)
6	(0.5, 0.5, 0.5)
7	(0.5, −0.5, 0.5)

Unfortunately, this cube is centered on the eyepoint, rather than being out in the area of interest, beyond the frame. By adding 3 to each z-coordinate, we get a cube in a more reasonable location:

Index	Coordinates
0	(−0.5, −0.5, 2.5)
1	(−0.5, 0.5, 2.5)
2	(0.5, 0.5, 2.5)
3	(0.5, −0.5, 2.5)
4	(−0.5, −0.5, 3.5)
5	(−0.5, 0.5, 3.5)
6	(0.5, 0.5, 3.5)
7	(0.5, −0.5, 3.5)

3.3.1 Drawing

The vertices of a cube are indeed interesting points, but we *should* draw points from all over the surface of the cube to really mimic Dürer's drawing. On closer inspection, however, we see that the points selected by the two apprentices lie on what we could call "important lines" like the outline of the lute, or sharp edges between pairs of surfaces.[3] For the cube, these important lines consist of all the points on the edges of the cube. Drawing all these points (or even a large number

3. The question of which lines in a scene are important is of considerable interest in nonphotorealistic or expressive rendering, as discussed in Chapter 34.

of them) might be needlessly computationally expensive; fortunately, there's a way around this expense: If A and B are vertices with an edge between them, and we project A and B to points A' and B' in the drawing, then we can see that the points between A and B will project to points on the line segment between A' and B'. We could verify this geometrically, or simply rely on the familiar experience that photographs of straight lines always appear straight.[4] So, instead of finding many points on the edge and drawing them all, we'll simply compute A' and B' and then draw a line segment between them.

Inline Exercise 3.4: The preceding paragraph suggests that we can say that "Perspective projection from space to a plane takes lines to lines" or "takes line segments to line segments." Think carefully about the first claim and find a counterexample. Hint: Is our perspective projection defined for every point in space?

A word to the wise: The sorts of subtleties you discovered in this exercise are not mere nitpicking! They are the kinds of things that lead to bugs in graphics programs. Because graphics programs tend to operate on a great deal of data, nearly every possible part of a program will often be tested in a sample execution. Thus, bugs that might survive testing on a less demanding system will often reveal themselves early in graphics programs.

For those who did not come up with counterexamples, we give them here. First, for a line that passes through the eye, the perspective projection is not even defined for the eyepoint. And the non-eyepoints of the line project to a single point rather than to an entire line. If we agree to ignore points at which the projection is undefined, there's a further problem: The eye is at location $(0, 0, 0)$, and the projection plane is parallel to the xy-plane. This means that any line segment that passes through the $z = 0$ plane contains a point that cannot be projected. The projection of such a segment will consist of two separate pieces. Convince yourself of this by projecting the segment from $(\frac{1}{2}, 0, 1)$ to $(\frac{1}{2}, 0, -1)$ onto the $z = 1$ plane by hand.

 In the language of projective geometry, "perspective projection takes extended lines to extended lines, except that it is undefined on the pencil of lines containing the projection point."

Returning to our program, we enhance our model of the cube to include a list of edges, described by the vertex indices of their endpoints:

Index	Endpoints
0	(0, 1)
1	(1, 2)
2	(2, 3)
3	(3, 0)
4	(0, 4)
5	(1, 5)
6	(2, 6)
7	(3, 7)
8	(4, 5)
9	(5, 6)
10	(6, 7)
11	(7, 4)

4. For cameras with high-quality, nondistorting lenses anyhow!

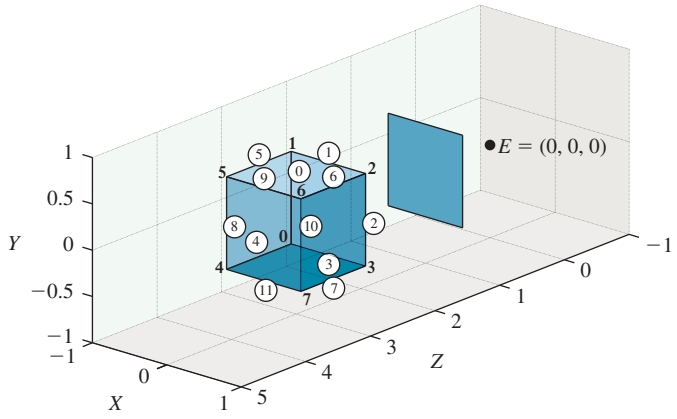


Figure 3.6: The labels for the vertices and edges of our cube model. Edge indices are in circles. The eyepoint and frame from the Dürer woodcut are also included, although we have chosen to adjust their relative positions by placing the frame in this case so that it extends from $-\frac{1}{2}$ to $\frac{1}{2}$ in both x and y. Thus, we're viewing the cube "at eye level" rather than "from above," as Dürer viewed the lute.

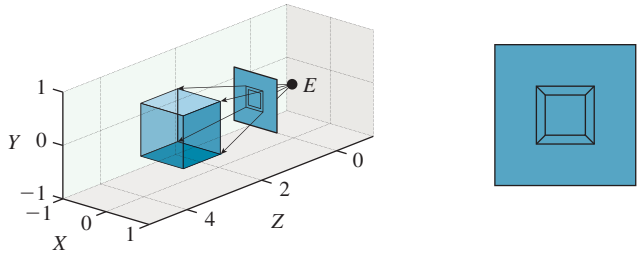


Figure 3.7: The result of the rendering algorithm, (a) shown in place (i.e., drawn in the frame), with rays from the eye to the four near corners of the cube shown, projecting those corners onto the picture plane, and (b) seen directly, with the surrounding square (which ranges from $-\frac{1}{2}$ to $\frac{1}{2}$ in both x and y) drawn to give context.

This enhanced model is shown in Figure 3.6.

Now let's determine a method for drawing this enhanced representation. To update our algorithm to a version that draws lines, we have to make a choice. Do we iterate through the edges, and for each edge, compute where its endpoints project, and then connect them with a line? Or do we iterate through the vertices first, computing where each vertex projects, and then iterate through the edges, using the precomputed projected vertices? Since each vertex is shared by three edges, the first strategy involves three times as many projection computations; the second involves three times as many data-structure accesses. For such a small model, the performance difference is insignificant. For larger models, these are important tradeoffs; the "right" answer can depend on whether the work is done in hardware or in software, and if in hardware, the precise structure of the hardware, as we'll see in later chapters. We'll use the second approach, but the first is equally viable. The results of this approach to rendering the cube are shown in Figure 3.7, both in 3-space and in the plane of the frame.

Furthermore, there's another problem: When we were transforming only points, we could perform clipping on a point-by-point basis. But now that we plan to draw edges, we have to do something if one endpoint is inside and the other

is outside the frame. We'll ignore this problem, and assume that if we ask our graphics library to draw a line segment in our picture, and the coordinates of the line segment go outside the bounds of the picture, nothing gets drawn outside the bounds. (This is true, for example, for WPF.) A preliminary sketch of the program is given in Listing 3.4.

Listing 3.4: Edge-drawing version of the Dürer rendering algorithm.

```
1  Input: a scene containing one object ob
2  Output: a drawing of the objects
3
4  initialize drawing to be blank;
5  for (int i = 0; i < number of vertices in ob; i++){
6      Point3D P = vertices[i];
7      pictureVertices[i] = Point(-P.x/P.z, P.y/P.z);
8  }
9  for {int i = 0; i < number of edges in ob; i++){
10     int i0 = edges[i][0];
11     int i1 = edges[i][1];
12     Draw a line segment from pictureVertices[i0]
13        to pictureVertices[i1];
14 }
```

Finally, we have to observe that this algorithm depends on the "picture rectangle" having coordinates that run from $(x_{\min}, y_{\min})$ to $(x_{\max}, y_{\max})$. We can, however, remove this dependency by using coordinates that range from 0 to 1 in both directions, as is very common in graphics libraries. We can convert the x-coordinates as follows. First, subtract $x_{\min}$ from the x-coordinates; the new coordinates will range from 0 to $x_{\max} - x_{\min}$. Then divide by $x_{\max} - x_{\min}$, and the new coordinates will range from 0 to 1. So we have

$$x_{\text{new}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}}; \tag{3.3}$$

an analogous expression gives us y-values that range from 0 to 1. Because we've required that $x_{\max} - x_{\min} = y_{\max} - y_{\min}$, we get no distortion by transforming x and y this way: Both are divided by the same factor. The program, modified to include this transformation, is shown in Listing 3.5.

Recall that we negated x so that the picture would be correctly oriented. When we negate x_{new}, however, the resulting values will range from -1 to 0. We therefore add 1 to return the result to the range 0 to 1, in line 11 of Listing 3.5.

Listing 3.5: Edge-based implementation with the limits of the view square included as parameters.

```
1  Input: a scene containing one object o, and a square
       xmin ≤ x ≤ xmax and ymin ≤ y ≤ ymax in the z = 1 plane.
2  Output: a drawing of the object in the unit square
3
4  initialize drawing to be blank;
5  for(int i= 0; i < number of vertices in o; i++){
6      Point3D P = vertices[i];
7      double x = P.x/P.z;
8      double y = P.y/P.z;
9      pictureVertices[i] =
10        Point(1 - (x - xmin)/(xmax - xmin),
11                 (y - ymin)/(ymax - ymin));
12 }
```

```
13 for{int i = 0; i < number of edges in o; i++){
14     int i0 = edges[i][0];
15     int i1 = edges[i][1];
16     Draw a line segment from pictureVertices[i0]  to
17          pictureVertices[i1];
18 }
```

These zero-to-one coordinates are often called **normalized device coordinates:** They can be thought of as describing a fraction of the left-to-right or top-to-bottom range of a display device; if the display device isn't square, then a coordinate of 1.0 represents the smaller of the two dimensions. A typical desktop monitor might have vertical coordinates that range from 0 to 1, but horizontal coordinates that range from 0 to 1.33.

The **normalization** process—converting from the range $[x_{\min}, x_{\max}]$ to the range $[0, 1]$—occurs often; it's worth memorizing the form of Equation 3.3.

Inline Exercise 3.5: Verify, in Listing 3.5, that a vertex that happens to be located at the lower-left corner of the view square, that is, $(x_{\min}, y_{\min}, 1)$, does indeed transform to the lower-left corner of the final picture, that is, the corresponding `pictureVertex` is $(0, 0)$; similarly, verify that $(x_{\max}, y_{\max}, 1)$ transforms to $(1, 1)$.

3.4 The Program

We'll use a simple WPF program, based on a standard test bed described extensively in the next chapter, to implement this algorithm. The resultant program is downloadable from the book's website for you to run and to experiment with. For this application, the critical elements of the test bed are the ability to create and display dots (small disks that indicate points) and segments (line segments between dots) on a `Canvas` that we call the `GraphPaper`. Positions on our graph paper are measured in units of millimeters, which are more readily comprehended than WPF default units, which are about $\frac{1}{96}$ of an inch. To make the drawing a reasonable size, we'll multiply our algorithmic results (coordinates ranging from zero to one) by 100. The important parts of the program are shown in Listing 3.6, with elisions indicated by `[...]`.

Before you examine the code, though, we'll warn you that this use of WPF is very different from what you saw in Chapter 2. In that chapter, we demonstrated the declarative aspects of WPF that are easy to expose through XAML. The test bed uses these declarative aspects to build a window, some menus, and controls, and to lay out the `GraphPaper` on which we'll do our drawing. But the production of actual pictures within the `GraphPaper` is all done in C#. That's because for most of the programs we'll want to write, expressing things in XAML is either cumbersome or impossible (not every feature of WPF is exposed through XAML). In general, it makes sense to use the declarative specification whenever possible, especially for layout and for data dependencies, and to use C# whenever substantial algebraic manipulations may be needed.

Listing 3.6: C# portion of the implementation of the Dürer algorithm.

```
public Window1()
{
    InitializeComponent();
    InitializeCommands();

    // Now add some graphical items in the main Canvas,
        whose name is "Paper"
    gp = this.FindName("Paper") as GraphPaper;

    // Build a table of vertices:
    int nPoints = 8;
    int nEdges = 12;

    double[,] vtable = new double[nPoints, 3]
    {
       {-0.5, -0.5, 2.5}, {-0.5, 0.5, 2.5},
       {0.5, 0.5, 2.5}, ...};
    // Build a table of edges
    int [,] etable = new int[nEdges, 2]
    {
       {0, 1}, {1, 2}, ...};

    double xmin = -0.5; double xmax = 0.5;
    double ymin = -0.5; double ymax = 0.5;

    Point [] pictureVertices = new Point[nPoints];

    double scale = 100;
    for (int i = 0; i < nPoints; i++)
    {
        double x = vtable[i, 0];
        double y = vtable[i, 1];
        double z = vtable[i, 2];
        double xprime = x / z;
        double yprime = y / z;

        pictureVertices[i].X = scale * (1 - (xprime - xmin) /
          (xmax - xmin));
        pictureVertices[i].Y = scale *       (yprime - ymin) /
          (ymax - ymin);
        gp.Children.Add(new Dot(pictureVertices[i].X,
          pictureVertices[i].Y));
    }

    for (int i = 0; i < nEdges; i++)
    {
          int n1 = etable[i, 0];
          int n2 = etable[i, 1];

          gp.Children.Add(new Segment(pictureVertices[n1],
              pictureVertices[n2]));
    }

    ...
}
```

We begin with a few comments. First, the code is not at all efficient (e.g., there was no need to declare the variables `x`, `y`, and `z`), but it follows the algorithm very closely. For making graphics programs that are debuggable, this is an excellent place to start in general: Don't be clever until your code works and you

find that you *need* to be clever. Second, we've used names for all the important things that we might consider changing, like `xmin` and `nEdges`. That's because most test programs like this one survive far longer than originally intended, and get altered for other uses; symbolic names help us communicate with our former selves (who wrote the program originally). Third, the program is not "software engineered": We didn't define a class to represent general shapes, for instance—we just used an array of a fixed size to represent one shape, the cube. That was deliberate. The program was meant to be used, experimented with, and thrown away (or perhaps reused in another experiment). The point of this program was to verify our understanding of a simple concept. It's not a prototype for some large-scale project. Indeed, if you find yourself building something of even a modest scale atop this framework, you're making a big mistake: The pieces of this framework were designed to make testing and debugging easy. If you run the program and place your cursor over one of the dots, for instance, a tool-tip will appear telling you the dot's coordinates; the same goes for the edges. That makes sense when you're debugging, but by the time you're drawing 10,000 edges, it's a huge amount of overhead. Remember that this framework is a test bed for experiments, and that all the code you write within it should be considered disposable. While this advice may seem contradictory—we've said to throw away code, but to code decently because you'll be reusing it—it reflects our experience: Even code we intend to throw away *does* get reused, often for other throwaway applications! It's worth investing a little time to make these future uses easier, but not worth investing so much that it takes hours to code up a simple idea and confirm your understanding.

The results are what we expect (see Figure 3.8): a perspective picture of a wire-frame cube. We've created our first rendering! There's clearly a long way to go

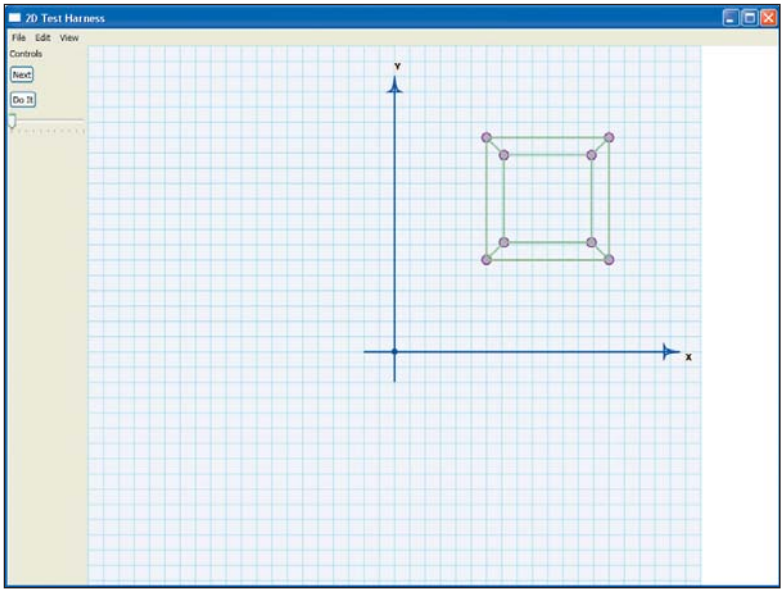


Figure 3.8: The result of the Dürer program: a wire-frame cube, shown in perspective, on a background that looks like graph paper. The axes on the graph paper are part of the `GraphPaper` *itself, set up by the test bed, and are not drawn by the Dürer rendering part of the program.*

from this to the kinds of special effects seen in video games and Hollywood films, but some of the important ideas—building a mathematical model, converting it to a 2D picture[5]—are present in our simple renderer in a basic form.

3.5 Limitations

We will now step back and look critically at this success. When you run the program, you see a perspective view of a cube, consisting of 12 line segments (with dots at the corners of the cube as well), as expected.

There are several limitations to the program. First, the wire-frame drawing means that we can see both the back and the front of the cube. We could address this in much the same way we addressed the drawing of edges: We could observe that all points on a **face** of a cube (one of the square sides of the cube) will project to points in the quadrilateral defined by the projections of the four vertices. So, instead of keeping an edge table, we could keep a face table, and for each face, we could draw a filled polygon in two dimensions. (We could draw both faces *and* edges, of course.) Using this approach, our main concern will be to find a way to draw only polygons that face toward the eye rather than away from it. There are many strategies for doing this, but all of them require either more mathematics than we wish to introduce in this chapter, or more complex data structures than we wish to discuss at this time.

Second, although we all know that we see objects because of the light they emit (which enters our eyes and causes us to perceive something, as described in Chapters 5 and 28), there's nothing in our current program that describes anything about light (except that our use of straight lines for projection is based on the understanding that light travels along straight lines). We would get the same picture of a cube whether we imagined any light being present or not. This also means that all kinds of other light-dependent features, like shadows and shaded parts of the surface, cannot manifest themselves. It's *possible* to add these without an explicit model of light, but it's a mistake to do so, according to the Wise Modeling principle.

Third, when this program runs, it only shows one model (the cube), and only shows it in one position. We did a lot of work for not much output; our program is not versatile enough to display other scenes without changing the code. This can be addressed by storing the data representing the cube in a file that can be read by the program; a typical representation would begin with a vertex count, a list of vertices, an edge count, and a list of edges. Although it's less compact, it's wise to make such files contain explicit tags on the data, and to allow comments; it will make debugging your programs much easier. Here's an example file for representing a cube:

```
1 # Cube model by jfh
2 nVerts: 8
3 vertTable:
4 0   -0.5  -0.5  -0.5
5 1   -0.5   0.5  -0.5
6 ...
7 7    0.5   0.5   0.5
8 # Note that each edge of cube has length = 1
9
```

5. We're using the term "picture" informally here, as in "something you can look at."

```
edgeTable:
0   0  1
1   1  2
...
11  7  8
```

Other formats are certainly possible; indeed, there are many, many formats for specifying models of all sorts, and programs for interconverting them (sometimes losing some data in the conversion). Because the choice of formats is subject to the whims of fashion, and changes quickly, we'll make no attempt to survey them.

With any such storage format, one can define multiple shapes, such as the cube, a tetrahedron, or even a faceted sphere, and enhance the program to load each one in turn, adding some variety. To do so will require that you understand the test bed more fully by reading parts of the next chapter, however.

We can also enhance the program by adding a limited form of animation: The *xy*-coordinates of the bottom (or top) four vertices of our cube are four equally spaced points on a circle of radius $r = \sqrt{2}/2$, namely $r(\cos\theta, \sin\theta)$, where $\theta = \frac{\pi}{4}, \frac{3\pi}{4}, \frac{5\pi}{4}$, and $\frac{7\pi}{4}$. We can create a slightly rotated cube by making $\theta = \frac{\pi}{4}+t$, $\frac{3\pi}{4}+t$, $\frac{5\pi}{4}+t$, and $\frac{7\pi}{4}+t$ for the four corners, for some small value of t. By gradually increasing t, and redrawing the model each time, we can display a rotating cube.

This method of explicitly changing the coordinates of the cube and then redisplaying it is not particularly efficient. The cube effectively becomes a **parameterized model,** with the rotation amount, t, serving as the parameter. The problem is that when we want to rotate the cube[6] in the *yz*-plane instead of the *xy*-plane, we need to change the model. And if we want to rotate first in one plane, then in the other, we must do some messy algebra and trigonometry. It's actually far simpler to model the cube just once, and then learn how to transform its vertices by a rotation (or other operations). We'll discuss this extensively in the next several chapters.

On the other hand, there *are* models that are defined parametrically, and are animated by changing these parameters. So-called "spline" models are a particularly important example, discussed in Chapter 22, but others abound, particularly in physical simulations: A model of fluid, for instance, has parameters like the viscosity and the density of the fluid, as well as the initial positions and velocities of the fluid particles. The effects of these parameters on the appearance of the fluid at some time are rather indirect—we have to perform a simulation to understand the effects—but it is a parameterized model nonetheless.

3.6 Discussion and Further Reading

The "rendering" in this chapter was a little unusual, in the sense that we converted our 3D scene into a collection of 2D things (points and lines), which we then drew with a 2D renderer. In this sense, the process somewhat resembles a compiler that turns a high-level language into low-level assembly language. Only when this assembly language is further processed into machine language and executed does the computation take place. In the same way, only when we actually draw

6. We speak of rotation in the *xy*-plane rather than "around the *z*-axis," because rotation in a plane generalizes to all dimensions, while rotation about an axis is specific to three dimensions. Chapter 11 discusses this further.

the points and lines with WPF's 2D rendering tools do we get a picture. Transformations like this into intermediate representations occur elsewhere in graphics as well. In some expressive rendering algorithms, input images are transformed into an intermediate representation that consists of edges detected by some image-processing algorithms and regions bounded by those edges, for instance. Choosing a good intermediate representation can make the difference between success and failure.

The discovery of perspective in Western art, and the working out of the associated mathematics, is a fascinating subject. When lines that are parallel in the world appear to converge in the rendered picture, the eye is drawn to the point of convergence, or **vanishing point.** Almost as fascinating as the development of perspective is the clever use by artists of this property of vanishing points; sometimes artists created different vanishing points for different regions of a picture, drawing the viewer's eye to multiple things in the scene (*The Resurrection,* by Piero della Francesca, is said to have this property.) It's not known whether this was in fact deliberate. There are many systematic ways of creating proper perspective renderings using so-called "vanishing points" and ideas from projective geometry. These too can be considered basic rendering engines like the one shown in the Dürer woodcut.

Also intriguing is the understanding one can gain of non-Western art. Rock's book on perception [Roc95], for instance, explains that the view seen in some Chinese scroll paintings, which appears odd to the Western eye, is actually a perspective-correct view for a view from a very high point, with the projection plane perpendicular to the ground plane.

Perspective projections do not preserve relative lengths (think of a picture of railroad tracks disappearing into the distance—the equal distances between adjacent railroad ties become unequal distances in the picture), but they do preserve some other properties. The study of projections and the transformations of space that are associated to them became the field of **projective geometry;** for the mathematically curious, the books of Hartshorne [Har09]and Samuel [SL88] are excellent introductions.

Representations of polyhedral models by arrays of vertices together with arrays of faces described by vertex indices are sometimes called **indexed face sets;** a large repository of models stored in this way has been collected in the Brown Mesh Set [McG]. While not particularly compact, it is an easy-to-parse format. Similarly simple is the `PLY2` format. Many example models are available on the Internet in both forms. More complex model formats, using more compact binary representations, abound. One fairly common format was `3DS`, developed for use with the 3D Studio Max software (now known as 3ds Max), and widely adopted or imported by other tools. 3ds Max now uses the `.max` format, which is proprietary, but many `3DS` models are still available. And Maya, another popular shape-modeling program, has its own proprietary format, `.mb`. Both are essentially meta-formats, which specify the plugin (shared library) that should be used to parse each subpiece of a model; in practice, it's impractical to reverse-engineer such a format, and as a result, for hobbyist and classroom use people continue to use older and simpler formats.

What we called the "view region" in this chapter—the portion of the world that's visible in the picture we're making—is an instance of the more general notion of **view volume;** the difference is that a view volume, rather than being an infinite pyramid, may be truncated so that objects farther away than some distance

are ignored, as are those closer than some other distance. These "near" and "far" distances can be adjusted to make some algorithms more efficient (by reducing the set of objects we need to consider during rendering) or more accurate (by using fixed-point arithmetic to more precisely represent a range of values). The general matter of defining a view volume is discussed extensively in Chapter 13.

3.7 Exercises

Exercise 3.1: Suppose that the apprentice holding the pencil in the Dürer drawing not only made a mark, but also wrote next to it the height (above the floor) of the weight at the end of the string. That number would be the distance from the eye to the object (plus some constant). Now suppose we make a first drawing-with-distances of a lute, whose owner then takes it home. We later decide that we'd really like to have a candlestick in the picture, in front of the lute (i.e., closer to the eye).

(a) If we place the candlestick on the now-empty table, but in the correct position, and make a second drawing-with-distances, describe how one might algorithmically combine the two to make a composite drawing showing the candlestick in front of the lute. This idea of **depth compositing** is one of many applications of the z-buffer, which you'll encounter again in Chapters 14, 32, 36, and 38. The recording of depths at each point is rather like the values stored in a z-buffer, although not identical.

(b) Try to think of other things you might be able to do if each point of an image were annotated with its distance from the viewpoint.

Exercise 3.2: The dots at the corners of the rendered cube in Figure 3.8 appear behind the edges, which doesn't look all that natural; alter the program to draw the dots *after* the segments so that it looks better. Alter it again to not draw the dots at all, and to draw only the segments.

Exercise 3.3: We can represent a shape by faces instead of edges; the cube in the Dürer program, for instance, might be represented by six square faces rather than the cube's 12 edges. We could then choose to draw a face only if it faces toward the eye. "Drawing," in this case, might consist of just drawing the edges of the face. The result is a rendering of a wire-frame object, but with only the visible faces shown. If the object is convex, the rendering is correct; if it's not, then one face may partly obscure another. For a convex shape like a cube, with the property that the first two edges of any face are not parallel, it's fairly easy to determine whether a face with vertices $(P_0, P_1, P_2, \ldots)$ is visible: You compute the cross product[7] $\mathbf{w} = (P_2 - P_1) \times (P_1 - P_0)$ of the vectors, and compare it to the vector $\mathbf{v}$ from the eye, E, to P_0, that is, $\mathbf{v} = P_0 - E$. If the dot product of $\mathbf{v}$ and $\mathbf{w}$ is negative, the face is visible. This rule relies on ordering the vertices of each face so that the cross product $\mathbf{w}$ is a vector that, if it were placed at the face's center, would point into free space rather than into the object.

(a) Write down a list of faces for the cube, being careful to order them so that the computed "normal vector" $\mathbf{w}$ for each face points outward.

(b) Adjust your program to compute visibility for each face, and draw only the visible faces.

(c) The cross- and dot-product based approach to visibility determination described in this exercise fails for more complex shapes, but later in the book

7. We review the dot product and cross product of vectors in Chapter 7.

we'll see more sophisticated methods for determining whether a face is visible. For now, give an example of an eyepoint, a shape, and a face of that shape with the property that (i) the dot product of **v** and **w** is negative, and (ii) the face is not visible from the eye. You may describe the shape informally. Hint: Your object will have to be nonconvex!

Exercise 3.4: As in the preceding exercise, we can alter a wire-frame drawing to indicate front and back objects in other ways. We can, for instance, consider all the lines of the object (the edges of the cube, in our example) and sort them from back to front. If two line segments do not cross (as seen from the viewpoint) then we can draw them in either order. If they *do* cross, we draw the one farther from the eye first. Furthermore, to draw a line segment (in black on a white background), we first draw a thicker version of the line segment in white, and then the segment itself in black at ordinary thickness. The result is that nearer lines "cross over and hide" farther lines.
(a) Draw an example of this on paper, using an eraser to simulate laying down the wide white strip.
(b) Think about how the lines will appear at their endpoints—will the white strips cause problems?
(c) Suppose two lines meet at a vertex but not at any other point. Does the order in which they're drawn matter? This "haloed line" approach to creating wire-frame images that indicate depth was used in early graphics systems [ARS79], when drawing filled polygons was slow and expensive, and even later to help show internal structures of objects.

Exercise 3.5: Create several simple models, such as a triangular prism, a tetrahedron, and a $1 \times 2 \times 3$ box, and experiment with them in the rendering program.

For the final two exercises, you may need to read parts of Chapter 4.

Exercise 3.6: Enhance the program by adding a "Load model" button, which opens a file-loading dialog, lets the user pick a model file, loads that model, and makes a picture of it.

Exercise 3.7: Implement the suggestion about displaying a rotating cube in the program. Add a button that, when the cube is loaded, can update the locations of the cube vertices by computing them with a new value of t, the amount to rotate. To make the animation look smooth, try changing t by .05 radians per button click.

This page intentionally left blank

Chapter 4

A 2D Graphics Test Bed

4.1 Introduction

Now that you are familiar with some aspects of WPF and have seen our promised test bed in use in building the Dürer renderer, we show you the details of the test bed—a framework for testing out ideas in graphics without a huge amount of overhead. There are actually two of these: one for 2D and one for 3D. This chapter introduces the 2D test bed; more complete documentation, along with the 3D test bed, are available on the book's website. We refer to these as test beds because they resemble the test apparatus that an electrical engineer might use: a collection of instruments, power sources, and prototyping boards on which various circuits can be assembled and tested. Our design aims are modest: We want a basic framework in which it's easy to write and debug simple programs. Ease of debugging is favored over speed; simplicity is favored over generality.

Throughout this book, we present exercises that involve writing small programs designed to investigate some subject that you've studied. For instance, when we discuss the mathematical modeling of shapes, we start with polygons and show how to make curves shaped by them. Figure 4.1 shows an example of this: You can take a closed **polyline**—a connect-the-dots sequence of segments in a plane—and perform a **corner-cutting** operation on it in which each segment is divided into thirds, the first and last thirds of each segment are removed, and the remaining segments are joined up in sequence.

It appears that if you do this repeatedly, the resultant polyline becomes increasingly smooth, approximating a smooth curve. While it's possible (and valuable) to analyze this process mathematically, we believe that a picture is worth a thousand words and that interaction is worth many thousands more. We therefore give you the tools to answer a variety of questions you may have—such as whether the curve will *always* get smoother, whether there are starting polylines that result in curves that have sharp corners, regardless of the number of iterations, and what happens if you keep the middle *half* of each segment instead of the middle third—so that you can get a visceral understanding of the process. At the same time, writing programs to implement such ideas frequently helps you understand subtleties.

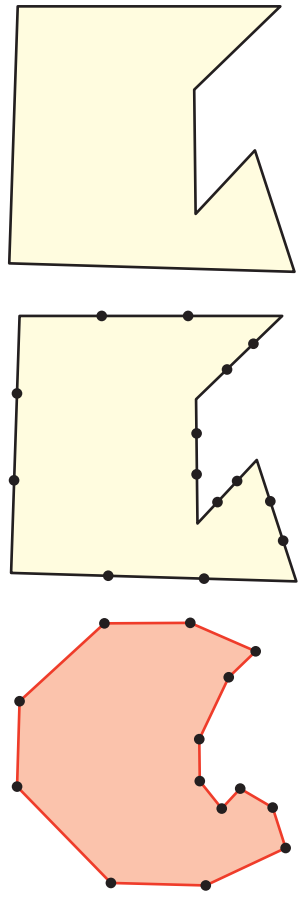


Figure 4.1: Top: A polyline in the plane. Middle: Each segment has been divided into thirds, and the division points have been marked with dots. Bottom: The middle thirds of the segments are connected together to form a new, smoother polyline.

If we chose to discuss corner cutting for polylines rather than *closed* polylines, you would have to determine what to do with the starting and ending segments as you write the code; such boundary cases are often the source of both complexity and understanding, as well as errors.

We *could* have opted to explain how to draw individual points, and then how to draw lines, detect mouse clicks, and create menus and buttons and associate them with actions in your program, all of which are interesting and valuable topics. But we want you to be able to start learning graphics *visually* right away. To do that requires that we give you some tools whose inner workings you won't yet understand.

With this goal in mind—providing easy-to-modify programs in which you can experiment with various ideas in graphics—the remainder of this chapter presents our 2D test bed.

Following the introduction of the 2D program, we show how to use it to implement corner cutting in 2D, and provide some exercises in which you can conduct some experiments that will prepare you for the ideas presented in later chapters.

The test-bed program itself is meant to isolate you, at this point, from understanding many of the details of a graphics program. All on-screen graphics eventually involve setting the color values for individual pixels on the screen. In Chapter 38 we discuss the underlying software and hardware that work at this low level. For now, we are letting that hardware and software do the heavy lifting for us so that we can work at a higher level of abstraction: We assume that we can create high-level shapes and images and that WPF and Direct3D will take care of making the pixels look the way we indirectly said they should.

4.2 Details of the Test Bed

As we said, our program is based on WPF, a subject to which entire books and thousands of web pages are devoted. In our experience, it is always easier to modify an existing program than to start from scratch. We therefore built the 2D test bed and included a sample program with it that does a number of things. You'll typically use the test bed by copying this sample, deleting most of it, and then modifying the remainder. The sample displays things such as a photographic image, an image created in software, a polygon, a mesh, and a quiver of arrows (a collection of arrows specified by their basepoints and directions). It also has some buttons (one of which changes which software image is displayed) and a slider (which can be used to move one vertex of a polygon). From these examples, it should be easy for you to generalize and add your own interaction elements to the program.

Note that this program is constantly under development; we anticipate augmenting it as students indicate ways in which they would like to see it enhanced. Because those augmentations may cause minor changes to the program, we will include notes about any such changes on the book's website. The website also includes more thorough documentation of the entities included in the test bed; this chapter is an introduction to the test bed and its use, not the complete documentation.

4.2.1 Using the 2D Test Bed

At this point in reading this chapter, you should pause, visit the book's website, and download the 2D test bed as directed there. You should also prepare a

development environment in which you would like to work. In the examples in this book, we used Visual Studio 2010 in a freely distributed "basic edition," along with the Microsoft Windows SDK for browsing documentation, among other things. Follow the directions on the website for getting the 2D test bed up and running, and then experiment with the sample program: Try clicking a button or moving a slider and see what happens. Browse through the code to see if you can begin to piece together how it works. Then read on.

We assume that you are familiar with an integrated development environment like Visual Studio and that you know a programming language like C#, C++, or Java. (The test bed is written in C#, but anyone familiar with C++ or Java will find it easy to use C#.)

4.2.2 Corner Cutting

In Section 4.6, we develop the corner-cutting application step by step. To prepare for that exercise, download the `Subdivide` application from the book's website and then run it. When the application starts, click several times in the main window to create a polygon; then click on the Subdivide button to see the corner-cutting operation, or the Clear button to start over.

Examine the `Window1.xaml` code. You'll see the words "Subdivide" and "Clear"; the XAML code around these words creates the buttons you were clicking, and `Click="b1Click"` tells WPF that when one of those buttons is clicked, a procedure called `b1Click` should be invoked. We'll see more details of this later; here, we want you to gain a broad picture of where the working parts of this example lie.

Now examine the `Window1.xaml.cs` code. The initializations for the Window1 class create a pair of `Polygon` objects, which are initialized in the constructor for `Window1` and are then `add`ed to something called `gp` (for "graph paper"), which represents everything that will be drawn on the display. The polygon initialization code sets up certain properties for the polygons; lots of other properties could also be set, but these are left with their default values. Finally, the `b2click` and `b1click` procedures describe what should happen when the user clicks on the two buttons. Look at them briefly. The handling of the Clear button should seem reasonably simple to you; the handling of the Subdivide button is more complicated, but you can see that at its core, it involves multiplying various coordinates by $1/3$ and $2/3$, as you'd expect.

Those are the essential parts of the corner-cutting application. Almost everything else is **boilerplate**—the bits of code that make it easy to write applications like this. In fact, we created this corner-cutting application by starting with the test bed, which creates points, lines, arrows, a mesh, and various moving things, deleting most of them, and then editing the XAML to remove some user-interface components that weren't needed and renaming a few others. With this example in mind, we can now look at the rest of the test bed.

4.2.3 The Structure of a Test-Bed-Based Program

As you learned in Chapter 3, WPF applications are typically specified in two parts: one in XAML, the other in C#. The two parts cooperate to make the whole. Indeed, one can have object classes that have this same two-part separation, or that are purely C#; our code has both types.

The highest-level portion of the application is called `Testbed2DApp`; it is implemented in the files `Testbed2DApp.xaml` (the XAML file) and `Testbed2DApp.xaml.cs` (the associated C# file).

The XAML file (see Listing 4.1) declares that `Testbed2D` is an object of class `Application`, which means that it has certain properties, events, and methods predefined; we use almost none of these, with the exception of the `Startup` event handler, which we'll see in the C# file.[1]

Listing 4.1: The code in `Testbed2DApp.xaml`.

```
<Application x:Class="Testbed2D.Testbed2DApp"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Startup="AppStartingUp">
  <Application.Resources />
</Application>
```

The code in Listing 4.1 simply declares an application and some information about where to find certain name resolutions (the `xmlns` lines) for this XML namespace. The key element, from our point of view, is the line `Startup="AppStartingUp"`, which says that the `Startup` event is to be handled by code that will be found in the `AppStartingUp` method of the `Testbed2D.xaml.cs` file. This is the equivalent of `main()` in a C++ or Java program.

The corresponding C# file is shown in Listing 4.2. The keyword `partial` tells us that part of the class's description is here, but part of it is elsewhere (in the XAML file). The `AppStartingUp` method has been defined to create a `Window1` and to show it. The arguments to `AppStartingUp` are unused. The remaining event handlers, methods, etc., for the `Testbed2DApp` remain unchanged from the defaults inherited from the `Application` class.

Listing 4.2: The corresponding C# file, `Testbed2DApp.xaml.cs`.

```
using System;
using ...

namespace Testbed2D
{
  public partial class Testbed2DApp : Application
  {
     void AppStartingUp(object sender, StartupEventArgs e)
     {
         Window1 mainWindow = new Window1();
         mainWindow.Show();
     }
  }
}
```

So, if we run the `Testbed2DApp`, upon startup a `Window1` will be created and shown.

This `Window1` class is somewhat richer than the `Testbed2DApp` class: It corresponds to the main window of a conventional application and includes things like

1. Other events are things like `OnExit`, which occurs when the program exits, and `Activated`, which occurs when the application becomes the foreground application; all the details of every class in WPF are documented online, but part of the goal in the test bed is to shield you from having to know most of them.

a menu bar, buttons, and sliders, as well as a large area in which we can draw things. The arrangement of these items is determined in the `Window1.xaml` file: When you want to add a button to the test bed, you'll edit that file; when you want to arrange a connection between a slider drag and a certain action in your program, you'll edit that file, etc.

The `Window1.xaml.cs` file is concerned almost entirely with creating the contents of the one large area in which we can draw things.

We'll examine both of the `Window1` files now. We'll omit large sections of each, sections that are repetitive or boilerplate, and concentrate on the details that you'll use as you write your own programs.

A complex XAML file like `Window1.xaml` (see Listing 4.3) can describe several things at once: **layout** (the positioning of things within the window), **event handling** (what happens when you press a keyboard key or click on a button), **styles** (the font in which text appears, the color of a button), etc. For now, we'll concentrate on the layout aspects. Start by examining the code and trying to figure out what's going on, and we'll explain a few details shortly.

Listing 4.3: The XAML code for the test bed.

```
<Window
    x:Class="Testbed2D.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/ ..."
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:h="clr-namespace:Testbed2D"
    Title="2D Testbed"
    KeyDown="KeyDownHandler"
    Height="810"
    Width="865"
    >
  <DockPanel LastChildFill="True">
    <Menu DockPanel.Dock="Top">
      <MenuItem Header="File">
        <MenuItem Header="New" Background="Gray"/>
        <MenuItem Header="Open" Background="Gray" ...
      </MenuItem>
      <MenuItem Header="Edit"/> ...
    </Menu>

    <StackPanel DockPanel.Dock ="Left" Orientation="Vertical" Background="#ECE9D8">
      <TextBlock Margin="3" Text="Controls"/>
      <Button Margin="3,5" HorizontalAlignment="Left" Click="b1Click">Next </Button>
      <Button Margin="3,5" HorizontalAlignment="Left" Click="b2Click">Do It </Button>
      <Slider Width="100" Value="0" Orientation="Horizontal"
        ValueChanged="slider1change" HorizontalAlignment="Left"
        IsSnapToTickEnabled="True" Maximum="20" TickFrequency="2"
        AutoToolTipPlacement="BottomRight" TickPlacement="BottomRight"
        AutoToolTipPrecision="2" IsDirectionReversed="False"
        IsMoveToPointEnabled="False"/>

    </StackPanel>
    <h:GraphPaper x:Name="Paper">
    </h:GraphPaper>
  </DockPanel>

</Window>
```

First, there are several namespaces involved in this code: We use standard WPF entities, which are defined in a WPF namespace, as well as entities like

our GraphPaper class, which are defined in the Testbed2D namespace. The initial xmlns statements say that we'll use the namespaces that WPF requires. We'll explain the last two xmlns statements shortly.

Recall that each large-scale element (Window, DockPanel, StackPanel, etc.) is paired with a closing for that element (/Window, /DockPanel, etc.). Between these are other items that constitute the content of these large-scale elements. Thus, everything in the XAML file is between Window and /Window, indicating that everything will be within a window.

Examining the XAML, we see that the window contains a DockPanel, which in turn contains everything else. A DockPanel is a **panel** (a rectangular area of the window) in which other elements can be placed, and such elements are automatically placed relative to the boundaries of the DockPanel in a way indicated by the XAML. For instance, line 17,

```
<Menu DockPanel.Dock="Top">
```

indicates that we want a Menu in the DockPanel, and we want it docked (attached) at the Top of the panel. Other options are Bottom, Left, Right, and None. The dock panel also contains a StackPanel, docked at the left, and a GraphPaper, whose docking position is unspecified. Because the DockPanel has its LastChildFill property set to True, this GraphPaper will fill up all the available space in the DockPanel. We'll soon discuss what a GraphPaper actually *is;* for now, suffice it to say that it's a specialized kind of Canvas.

WPF transforms this entire specification of the layout into a user interface that looks the way you've specified it should look. There's no need to say how many pixels tall the menu bar is, for instance, and if the user resizes the application while it's running, reasonable things will happen automatically to adjust the layout appropriately. This is one of the enormous advantages of the XAML portion of WPF: The language for specifying the appearance of a user interface is very high-level.

Continuing down a level, it's easy to read the MenuItem blocks of XAML: They say that there's a File menu with elements New and Open, and an Edit menu, among other things. (Some of these use the <TAG ... /> syntax, in which a final / in a tag replaces the closing </TAG>.)

Inline Exercise 4.1: Modify the menu bar to add a new menu, named Foo, with menu items named Bar and Baz, and rerun the program to ensure that they appear. Then remove them.

In the StackPanel (which simply adds elements from the top down, "stacking" them from top to bottom[2]) we find a TextBlock, two Buttons, and a Slider. The TextBlock serves as a label for this panel (it says "Controls"). The buttons and slider let us control the appearance of the canvas. The Margins in each tell WPF how much room we'd like around the sides of each element in the StackPanel; the HorizontalAlignment tells WPF how to position the item within any additional space. The Click=b1Click specifies what method (in this case, b1click) should be called when the user clicks the Next button. And finally, between the <Button>

2. This top-to-bottom order is because the StackPanel has its Orientation set to Vertical; if it were Horizontal, the stacking would be left-to-right.

and </Button> tags, we have the button's content, which is just a simple piece of text.

The Slider in lines 25–30 is similar: We've set several options, specifying the slider's width, its initial value (0), its maximum value (20, when the slider thumb is all the way to the right), the fact that it should be horizontal, where to place tick marks, and how many to place. The most interesting feature is the ValueChanged=slider1change, which means that when the user changes the slider's value, WPF should call the slider1change method of the Window1 class.

We've now examined almost all the XAML code. You could probably, with some confidence, edit this file to add a few more buttons and a few more sliders, or even figure out how to change the colors of the buttons, or break the StackPanel into a pair of StackPanels, one for buttons and one for sliders, side by side. (Hint: You could include both in a new DockPanel with its orientation being horizontal.)

Inline Exercise 4.2: Add a new button or slider to the StackPanel, but don't include the Click= or ValueChanged=, respectively. Run your program to ensure that the new item actually appears where you expect it.

Inline Exercise 4.3: Rearrange the Controls StackPanel, as suggested above, into one panel full of buttons and another panel containing only sliders. Verify that your modification works.

If you were to add a new button, set Click=b3Click, and then try to compile the application, it would fail because you'd need to write the b3click method of the Window1 class; we'll discuss this in more detail in a moment.

Inline Exercise 4.4: Add a button with Click=b3Click, and verify that the application no longer works. Try to parse the error message and make sense of it. Then remove the new button.

The last item in the XAML code is a GraphPaper. The syntax here is a little peculiar. Listing 4.4 shows the XAML code, much reduced.

Listing 4.4: The part of Window1.xaml that creates a GraphPaper.

```
1  <Window
2      x:Class="Testbed2D.Window1"
3      xmlns= ..."
4      xmlns:h="clr-namespace:Testbed2D"
5      ...>
6      ...
7      <h:GraphPaper x:Name="Paper">
8      </h:GraphPaper>
9   </DockPanel>
10
11 </Window>
```

The highlighted xmlns line indicates that the XML namespace called h refers to the Common Language Runtime (clr) namespace defined by Testbed2D. This means that GraphPaper is not a standard WPF class, but rather a class defined by this project, a class called GraphPaper instead of WPF's Canvas class. In fact, a GraphPaper is a lot like a Canvas (it is, in fact, *derived* from Canvas), except that it

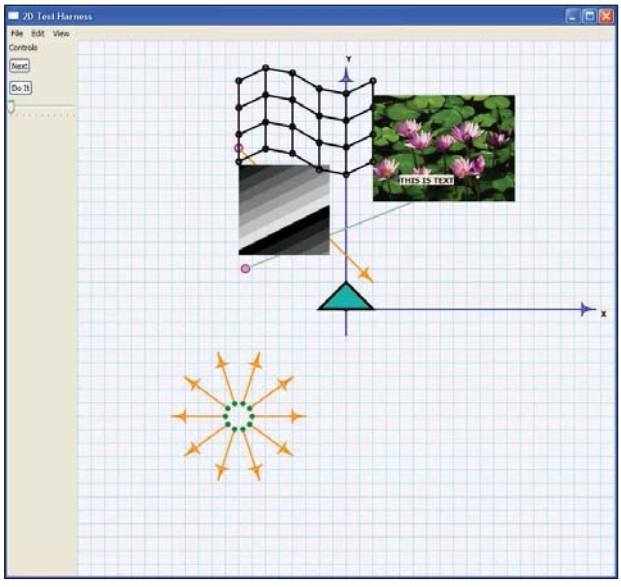


Figure 4.2: The `Testbed2D` *application running. All the things in the large graph-paper window are drawn by the* `Window1.xaml.cs` *code and are not described in the XAML file.*

comes with a graph-paper-like grid and coordinate axes predrawn, and distances on a `GraphPaper` are measured in millimeters rather than in WPF units (which are sized at 1/96 of an inch).[3] We won't examine the entire description of the `GraphPaper` class; we'll just use it. After you've read the next two chapters, it'll be worth your time to examine the XAML and C# that together define `GraphPaper`.

Down at the bottom of the `Window1` XAML file we create an instance of a `GraphPaper`; to do so, we have to say that it's an `h:GraphPaper`, telling the program what namespace to find it in. We also give it a name—`Paper`—by which we can refer to it within the C# file.

Figure 4.2 shows the appearance of the program when it's run. All the interesting content is generated by the C# code, but the entire appearance of the interface is determined by the XAML code.

4.3 The C# Code

The associated C# code (`Window1.xaml.cs`) is also relatively simple (see Listing 4.5). There is no fancy software engineering involved; this test bed is meant to act as the equivalent of a pad of scratch paper, not as a foundation for large systems. If you plan to build a large application atop WPF, you should get familiar with WPF itself, not the distilled portion of it that we have used for this 2D test bed.

Recall that we've only *partially* described a `Window1` in the XAML; the C# file contains the remainder of the definition. Because we're displaying a polygon,

3. There's one exception: If you set your display's dpi setting to something that does not match the actual number of dots per inch of your physical display, then a WPF unit will not appear as 1/96 of an inch.

three images, a mesh, and a quiver of arrows in this sample, we'll declare each of those as an instance variable for the class. If you were to write a program that needed to display only a single image, you'd delete all but one of these.

Listing 4.5: The C# portion of the `Window1` class definition.

```
using ...

namespace Testbed2D
{
    public partial class Window1 : Window
    {
      GraphPaper gp = null;

      Polygon myTriangle = null;
      GImage myImage1 = null;
      GImage myImage2 = null;
      Mesh myMesh = null;
      Quiver myQuiver = null;

      // Are we ready for interactions like slider-changes to
      // alter the parts of our display (like polygons or images
      // or arrows)?
      // Probably not until those things have been constructed!
      bool ready = false;

      public Window1()
      {
          InitializeComponent();
          InitializeCommands();

          // Now add some graphical items in the main Canvas,
          // whose name is "GraphPaper"
          gp = this.FindName("Paper") as GraphPaper;

          // A triangle whose top point will be dragged
          // by the slider.
          myTriangle = new Polygon();
          myTriangle.Points.Add(new Point(0, 10));
          myTriangle.Points.Add(new Point(10, 0));
          myTriangle.Points.Add(new Point(-10, 0));
          myTriangle.Stroke = Brushes.Black;
          myTriangle.StrokeThickness = 1.0; // 1 mm thick line
          myTriangle.Fill = Brushes.LightSeaGreen;
          gp.Children.Add(myTriangle);

          // A draggable Dot, which is the basepoint of an arrow.
          Dot dd = new Dot(new Point(-40, 60));
          dd.MakeDraggable(gp);
          gp.Children.Add(dd);

          Arrow ee = new Arrow(dd, new Point(10, 10),
                       Arrow.endtype.END);
           gp.Children.Add(ee);
[lots more shape-creating code omitted]
           ready = true;  // Now we're ready to have sliders and
                          // buttons influence the display.
           }
[interaction-handling code omitted]
```

The constructor for `Window1` first initializes the component—a step that's required for any top-level WPF window, causing all the subparts to be laid out;

it then initializes the menu- and keyboard-command handling. Following this, we add the graphical items to the `GraphPaper` that we named `Paper`; we locate that with the `FindName` method. A `Canvas` has a `Children` collection, and with `gp.Children.Add(myTriangle)`, we make the triangle we've created a child of the `Canvas`, which makes it get displayed within the `Canvas`.

Now let's look at the details of the creation of the triangle. We declare the triangle to be a new `Polygon` and then add several `Point`s to the `Polygon`. This describes the geometry of the triangle, but not its appearance. In WPF, appearance is characterized by a `Stroke` (how lines are drawn) and a `Fill` (how regions are filled in), each of which is defined by a `Brush`, which can be remarkably complex; in our case, we'll use simple predefined strokes and fills, available from the `Brushes` class. The lines will be drawn in black, and the triangle will be filled with a light green color. The thickness of the stroke is set to `1.0`; because of the way `GraphPaper` is defined, that's 1.0 mm. Indeed, recall that all units in a `GraphPaper` refer to millimeters. The grid is based on 5 mm distances between grid lines, and the triangle we've created will be 10 mm tall (although with a brush thickness of 1.0, it will actually be slightly taller; we imagine the brush being dragged around the outline, with the brush *center* passing along the geometric figure[4]). In summary, the `GraphPaper` called `Paper` provides a place in which to draw geometric shapes; the units used on the graph paper are millimeters, and the coordinates have been set up so that $(0, 0)$ is positioned more or less in the center of the canvas, with the first coordinate increasing as we move to the right and the second coordinate increasing as we move up. This latter coordinate direction is *not* the choice made in WPF, in which the second coordinate would increase as we move downward.

4.3.1 Coordinate Systems

Why does the second coordinate increase downward in WPF? There are a number of possible justifications for this (some of which may not even have been in the designers' minds), and some reasonable criticisms as well. The natural criticism is that anyone who has studied mathematics is used to the conventional Cartesian coordinates, in which the y-axis points upward; those only slightly more familiar with mathematics are used to angles measured from the x-axis and increasing as the other ray of the angle moves *counterclockwise*. Why, with all these years of experience, would one change things?

The counterargument is that several *other* things are naturally described with a coordinate that increases as we move *downward*. Matrices, for instance, have row and column indices, with the first row at the top, the second row below it, etc. If we think of a matrix as containing a collection of grayscale values, we can think of it as describing a black-and-white image. It would be nice if, when it was displayed "in the obvious way," the resultant image looked the way we expected from the matrix representation (see Figure 4.3). But if we use Cartesian coordinates in the obvious way, the resultant image ends up upside down. [By the way, there's the further problem that matrix indices are almost always given as "(row, column)" pairs; unfortunately, the column corresponds to horizontal position and the row

4. The brush in this case actually draws lines that are mitered at the corners so that the shape remains triangular; this can be adjusted as a property of the `Brush`, however. At very sharp corners, the miter can extend a long distance; because of this, one can also set a `MiterLimit`.

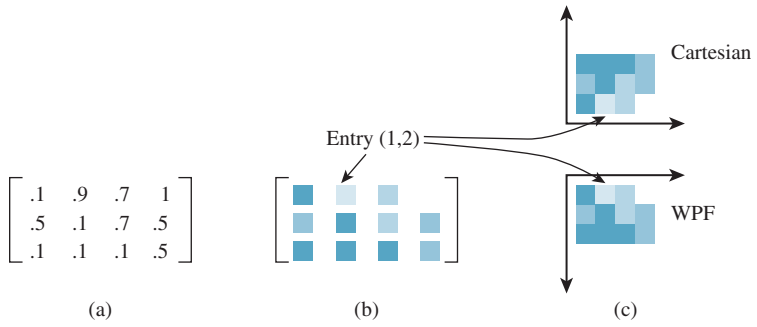


Figure 4.3: (a) A 3×4 *matrix with entries between 0 and 1 representing various shades of gray, from black (0) to white (1). If we convert each entry to a small rectangle of the corresponding shade of blue, we get the "shaded matrix" shown in (b). (c) If we display the matrix by displaying the values in column j and row i as a small square centered at location* (i, j) *in the Cartesian plane, the resultant image is flipped across the horizontal axis. If, on the other hand, we do the same thing in WPF coordinates, the result is* not *inverted.*

to vertical position, which is exactly the opposite of the (x, y) convention of both Cartesian and WPF coordinates.]

A further argument, beyond images, is that text in Western writing flows from top to bottom and from left to right, and thus our way of thinking about 2D layout also flows in that way. Furthermore, our conventional interfaces start at the top (that's where the menu bar is) and draw our eyes downward into the content. Finally, many early raster graphics systems tended to describe the individual pixels in this way, with $(0, 0)$ in the top-left corner of the screen.

Regardless of the rationale, this is the convention chosen by the WPF developers. Fortunately, they also included a mechanism by which one can change the coordinates used in a canvas.[5] Thus, our test bed has conventional Cartesian coordinates.

A second version of the test bed, using the *y*-increases-down coordinate system, is also available on the book's website. You'll probably choose to use that version when you're experimenting with image data, for instance.

4.3.2 WPF Data Dependencies

Our triangle is a `Polygon` made from three `Point`s. Built into WPF is the capability to determine when things have changed in a way that necessitates redrawing. When we say that the `GraphPaper` is to add this triangle to its collection of children, we're not saying anything about actually drawing the triangle. When WPF decides that it needs to display the canvas, it displays all the canvas's children. We have simply told it what one of those children *is*. One thing that prompts WPF to redisplay the canvas is the information that one of the children has changed. For example, if we removed the triangle as a child, the canvas would be automatically

5. More precisely, they included something that says how canvas coordinates should be converted to coordinates in the containing object—a window, a panel, etc.—and this allows us to invert the *y*-coordinate and to move $(0, 0)$ to the center of the canvas, much as we did with the clock example in Chapter 2.

redisplayed, with the triangle absent. If we somehow *changed* the triangle, that too would prompt a redisplay.

Unfortunately, this "watching for changes in children" extends to only a certain level of detail. If the `Collection` of `Point`s that makes up the `Polygon` changes, this in fact becomes a change in the `Polygon` itself, which prompts a redisplay of the containing `GraphPaper`. But what does it mean for a `Collection` to change? The designers of WPF chose to make it mean "the collection of references must change" rather than "one of the objects referred to by the references must change." Thus, deleting or inserting an item in the collection is treated as a change, but merely altering an item—for example, changing the *x*-coordinate of the first point—is not. So, if we alter the first coordinate of the first `Point` that defines the polygon, nothing happens: The `Point` itself changes, but the `Collection` containing the `Point` does not recognize the change and propagate it. Instead, we can *remove* the `Point` from the collection, create a new `Point` with a changed coordinate, and insert this new `Point` into the collection, resulting in a change that propagates upward. We'll see details of this in the next section.

The choice of what constitutes a "change" in a system like WPF can have a huge impact on performance: If it's too fine-grained, all the platform's computational power will be spent watching for changes; if it's too coarse-grained, application programmers will need to do lots of change notifications and may eventually abandon the system-provided change tracking in favor of handling it all themselves, because they can then do so consistently.

4.3.3 Event Handling

WPF takes user interaction, in the form of key presses, mouse clicks, and mouse drags, and treats them as **events.** When an event is detected, it sets off a complex sequence of activities, which ends with some event handler being called by WPF. To be more precise, a great many event handlers may be called by WPF, but in our case, we'll use just a single handler for an event and then mark the event as "handled" so that no further handlers will be called. Sometimes the event handlers are part of WPF; sometimes they are callback procedures provided by the application programmer.

In particular, when WPF detects that a button has been clicked, it invokes that button's `Click` handler. In the case of our first button (defined on line 25 in Listing 4.3), we set the `Click` handler to be `b1Click` back in the XAML code. The `b1Click` method is rather simple: It prints a debugging message and then declares that this click has been handled by setting a flag on the event (see Listing 4.6).

Listing 4.6: The handler for a click on the first button.

```
1   public void b1Click(object sender, RoutedEventArgs e)
2   {
3       Debug.Print("Button one clicked!\n");
4       e.Handled = true; // don't propagate the click any further
5   }
```

The `sender` in this code is the entity in WPF from which the click was relayed to us; this relaying of clicks is part of a complex hierarchy, one that we can largely ignore, in which a click on a piece of text displayed on a button in a grid on a canvas will trigger responses from the text object, the button, the grid, and the canvas. To break this chain of events, we set the `RoutedEventArgs` `Handled` field

to `True`, which says that we've processed the button click, and no other part of our code needs to do anything about it.

A more complex example of event handling is demonstrated by the handler for a change in the value of the slider, namely `slider1change`, which is shown in Listing 4.7. As you can see, we print a message indicating the new value (`e.NewValue`) that was passed in, and then adjust the *x*-coordinate of the triangle's first point. Rather than removing and reinserting the point, we simply reassign the entire `Points` array, since there are only three points. This causes the triangle to be marked as "changed," which provokes a redisplay of the entire canvas, with the result that as we adjust the slider with the mouse, the top of the triangle moves from side to side.

Listing 4.7: The `slider1change` method that moves a triangle vertex.

```
1   void slider1change(
2      object sender,
3      RoutedPropertyChangedEventArgs<double> e)
4   {
5       Debug.Print("Slider changed, ready = " + ready +
6                   ", and val = " + e.NewValue + ".\n");
7       e.Handled = true;
8       if (ready)
9       {
10          PointCollection p = myTriangle.Points.Clone();
11          Debug.Print(myTriangle.Points.ToString());
12          p[0].X = e.NewValue;
13          myTriangle.Points = p;
14      }
15  }
```

What about the flag called `ready` in lines 19 and 50 of Listing 4.5, and line 8 of Listing 4.7? Well, in the course of creating the `Window1` in our application, WPF creates the slider and sets its initial value to the predetermined value in the XAML code. That in turn raises a `ValueChanged` event,[6] which makes WPF invoke the `slider1change` method. But all this happens in the course of the `InitializeComponent` process, so the triangle has not yet been created. If we try to alter the coordinates of one of its `Point`s, that'll be a bug. We therefore ignore all events like this until the `Window1` is ready, as indicated by the `ready` flag.

Inline Exercise 4.5: Modify the test bed so that it displays a diamond-shaped object rather than a triangle, and make the slider adjust the *x*-coordinate of both the top and bottom vertices at the same time (and in the same direction). Add a second slider, and make it vary the *y*-coordinate of the left and right vertices of the diamond too.

4.3.4 Other Geometric Objects

The test bed also supports several kinds of geometry besides polygons. These include dots, which are like points but are displayed and have a tool-tip that says where they're located; arrows (used for depicting vectors) and quivers, which are collections of arrows; segments, which are line segments between either points or

6. The value has changed from "undefined" to the specified initial value.

dots; meshes; and images. The details of these various entities are given on the book's website.

4.4 Animation

As you saw in Chapter 2, WPF contains tools for creating animations. You can animate almost any kind of value—a `double`, a `Point`, etc.—and then use that changing value to make the display change. You can define animations in XAML code or in C#. For XAML, there are many predefined animations that you can combine to make quite complex motions. In C# code, you can use these predefined animations or you can create your own using arbitrarily complex programs. You might, for instance, write a program that computes the position of a bouncing ball over time, and then use that varying position to control the location of some shape (like a disk) shown on the display. Writing the simulation program in XAML isn't really feasible, so it's more natural to write such a program in C#.

In our sample, we have just one animation, but it shows off the key ideas (see Listing 4.8). In the code, we constructed a segment with one endpoint being at a `Dot` called `p1`. In this code, we animate a point by specifying the starting and ending locations, how long the animation should take (it starts at time 0, i.e., when the program is run, and takes five seconds), and the fact that it should auto-reverse and should repeat forever. (This relatively simple animation is an example of something that could be easily described in XAML, by the way.) The result is a point that moves back and forth between the two specified locations over time. Note that this point is not actually displayed on the `GraphPaper`; it's just a `Point` whose value changes constantly. But the line `p1.BeginAnimation(Dot.PositionProperty, animaPoint1);` says that the `PositionProperty` of the `Dot` called `p1` should be animated by `animaPoint1`; this causes that `Dot` to move back and forth on the display. Once again, WPF's dependency mechanism is doing the hard work: It is detecting every change in `animaPoint1` and making sure that the `PositionProperty` of `p1` is changed as well; this property is what determines the location of the `Dot` on the canvas so that we see motion.

Listing 4.8: Code to animate a `Point`.

```
1  PointAnimation animaPoint1 = new PointAnimation(
2      new Point(-20, -20),
3      new Point(-40, 20),
4      new Duration(new TimeSpan(0, 0, 5)));
5  animaPoint1.AutoReverse = true;
6  animaPoint1.RepeatBehavior = RepeatBehavior.Forever;
7  p1.BeginAnimation(Dot.PositionProperty, animaPoint1);
```

We've been working with `Point`s quite freely, assigning values to their *x*- and *y*-coordinates. Rigid adherence to object-oriented programming doctrine might require that we treat these coordinates as instance variables to be hidden, and that we access them only through accessor/mutator (or get/set) methods. In fact, a `Point` is rather more like a Pascal `record` or a C `struct` than a typical C++ or Java object. In blurring the distinction between **records** (collections of related values) and objects, C# allows such uses, which can have a huge impact on efficiency with relatively little impact on debuggability.

4.5 Interaction

We already discussed how button clicks and slider-value changes are handled in the test bed (and in WPF in general): A `Click` or `ValueChanged` method is called.

Keyboard interaction is a little different. Key presses that happen anywhere in the main `Window` are handled in two stages: First, some of them are recognized as commands (like "Alt-X" for "Exit the program"); second, those not recognized as commands are handled by a method called `KeyDownHandler`, which responds to all key presses by either ignoring them (for presses on modifier keys like Control or Shift) or displaying a small dialog box indicating which key was pressed. As you write more complex programs, you may want to adapt this part of the code to do particular things when certain keys are pressed.

Finally, the actions taken when menu items are selected get defined in the `InitializeCommands` method. For many commands (like `Application.Close`, indicating that the application should close a window), there's already a predefined way to handle them and a predefined key press associated with the command. For these, one can simply write things like `CommandBindings.Add(new CommandBinding(ApplicationCommands.Close, CloseApp));` where `CloseApp` is a small procedure that pops up a dialog to confirm that the user does, in fact, want to close the application. For others, slightly more complex mechanisms must be invoked. As we do not expect you to add any new commands, we leave it to you to decide whether to examine the mechanism for doing so.

4.6 An Application of the Test Bed

Let's now return to the corner-cutting example we discussed at the beginning of the chapter. To create an application that demonstrates this, we'll need to remove most of the code in `Window1.cs` and start with a simple polygon created by the user. We'll describe the interaction sequence and then write the code.

The `GraphPaper` starts out empty; there are two buttons, labeled "Clear" and "Subdivide". When the user clicks on the graph paper a polygon $P1$ appears, with vertices at the clicked locations (after two clicks, the polygon consists of two identical line segments; after three, a triangle; etc). When the user clicks the Subdivide button, a subdivided version, $P2$, of the first polygon appears; subsequent clicks on Subdivide replace $P1$ with $P2$, and $P2$ with a subdivided version of $P2$, etc. so that a polygon and its subdivision are always both shown. A click on the Clear button clears the graph paper. Once the user has subdivided the polygon, we disable further clicks on the graph paper; you might want such clicks to add new points to the subdivided polygon, but it's not clear where, in the subdivided polygon, the new points ought to be added, and so we simply avoid the issue.

With that description in mind, let's write the code.[7] We'll need an `isSubdivided` flag (initially `false`) to tell us whether the user has subdivided the polygon or not. The Clear button should reset this, as well as clearing the graph paper. And if there are no points in the polygon yet, Subdivide should have no effect.

We start from a copy of the test bed code, and modify the XAML to remove the slider and to change the text on the buttons:

7. The entire program is available on the book's website for download.

```
<StackPanel DockPanel.Dock ="Left"
            Orientation="Vertical" Background="#ECE9D8">
  <TextBlock Margin="3" Text="Controls"/>
  <Button Margin="3,5" HorizontalAlignment="Left"
  Click="b1Click">Subdivide </Button>
  <Button Margin="3,5" HorizontalAlignment="Left"
  Click="b2Click">Clear</Button>
</StackPanel>
```

Now we modify the C# code in `Window1.xaml.cs`. We start by initializing both polygons to be empty:

```
public partial class Window1 : Window
{
    Polygon myPolygon = new Polygon();
    Polygon mySubdivPolygon = new Polygon();
    bool isSubdivided = false;
    GraphPaper gp = null;
    [...]
    public Window1()
    {
        [...]
        initPoly(myPolygon, Brushes.Black);
        initPoly(mySubdivPolygon, Brushes.Firebrick);
        gp.Children.Add(myPolygon);
        gp.Children.Add(mySubdivPolygon);

        ready = true; // Now we're ready to have sliders
                      // and buttons influence the display.
    }
```

The polygon initialization procedure also sets the polygons to have different colors and a standard line thickness, and to make sure that when two edges meet at a sharp angle they are truncated, as shown at the bottom of Figure 4.4, instead of having the joint extended in a long miter as shown in the middle.

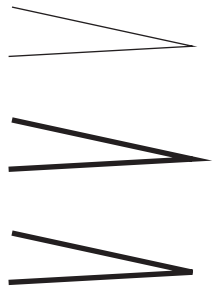


Figure 4.4: When we thicken the vertex join at the top, we must miter it, as shown in the middle. At the bottom, the miter is limited.

```
private void initPoly(Polygon p, SolidColorBrush b)
{
    p.Stroke = b;
    p.StrokeThickness = 0.5;// 0.5 mm thick line
    p.StrokeMiterLimit = 1; // no long pointy bits
    p.Fill = null;          // at vertices
}
```

Handling a click on the Clear button is straightforward: We simply remove all points from each polygon and set the `isSubdivided` flag back to `false`:

```
// Clear button
public void b2Click(object sender, RoutedEventArgs e)
{
    myPolygon.Points.Clear();
    mySubdivPolygon.Points.Clear();
    isSubdivided = false;

    e.Handled = true; // don't propagate click further
}
```

The Subdivide button is more complex. First, if the polygon is already subdivided, we want to replace `myPolygon`'s points with those of the subdivided polygon. Then we can subdivide `myPolygon` and put the result into `mySubdivPolygon`.

Subdivision amounts to determining, for each vertex, the previous and next vertices and then combining coordinates in a two-thirds-to-one-third fashion to find the location of the corner-cutting points. (This combination is closely analogous to the idea of averaging the coordinates of two points to find the coordinates of the midpoint of the segment between them, which you'll recall from elementary geometry.)

```
  // Subdivide button
  public void b1Click(object sender, RoutedEventArgs e)
  {
      Debug.Print("Subdivide button clicked!\n");
      if (isSubdivided)
      {
          myPolygon.Points = mySubdivPolygon.Points;
          mySubdivPolygon.Points = new PointCollection();
      }

      int n = myPolygon.Points.Count;
      if (n > 0)
      {
          isSubdivided = true;
      }
      for (int i = 0; i < n; i++)
      {
           int nexti = (i + 1) % n; // index of next point.
           int lasti = (i + (n - 1)) % n ; // previous point
           double x = (1.0f/3.0f) * myPolygon.Points[lasti].X
                    +(2.0f/3.0f) * myPolygon.Points[i].X;
          double y = (1.0f/3.0f) * myPolygon.Points[lasti].Y
                    +(2.0f/3.0f) * myPolygon.Points[i].Y;
          mySubdivPolygon.Points.Add(new Point(x, y));

          x = (1.0f/3.0f) * myPolygon.Points[nexti].X
             +(2.0f/3.0f) * myPolygon.Points[i].X;
          y = (1.0f/3.0f) * myPolygon.Points[nexti].Y
             +(2.0f/3.0f) * myPolygon.Points[i].Y;
          mySubdivPolygon.Points.Add(new Point(x, y));
      }
      e.Handled = true; // don't propagate click further
  }
```

Finally, we must handle mouse clicks. Anytime the user presses the mouse button, we want to add a new vertex to the polygon *unless it's already subdivided*. We therefore check the `isSubdivided` flag, and if it's `false`, we add the point to our `Polygon`.

```
    public void MouseButtonDownA(object sender,
                                 RoutedEventArgs e)
    {
        if (sender != this) return;
        System.Windows.Input.MouseButtonEventArgs ee =
          (System.Windows.Input.MouseButtonEventArgs)e;
        if (!isSubdivided)
        {
            myPolygon.Points.Add(ee.GetPosition(gp));
        }
        e.Handled = true;
    }
  }
}
```

That's it! You can run the program to see how well it works. When you click only two points, your polygon looks like a line segment. When you then subdivide, the line segment appears shorter. Explain to yourself why subdividing again doesn't make it shorter still.

A similar process can be used to take a 3D polyhedron and "cut corners" in an attempt to smooth it out. Will it always get smoother? We'll discuss this further in Chapter 23. In the meantime, having seen how rapidly subdivision seems to smooth out curves, you can ask yourself, "How could I analyze whether the curve approaches a limit, and whether such a limit is smooth at a particular point?" This is addressed (for a somewhat different subdivision method, but the principles are similar) in Chapter 22.

4.7 Discussion

We've given you the tools to create simple applications in WPF for experimenting with ideas from graphics, and worked through a subdivision application as an example of using those tools. The exercises in this chapter will give you the opportunity to explore the power of the 2D test bed, and to discover some interesting ideas in graphics that you'll encounter again later in this book. We strongly advise you to complete at least a couple of these exercises so that working with this framework will be simple for you in the future.

This test bed was built for a reason. Lots of years of working in computer graphics have taught us another principle:

> ✓ **THE FIRST PIXEL PRINCIPLE:** The first pixel is the hardest.

When you write a new program in graphics, the most common first result is a completely black screen. That's almost impossible to debug, because any of a million things could be causing the problem. Usually when you manage to get *anything* to show up on the screen, you're finished with a great deal of your debugging. It's therefore proven useful to start with a program that does something like what you're trying to do, and to gradually modify it until it *does* do what you want, but make the modifications so that at every stage you can tell, by looking at it, that it's doing what you expected. The test bed provides you with a starting point for a whole host of possible programs. We hope it will save you countless hours of debugging.

4.8 Exercises

Exercise 4.1: Modify the corner-cutting program to place the cutoff points one-fourth and three-fourths of the way along the line; describe the results.

Exercise 4.2: Modify the corner-cutting program to be a "dualizing" program: It replaces a polygon with its **dual**, a polygon whose vertices are the midpoints of the original polygon, connected in the same order, so that the dual of a square is a diamond. Experiment: Does repeated dualizing tend to remove crossings of self-intersecting polygons? Can you find a polygon whose successive duals are *always* self-intersecting?

Exercise 4.3: Modify the test bed to display only a single image (e.g., the water lilies) until a button is clicked; each time the button is clicked, a new image should be displayed (perhaps cycling through a collection of four or five images). To make a new image appear, you'll need to update the `BitmapSource`. This exercise is somewhat harder than the previous ones, because we've given you less guidance.

Exercise 4.4: Read about motion-induced blindness at the end of Chapter 5, and then write a program that lets you experiment with it, including changing the grid spacing and colors, changing the color and size of the "disappearing" dots, and altering the speed with which the grid rotates. Try to determine the settings that induce the "blindness" in you most effectively.

This page intentionally left blank

Chapter 5

An Introduction to Human Visual Perception

5.1 Introduction

The human eye dominates the field of computer graphics, for without it, graphics would be almost useless. Everyone working in graphics should understand something about how the visual system works. Until some day when graphics is "perfect" and somehow indistinguishable from reality, we must try to best use our computational and display resources to convince the visual system that it's perceiving reality, which entails omitting work that produces undetectable (or barely detectable) differences from the ideal.

This chapter introduces some important basic ideas, as well as outlining the limitations of our current understanding. The science of human vision and the related science of machine vision are lively areas of research precisely because of the richness of these limitations. Of course, a great deal *is* known, and we'll summarize some of it here.

The visual system's remarkable parallel processing powers allow enormous amounts of information to be transferred from the computer to the user. (The limitations on bandwidth in the other direction—human to computer—are a source of frustration and opportunity for clever design; see Chapter 21.) The visual system is both tolerant of bad data (which is why the visual system can make sense of a child's stick-figure drawing, or an image rendered with a very crude lighting model), and at the same time remarkably sensitive. Indeed, the eye is so sensitive to certain kinds of error that debugging graphics programs entails special challenges: A single tiny error (one red pixel in a 1-million-pixel grayscale image of a lighted sphere) stands out, while a one-in-a-million error in many other computations might never be noticed. There's a converse to this as well, which we mentioned earlier: We can use imagery to convey an enormous amount about what a program is doing, so good graphics programmers use visual displays to help them understand and debug their code.

✓ **THE VISUAL DEBUGGING PRINCIPLE:** Use visual displays to help you debug and understand your graphics programs.

In graphics, output from the computer to the user is typically in the form of light emitted by a display toward the user's eyes. The display might be a conventional flat-panel display, a projector, a head-mounted display, or a heads-up display for an aircraft pilot or automobile driver. In all cases, the light reaches us through the eye. The eye's responses to that light are processed by the visual system.

There are *other* modes of interaction as well, of course: Haptics (touch) and sound are often used as part of the computer-to-human communication channel. But the great bulk of the communication is through the visual system, which is why we concentrate on it. The visual system is powerful in part because light, which carries information to the visual system, has some special properties that are not shared by sound, touch, smell, or taste. For instance, light isn't **directionally diffuse:** A beam of light that starts in some direction travels in that direction only; it can travel without a supporting medium, and when traveling through air (the most common medium) it's largely uninfluenced by the air (although variations in the air's index of refraction as a function of density can distort light—think of seeing the desert "ripple" on a hot day). By contrast, the chemicals responsible for smell and taste not only diffuse, but also are advected by moving air, and sound's direction of propagation can be substantially altered by wind shear. Light is remarkably good at carrying information from a source to our eyes. Touch, by contrast, only works when the sensor (e.g., your finger) is collocated with the thing being observed.

It's tempting to try to reduce the visual system's response to stimuli in various ways that will make it easier to formulate a model of it. For example, because the first step in our processing of light is detection by the sensory elements of the eye, it's tempting to say, "The response of the visual system depends *only* on the incoming light; if you apply the same pattern of light, you get the same response." That's wrong, however, at both the physical and mental levels. At the physical level, seeing a sunny beach after walking out of a dark restaurant, for instance, causes you to squint your eyes reflexively, while seeing that same beach after having been outdoors for a few minutes causes no such physical, physiological, or psychological reaction. At the mental level, it's been shown that if you've recently been shown an object, you'll notice another object like it in a jumble of others more quickly. So, any model of visual processing must depend not only on the *current* stimuli, but on the recent past as well. More significantly, our pattern recognition ability is also influenced by training and learning. Once you've learned to identify a shape, you will recognize it much faster the next time you encounter it; a good example is the reading of the characters or glyphs that make up text. Almost every aspect of the visual system is similarly complicated; there seem to be no easy explanations. On the other hand, there is a wealth of experimental evidence that helps us understand some of what the visual system is doing [Roc95]. In this chapter, we focus on the visual system and how it perceives the world, but the discussion is necessarily abbreviated; we limit the discussion to the aspects of the system that are likely to have an impact within graphics systems. The chapter

does however, conclude with a few brief remarks on the relationship between the visual system and other perceptual modes, like hearing and touch.

Each section concludes with a paragraph or two labeled "Applications," in which ideas from the section are related to applications in graphics.

5.2 The Visual System

The human visual system (see Figure 5.1) consists of the eye (which focuses light and contains sensors that respond to incoming light), the optic nerve, and parts of the brain collectively called the **visual cortex.** The exact functioning of the parts of the visual cortex is not completely known, but it is known that some "early vision" parts (i.e., those that handle the first few steps in the processing of the visual signal) detect sharp contrasts in brightness, small changes in orientation and color, and **spatial frequencies,** that is, the number of alternations between light and dark per centimeter. We could summarize this by saying that we are adept at detecting and noting changes in what we might loosely call "patterns." The detection of orientation or color or frequency changes is **local,** that is, we are sensitive to adjacent things having different colors, but small color differences between things that are far apart in our visual field are not detected by the early-vision system. Also in the early-vision system are parts that assemble the local information into slightly larger-scale information ("This little bit of edge here and this little bit next to it constitute a larger piece of boundary between two regions").

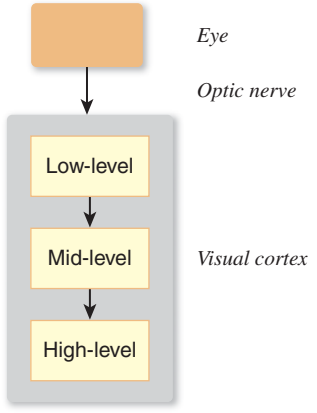


Figure 5.1: The components of the visual system.

Later regions of the visual cortex seem to be responsible for detecting motion, objects ("This is the thing in the foreground; all that other stuff is in the background"), and shapes, handling "attention," and providing control of the eye (i.e., muscle control to help the eye track an object of interest).

The simplicity suggested by Figure 5.1 is misleading: While there is certainly a "pipeline" structure to the visual system at a large scale, a wealth of parallel processing goes on as well, along with substantial feedback from later levels to earlier ones.

The visual system performs many tasks extremely well, such as determining size and orientation regardless of your viewpoint or distance, recognizing color invariantly under a variety of lighting conditions, and recognizing shapes, even in the presence of noise and distortions. It performs other tasks poorly, such as determining absolute brightness, recognizing parallel lines, and detecting identical but nonadjacent colors. And some of these strengths and weaknesses seem almost contradictory: We're great at noticing a tiny thing that's different from its surroundings (e.g., a black pebble on white sand), but we're also great at *ignoring* many things that are different from their surroundings, which lets us watch old films with lots of film grain and scratches and other noise and not be distracted. It's natural to explain the visual system's particular "talents" on an evolutionary basis, often based on the ideas that the visual system helps us (a) find food, and (b) avoid predators.[1] Thus, for example, humans are very sensitive to motion (which would help one detect predators that are trying to camouflage themselves), but we're not particularly good at remembering colors from one day to the next. The visual system is also very good at detecting color similarity under different lighting conditions (you want to be able to recognize food both at noon and at dusk, and

1. Mating and obstacle avoidance may also be influences.

you want to recognize that something is *all* banana, even if part of it is in sunlight and part of it is in shade). It's reasonably good at determining depth, especially for nearby objects—which is very useful for coordinating your hand movement as you reach out to pick a berry or fruit. Indeed, hand–eye coordination, especially in the movements of an athlete or craftsperson, is a marvel of multiple systems working together efficiently while completely bypassing the conscious aspects of the cognitive system.

The details of color perception are discussed extensively in Chapter 28; we'll touch on them only briefly in this chapter. Similarly, the perception of motion is discussed in Section 35.3.2.

It's rather tempting to believe that we know how we see. We say things like, "Well, it's *obvious* that I look for things with similar color, like the leaves of a tree, and try to group them together into a coherent whole so that I perceive the leaves and the trunk-and-branches as separate groups." But what's obvious is not necessarily true; a few moments spent examining various so-called "optical illusions" demonstrates this immediately [Bac].

The visual system's functioning matters in immediate ways in computer graphics. In graphics, we often want to ask, "Is the image I have rendered perceptually different from the ideal image, or is it close enough to generate the same percept in the viewer's mind, in which case I need not compute anything further?" In other words, the ultimate measure of rendering-and-display success is perceptual. There's an easy way to measure the similarity between two images (take the pixel-by-pixel difference of the image, square all the resultant numbers, sum them, and take the square root; this is called the **sum-squared difference**, **L^2 difference,** or **L^2 distance**[2]), but this measure of difference does not always match actual differences in perception. Figure 5.2 shows a grayscale instance of this: The L^2 distances from the top 41×41-pixel image (in which all pixel values are 118 in a range of 0 [black] to 255 [white]) to the middle (all pixels 128) is the same as the distance to the bottom (all pixels 118, except the middle, which has a value of 255), even though the bottom one looks much more different.

There's been substantial work in trying to develop a "distance function" that tells how far apart two images are, perceptually speaking [LCW03], but much remains to be done. In the meantime, there are some useful rules to guide design choices. The logarithmic sensitivity of the visual system described later means that our visual system is more sensitive to radiance[3] errors (of a fixed magnitude) in dark areas than in light ones. The local adaptability of the visual system means that *changes* in intensity tend to matter more than absolute intensity (as suggested by Figure 5.2); if you have a choice, you should aim to get the gradients (i.e., the local *changes* in the intensity) right rather than the values.

We now know the following about the visual system: first, that our perception of things is fairly independent of lighting (e.g., when you see an object lit by bright sunlight or by the remaining light at dusk, you still identify it as "the doorknob to my home"), and second, that the early portions of the visual system tend to detect edges (i.e., boundaries between regions of different brightness) and assemble them into something that the brain perceives as a whole. From these, it seems reasonable to say that images are similar if the pixel-by-pixel ratio of

Figure 5.2: Three 41×41*-pixel images. The top image has all pixel values 118; the middle has all pixel values 128; the bottom has all pixel values 128, except the center, which is 255. The* L^2 *distances from the top image to each of the others are approximately equal, but this does not match our own understanding of "sameness."*

2. Closely related is the notion of "root mean square" or RMS difference, which is the L^2 difference *per pixel*.
3. Radiance is a physical unit for measuring light, described in detail in Chapter 26.

brightness is locally fairly constant, and if the set of "edges" in each image are in the same locations. The notion of "locally" depends on how the image is viewed: If each pixel subtends 1° at the eye, "locally" may mean "over a region a few pixels wide," while if each pixel subtends $0.01°$, "locally" may mean "several hundred pixels." Indeed, it's possible to make images that appear similar at one distance but distinct at another distance. A simple example is a black-and-white checkerboard pattern and a gray rectangle: Close up, they're quite distinct; at a large distance (great enough that the visual system cannot distinguish the individual checkerboard squares) they appear identical. More complex examples (see Figure 5.3) are described by Oliva [OTS06].

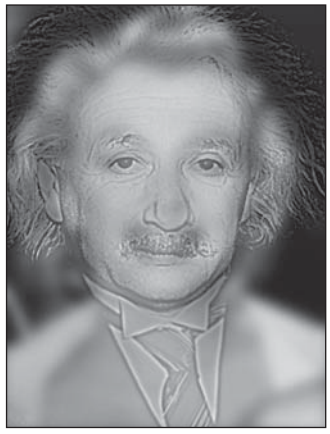


Figure 5.3: Close up, you see Einstein; from a distance, you see Marilyn Monroe. (Image courtesy of Aude Oliva, MIT.)

Applications. How much does all this matter for graphics? Since much of graphics is used to make people say that they are seeing some particular thing when they view their computer's display, it's quite important. On the other hand, our understanding of the visual system is still relatively sparse, so adapting our imagery to affect perception at the retinal level may be relatively easy, while trying to adjust it to affect the way in which whole objects are perceived may be more challenging and more prone to unexpected results. Furthermore, there's an interaction between **low-level vision** (the parts of our visual system responsible for detecting things like rapid changes in brightness in a particular area, typically the early-vision parts) and **high-level vision** (the parts responsible for forming hypotheses like "I'm seeing a surface with a pattern on it") that is still not well understood. Mumford, in an essay on pattern theory [Mum02], cites a remarkable analogous example from the auditory system: Psychologists recorded various sentences—"The heel is on the shoe," "The wheel is on the car," "The peel is on the orange"—and replaced the first phoneme of the second word in each sentence with noise, resulting in "The #eel is on the shoe," for example, where the hash mark denotes noise. Subjects who listened to these sentences perceived not the noise-replaced sentences, but the originals, and indeed, did not notice a phoneme was missing. Thus, as Mumford notes, the actual auditory signal did not reach consciousness. On the other hand, the replacement phoneme could only be determined from the larger context of the sentence. Mumford conjectures that vision may, in many cases, work the same way: While low-level information is often extracted from what you see, in some cases the way in which it's treated may be influenced by the results of higher-level understandings that you get from partially assembling the low-level information. For example, when you see someone leaning against a railing, you form the hypothesis that the railing continues behind the person, without ever consciously considering it. When you see something through the spinning blades of a fan, you assemble the parts you see at different times into a coherent whole, rather than assuming that the obscured parts at one instant are unrelated to the unobscured parts in the same portion of your visual field a moment later. Because of these interactions between high-level and low-level vision, we'll concentrate primarily on the low-level aspects, which are better understood.

Do we really "see" things? It's more accurate to say that our visual system contructs a model of the world from its input, forming this model with a combination of perceptual and cognitive processes that resolve apparent contradictions in the perceptual data (as in the experiment Mumford describes). This lets the brain eventually form an object hypthesis ("I see *this* thing!"), albeit with some backtracking if your cognitive abilities contradict what you think you saw ("That *can't* be a flying elephant!"). Thus, the end result of vision is a construction created by the mind, and not objective reality.

5.3 The Eye

Despite our limited understanding of the visual system, there are physical characteristics of the eye that limit what it can possibly do, and these can help govern the design of graphics systems. For instance, there is a smallest detectable brightness difference, and a smallest angular resolution for the eye. A display whose pixels could show brightness differences smaller than those, or whose pixels subtended an angle only $1/10$ as large as this smallest angular resolution, would be unnecessarily complex. We'll regard the eye as stopping at the optic nerve, with the nerve and the visual cortex constituting the remainder of the visual system.

5.3.1 Gross Physiology of the Eye

At a large scale, the eye consists of a globe-shaped object, held in place by the skull, various muscles that are attached to it, and other soft tissue surrounding it (see Figure 5.4).

The control of the rotation of a pair of eyes is coordinated by our visual system so that the received pattern of light on the retinas of the eyes can be integrated to form a single coherent view of the world; the left-eye and right-eye views of a scene are generally different, and the *disparity* between these views helps us estimate the depth of objects in the world. (You can experiment with this easily: Mark a spot on the wall of a room, and place several objects more or less between you and this mark, at different distances. While staring at the mark, cover first one eye and then the other, and notice how the left–right positions of objects near you seem to move as you switch eyes.)

At a coarse level, the path from an object that's either emitting light (a light bulb) or reflecting it (a book on your desk) to your retina—through the pupil and lens and vitreous humor (the gel-like liquid in the eyeball)—can be modeled by a simple lens, mounted between the light-producing object and an imaging plane (see Figure 5.5). Light from an object is emitted along many rays, which hit different points of a lens and are refracted (bent) as they pass into and out of the lens,

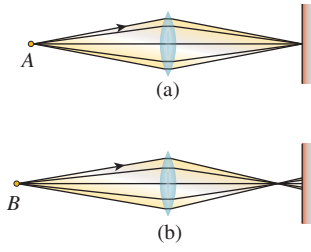


Figure 5.5: Light from point A is in focus when it arrives at the surface at right; point B is out of focus.

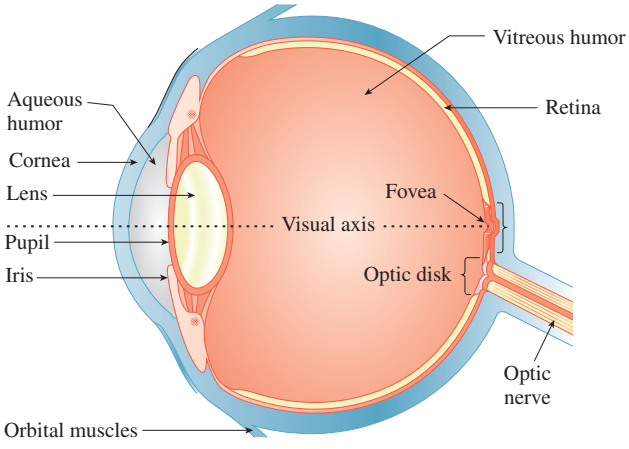


Figure 5.4: Light enters the eye through the pupil, then passes through the lens and vitreous humor, and arrives at the retina.

with the result (if the lens is properly shaped) that the rays all converge again at some point on the other side. If this point happens to lie on the imaging plane, we say that the object is "in focus." If the point of convergence is not on the imaging plane, then instead of producing a bright point of light, the rays generate a dim disk of light on the imaging plane. If the imaging plane is the sensor array for a digital camera, for instance, then the point B appears out of focus and blurry.

The convergence of all rays to a single point depends on the **index of refraction** (see Chapter 26)—a number that describes how much light bends as it passes from air to the lens and back to air—being independent of the wavelength of the light. For most materials, the index of refraction *does* vary slightly with wavelength; this can make objects of one color be in focus while those of another color are not, which accounts for the rainbow-colored fringe on the edges of objects when they're viewed through a magnifying glass, for instance.

Because the eyes can slightly modify their lenses' shape, the visual system can use focus/defocus to detect distance from the eye to an object, at least for nearby objects (defocus becomes less severe the farther away objects are). The amount of defocus-from-depth depends on the lens diameter. For very small diameters, there is a much larger depth range that's almost in focus (this range is described in photography as **depth of field**); for large diameters, the depth of field tends to be small. For an idealized pinhole camera, in which light passes through an infinitesimal hole on its way to the image plane, depth of field is infinite; unfortunately, the light-gathering ability of such an idealized device is zero. The human eye also has an adjustable pupil. In low light, the pupil opens wide and gathers more light, but at the cost of reduced depth of field; in bright light, the pupil closes, enhancing depth of field. Contrary to common wisdom, this pupil adjustment is hardly significant in the matter of adapting to a wide range of brightness levels—the pupil's area changes by a factor of, at most, ten, while the largest arriving radiance in ordinary experience is about ten orders of magnitude larger than the smallest, but the response *is* fast, making the pupil very effective at short-term adjustments. The longer-term adjustment is a chemical process in the receptors.

5.3.2 Receptors in the Eye

A large portion of the inner back surface of the eye, the **retina,** is covered with cells that respond to the light that arrives at them. These are primarily in two groups: **rods** and **cones,** which we discuss further in Chapter 28. Rods are responsible for detecting light in low-light situations (e.g., night vision), while cones detect light in higher-light situations. There are three kinds of cones, each responsive to light of different wavelengths; the combination of the three responses generates the sensation of color (discussed further in Chapter 28). There are far more rods than cones (a ratio of about 20:1), and the distribution of rods and cones is not uniform: At the **fovea,** a region opposite the pupil, the cone cell density is especially high. Deering [Dee05] gives detailed descriptions of these distributions, and a computational model for the eye's response to light. There's another special area of the retina, the **optic disk,** where the optic nerve attaches to the eye. In this region, there are no rods or cones at all. Despite this, you do not have the sense, as you look around, that there is a "blind spot" in your perception of the world; this is an instance of higher-level processing masking out (or filling in) the details of low-level information. The blind spot is very much present, but if you were to notice it all the time, it would distract you constantly.

There is another set of recently discovered cells in our eyes that respond primarily to light in the blue region of the spectrum; their responses are not carried by the optic nerve and do not go to the visual cortex. Instead, they are used in controlling circadian rhythms in mammals.

The receptors in the eye detect light, provoking a response in the visual system; very roughly speaking, each doubling of the arriving light at a receptor generates the same *increment* of response. If light *B* appears half as bright to you as a geometrically identical light *A,* then the energy emitted by *B* is about 18% that of light *A*. A light *C* whose energy is 18% of that from *B* will appear half as bright as *B*, etc. This logarithmic response helps us handle the wide range of illumination we encounter in everyday life. We discuss the perception of brightness of light further in Chapter 28. The logarithmic response of the visual system also determines something about *display* technology: An effective display must be able to show a wide range of intensities, and this range of intensities should not be divided into even steps, but rather into even *ratios* of intensity. This notion drives the idea of gamma correction discussed in Chapter 28. **Brightness** is the name used to describe the *perception* of light; by contrast, what we've been informally calling "intensity" of light is more precisely measured in units of radiance, described in detail in Chapter 26. What we've been saying is that, all other things being equal, brightness is roughly proportional to the log of radiance.

In general, it's useful to know that the eye *adapts* to its circumstances. When you're in your bedroom at night, reading, your eyes are adapted to the level of light in the room, an adaptation that's centered on the intensity[4] of the page you're looking at; when you turn off the light to go to sleep, everything in the room looks black, because the page's intensity is now well below the range of intensities to which your eye has adapted. But a few minutes later you can begin to distinguish things in your room that are illuminated by just moonlight, as your eye begins to adapt to the new, lower, light level. If you turn the light on again to resume reading, the page will initially seem very bright to you, until your eye has readapted.

The receptor cells in the eye do not act entirely independently. When the eye is generally adapted to ambient illumination, an extra bit of light arriving at one receptor will not only increase the sensation of brightness there, but also slightly reduce the sensitivity of the neighboring receptors, an effect known as **lateral inhibition.** The result of this (see Figure 5.6) is that the edge contrast between regions of light and dark is enhanced compared to the contrast between the centers of the regions: The dark side of the edge is perceived as being darker and the light side as being lighter. This is the origin of the Mach banding discussed in Section 1.7.

This has an important consequence for computer graphics. In early graphics systems, polygons were often "flat shaded." That is, relatively large areas of the screen were given constant colors. When a shape like a cylinder (approximated by an extruded polygon) was illuminated by light from one direction, adjacent facets were assigned differing constant shades depending on how directly they faced the light source. The eye, instead of blending together the slightly different adjacent

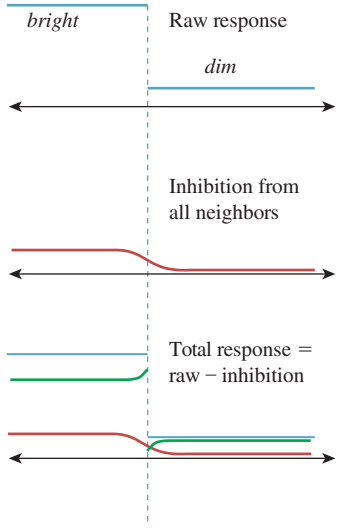


Figure 5.6: The raw response of receptors in the bright and dark regions (in blue, at top), the lateral inhibition amounts (in red, middle), and their difference—the actual response—shown in green at the bottom. Notice the enhanced contrast at the edge between light and dark, indicated by the dotted line.

4. We're using this term informally to describe the amount of light energy leaving the page and arriving at your eye.

shades, tended to enhance the differences at the edges, emphasizing the faceted structure.

Given this enhanced sensitivity to edges, it's natural to ask how small an edge the eye can detect. We can make a drawing of alternating parallel black and white stripes and move it away from the eye until it appears gray. This turns out to happen at a distance where two adjacent stripes subtend about 1.6 minutes of arc (a **minute** is $1/60$ of 1°).

The receptors in the eye adapt chemically to the overall brightness of what the eye is seeing. For many ordinary illumination levels, the eye can detect a ratio of intensities of about 100:1 within a small area. Figure 5.7 shows that this adaptation allows us to detect the brightness of arriving light only over a modest range for each level of adaptation. On the other hand, the eye can adapt very quickly to modest changes in illumination, so you can, for example, quickly search for a pencil in your dark backpack, even outdoors on a sunny day. Full adaptation to a major reduction in illumination, however, which involves chemical changes in the receptor cells, requires about a half-hour. After such adaptation, one can detect very low illumination levels; the ratio of the brightest distinguishable daytime levels to the dimmest distinguishable nighttime levels is more than 1,000,000:1. Many displays advertise contrast ratios of 10,000:1; since the eye can only discern ratios of about 100:1, why would such a range be important? Because the adaptation of the eye is partly *local:* As you stare from your unlit bedroom through a small window to the sunny outdoors, one part of your eye may be able to distinguish between things of different brightnesses in the room, while another distinguishes between things of different brightnesses outdoors. To generate this same percept, a display screen must be able to present comparable stimuli to the different regions of your eye. As an example of the extremes of perception, on a clear night you may be able to see a magnitude-3 star, while also seeing the moon clearly; the stellar magnitude for the moon is about -12.5. Since 5 stellar magnitudes represents a factor of 100 in intensity, this represents an intensity range of about 1 million. But if the moon is reasonably close (in your visual field) to that magnitude-3 star, you'll almost certainly be unable to see the star.

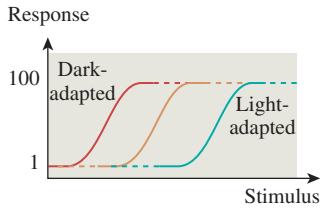


Figure 5.7: The dark-adapted eye's response to light "saturates" at a fairly low stimulus level; the light-adapted eye cannot detect differences between various low-light-level stimuli.

Applications. The visual system's ability to detect distance to an object through two different mechanisms—the eye can focus, or the two eyes together can use parallax, which we'll discuss presently—means that it's possible to have divergent distance detections when the eyes are fed different data. For instance, if a user wears a pair of glasses whose lenses are replaced by individual displays, we can fool the user into seeing "in 3D" by displaying different images on the two displays, making the user believe that the things seen are at various distances, creating a "stereo" effect. But to see these two distinct images at all, the user must focus on the displays, which are just a few inches from the user's eyes (or can be made to seem more distant with the use of lenses). The two percepts of depth contradict each other, and this makes many "3D display" experiences unpleasant for some users.

The adaptation of the eye to surrounding light levels, and the limited dynamic range within an adapted eye, means that we need not contruct displays with enormous contrast ratios between pixels, although it may be useful to be able to adjust the *mean* intensity over a large range. On the other hand, it also means that when we're displaying something very bright, like the sun shining through the leaves of a tree, we can eliminate most of the detail near the sun, since small variations in brightness of the leaves will be "masked" by the eye's local adaptation to the brightness of the sun.

The concentration of receptors near the center of the visual field means that we can afford to make peripheral displays less precise. Our sensitivity to motion in our peripheral field, however, means that we cannot be too sloppy.

The limitations of edge detection tell us how many lightness levels we need to be able to display to generate imagery that's apparently smooth.

5.4 Constancy and Its Influences

Somehow our visual systems go from received light to a perception of the world around us ("That's my car over there next to the red truck!"). The process is remarkably robust, in the sense that substantial changes in the input result in almost no change in the resultant percept: You can identify your car as being next to the red truck in bright sunlight, at dusk, or in late evening; you can identify it whether you're standing three feet away or 300 feet away (and when you're 300 feet away, you don't say, "Gosh, my car has shrunk!"); you can recognize it when you see it from the front or the right side or the left side or the back, without saying, "It's changed shape!"

On the other hand, the stimuli that provoke these constant percepts are very different: The light entering the eye from the car at night is very different from the light entering the eye from the car at midday. It's less intense, and probably has many more short-wavelength components (which humans tend to see as blue), at least if the streetlights use mercury-vapor lamps. Different cells are responding to the light (the rods are in the range of light at which they begin to discriminate illumination levels). So the visual cortex must do some interesting things to generate the same general percept. Of course, the percept is not entirely the same: You know you're seeing the car at night rather than during the day, but you *don't* believe, because of the different illumination, that the car's color has changed. This is an instance of **color constancy.** Similarly, you don't believe, when you look at it from a different location, that the car's shape or size has changed; these are examples of **shape constancy** and **size constancy.**

Constancy is a wonderful thing (in terms of preventing perpetual confusion). On the other hand, it's also responsible for making our visual systems rather bad at some things at which other visual systems (e.g., digital cameras) are good. As mentioned, we're not very reliable at determining when two colors are the same, unless they're adjacent. But a digital camera can do so quite reliably. One consequence of this is that, as we work in computer graphics, it's important to know what "visual system" will be processing the images we produce: If it's a human eye, then small color errors in patches that are far from one another may not matter; if, however, we're using computer-graphics-produced images to test a computer-vision system whose input comes from digital cameras, then such errors may be significant.

It's often helpful, in understanding a system, to know of instances where it fails (e.g., we use such instances in debugging). In the case of the visual system, "failure" may not be well defined, but we certainly have examples where our visual system does not do what we expect it to do. For instance, you can see how bad humans are at determining the absolute lightness of a region by noting your sensation of lightness when that region is surrounded by other regions of varying lightness, as in Figure 5.8.

This might seem like a failure of constancy—after all, the center squares in Figure 5.8 are "all the same." But if we model the center square and its surrounding

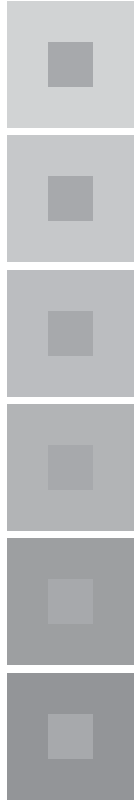


Figure 5.8: All the center squares have the same lightness; the apparent lightness, however, is profoundly influenced by the surrounding squares.

square as being painted on a surface and illuminated by lights of varying intensity, we get a very different set of images, as in Figure 5.9, in which the center square's gray values are all different, but you have the perception that the center squares are all fairly comparably dark. *This* is an instance of lightness constancy under varying illumination. (An even more spectacular example of lightness constancy—and its *non*relation to incoming intensity—is shown in Figure 28.15.)

The materials available on this book's website discuss further constancy effects.

Applications. The various constancy illusions show that surrounding brightness can affect our perception of the brightness of a surface or of a light. This leads to the use of different gamma values (discussed in Section 28.12) for studio monitors, theatre projection, and ordinary office or home displays, where the average brightness of the surroundings affects the appearance of displayed items. It also suggests that during rendering, if you want to visually compare two renderings, you should surround each with an identical neutral-gray "frame" to help avoid any context-based bias in your comparison.

The other consequence of constancy, at least for brightness, is that relative brightnesses matter more than absolute ones (which helps explain why edge detection is so important in early vision). This suggests that if we want to compare two images, it may be the *ratio* of corresponding pixels that matters more than the difference.

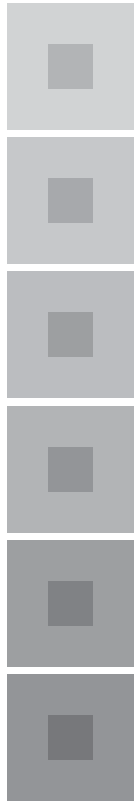


Figure 5.9: The ratio of the center square's darkness to the surrounding square's darkness is approximately the same in each example; you tend to see the center squares as exhibiting far less variation in lightness than those in the previous figure.

5.5 Continuation

When one object seems to disappear behind another, and then reappear on the other side (see Figure 5.10), your visual system tends to associate the two parts as belonging to a whole rather than as separate things; this is an instance of the idea from Gestalt psychology that the brain tends to perceive things as a whole, rather than just as individual parts.

◈ One proposed partial mechanism for this perception is the C^1 random walk theory [**?**, Wil94] in which we suppose that at T-junctions (where an outline of one object appears to pass behind another object), the brain "continues" the line in the same general direction it was going when it disappeared, but with some random variation in direction. Some such continuations happen to terminate at the *other* T-junction, headed in the appropriate direction. If we consider all such connections between the two, some are more probable than others (depending on the probability model for variation in direction, and on the lengths of continuations). Each point in the obscured area occurs on some fraction of all such continuations (i.e., there's a probability density p with the property that the probability that a random connection passes through the area A is the integral of p over A). The ridge lines of the distribution p turn out to constitute very plausible estimates of the "inferred" connection between the T-junctions, with the integral of p over the curve providing a measure of "likelihood" that the lines are connected at all. If the T-junctions are offset from each other (i.e., if the two segments are not part of a single line), the probability decreases; if the two segments are nonparallel, the probability decreases; only when the T-junctions are perfectly aligned is the probability of connection at its maximum. Is such a "diffusion of probability of connection" really taking place in the brain? That's not known. But this notion that the ridge lines of p form the most likely connections cannot, as stated, tell the

whole story, because there's a peculiar effect in the matching of diagonal lines, which we now discuss.

When you see a diagonal line pass behind a vertical strip, as in Figure 5.11, you tend to fail to correctly perceive when the diagonal parts are aligned (the explanation seems to involve misperception of acute angles). Nonetheless, the effect can be drastically reduced by placing ends on the vertical strip to make it a parallelogram (or to give other cues, like a texture that appears to have perspective foreshortening) so that the diagonal line appears to lie in a plane parallel to that of the strip (see Figure 5.12).

Applications. Such peculiarities of the visual system have an important impact when we examine nonphotorealistic or expressive rendering, in which we seek to create imagery whose goal is not faithfulness to reality, but rather the expression of the creator's intent, which may be to draw the eye to a particular portion of the image through judicious choices of what to show. Consider, for example, an illustration in an automobile repair manual, where the area being discussed is drawn in detail, and surrounding regions are simplified to just a few lines to avoid confusion. When we simplify our imagery by eliminating detail, are we also losing important cues that the visual system uses to understand the presented scene? In some cases, it's clear that we *do* lose important features; a failure to draw shadows can cause a viewer to misunderstand which objects are touching others, for instance. But even in an example like that of Figure 5.12, suppose that our abstraction removes the "texture" on the vertical strip in (b). The diagonal line then will appear mismatched, as in Figure 5.11(b).

Continuation can also be used to infer meaning from a user's sketch of a shape [KH06]: When one contour is obscured by another, we can use a model of continuation to infer where the user thinks it goes.

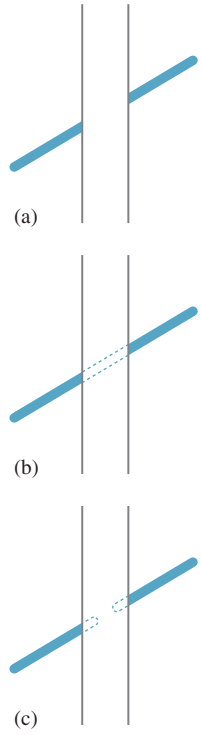


Figure 5.10: The diagonal line in (a) seems to pass behind the vertical strip. You strongly sense the two diagonal segments are part of a continuous whole, as shown in (b), rather than each terminating behind the vertical strip, as in (c).

5.6 Shadows

Shadows provide remarkably powerful cues to our visual system, but these cues are not always exactly what we think they are. For instance, shadows help us estimate the depth (distance from the viewer) of objects that are not on the ground plane. Keren et al. [LKMK97] demonstrated this compellingly with an example like the one shown in Figure 5.13, in which the motion of a ball in a scene is very strongly disambiguated by means of shadow cues: With no shadow cues, it's easy to convince yourself that the ball is either moving in a plane of constant distance from the eye, rising as it moves right, *or* moving at constant height from a point above the front-left corner of the tray to a point above the rear-right corner. When shadows are included, one choice or the other is forced on the perceptual system. It's interesting to experiment with this example, because it turns out that the effect is almost equally strong when the shadow does *not* correspond to the shape of the object—a small square instead of a disklike shadow, for instance. Furthermore, the shadow cue can easily overwhelm other visual cues like the foreshortening due to perspective (in the front-to-back motion, the sphere will subtend a smaller visual angle when it's far away than when it's close, so a constant-size sphere should appear to be always moving in the constant-distance plane; nonetheless, with a shadow cue, you see it moving along the front-left/rear-right diagonal).

From this, we might infer that shadows provide some kind of depth or position information, but are less informative about shape. But shadows where an

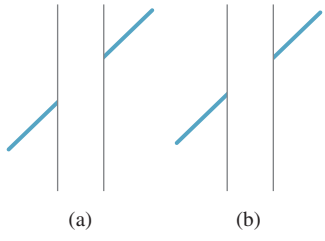


Figure 5.11: Which of (a) and (b) seems to be a single continuous straight line passing behind a strip, and which looks like the two segments are parallel but not part of the same line? Place a straightedge on the figure to determine the truth.

object meets a surface actually *do* convey something about shape, as shown in Figure 5.14. Such shadows are also very strong cues in helping us determine object contact; a drawing without contact shadows can lead us to see objects as "floating above" a surface rather than resting on it.

Applications. Although a shadow may be quite faint, and hence not terribly important in the L^2 difference between two images, the perceptual difference between the images can be huge. Rendering shadows is essential; getting them exactly right is not.

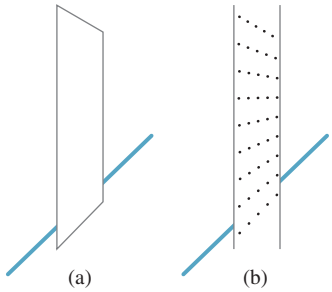


Figure 5.12: When we truncate the vertical lines so that the obscuring strip seems to be a plane parallel to one containing the line, the illusion from Figure 5.11 disappears; the same effect happens when the strip is given a texture that indicates this tilted orientation.

5.7 Discussion and Further Reading

Perception is a huge subject, of which this brief overview only touches on a few items of particular interest in graphics. There are physiological, mental, and philosophical aspects to the subject; there are also large unexplored areas. Static perception has been given a great deal of attention and study, but the effects of motion (not only how humans perceive motion, but also what effects motion has on our perception) are far less understood. Hoffman [Hof00] and Rock [Roc95] both provide fine overviews, but brain science is advancing at such a rate that you're probably best advised to look at recent journal articles rather than surveys in books to find out the best current thinking on the subject (which will surely change rapidly).

We've described constancy effects, but there are higher-level effects in vision as well. To some degree, what you see is highly dependent on what you're looking for. Simons and Chabris [SC99] showed that many viewers told to count how often a basketball is passed by some players fail to notice a person in a gorilla suit walking through the midst of them. Thus, semantic expectations regulate perception.

We haven't discussed stereo viewing in detail because it's rather specialized. In stereo the two eyes are presented with different images which the visual system must resolve. Typically, differences between the images result in the powerful percept of depth variation across the field of view. Unfortunately, as we mentioned earlier, the images presented to the eyes by a stereo-graphics system are typically displayed on flat surfaces that are not very distant from the eye; the adaptation of the eyes' lenses to focus on this display plane gives a depth signal that is at odds with the depths that the brain is inferring for the various things in the scene. This kind of contradictory evidence being presented to the visual system makes it quite difficult to know what a user will actually perceive. Furthermore, while stereo is a key cue in depth perception for most people, there are people who lack stereo

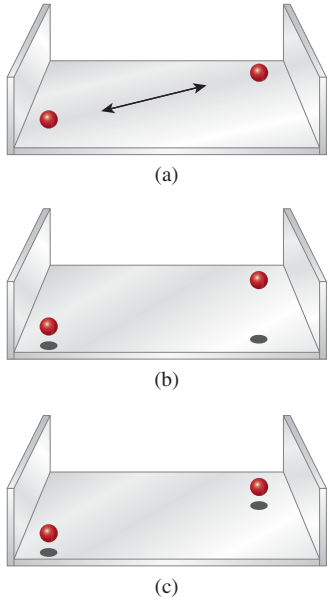


Figure 5.13: (a) A ball moves above a three-sided tray without shadows; its motion is not strongly determined. (b) and (c) Shadows force an interpretation of the motion as being in a vertical or horizontal plane.

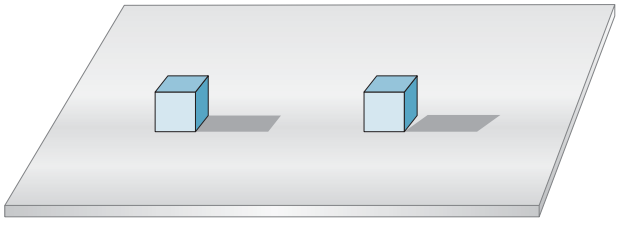


Figure 5.14: The appearance of a contact shadow tells us quite a lot about shapes and their relationship. You perceive the two identically drawn forms quite differently when shown their nonidentical shadows.

vision but have sufficient depth perception to perform tasks as complex as flying airplanes, thanks to their ability to read other cues such as perspective foreshortening, intensity modulation with distance, and especially motion parallax.

We've already mentioned that rendering is really a process of integration, and that the integration is typically done with randomized sampling. In building a renderer, we get to choose samples for the integrator. In regions of the image where there are more samples, we tend to get better estimates of the integrals we're computing. If that area is one which has little perceptual significance (e.g., if it's part of a salt-and-pepper texture), then the extra sampling effort is wasted; if it's perceptually salient (e.g., the edge of a dark shadow on a light surface), then the extra sampling is valuable. Greenberg et al. [Gre99] describe the inclusion of perceptual factors as a driving influence in rendering. One challenge is that to apply these ideas straightforwardly, one must model the perceptual process and then compare the ideal image (which may not be available) to the approximate one *in the post-perception state*. Ramasubramanian et al. [RPG99] developed an approach in which measurements made directly on images were closely related to perceptual measurements, allowing image formation to be more easily guided by estimated perceptual importance. Walter et al.[WPG02] applied a similar approach to reduce rendering effort substantially in textured regions where such a reduction would be imperceptible.

Perceptual difference measures are also used in image compression. JPEG image compression, for instance, attempts to approximate an input image in multiple ways, and then selects among these by choosing the one whose perceptual distance (in some measure) from the original is smallest. MPEG compression of moving image sequences operates similarly.

Differences that matter to the visual system when items are viewed in isolation may be ignored when they are viewed in a larger context, particularly one with lots of visual complexity. Recent work by Ramanarayanan et al. [RBF08] demonstrates that our perceptions of aggregates (a mixture of marbles and dice, or of two kinds of plants in a garden) have some surprising weaknesses. Related work [RFWB07] from the same group demonstrates that even though two pictures may be perceptually distinguishable, the distinction may not matter.

The field of perception is constantly advancing in new and surprising ways. As an example of the sort of stunning discovery that's being made even now, consider **motion-induced blindness,** in which certain objects can be made to disappear, depending on the motion of others. If the arrangement of crosses shown in Figure 5.15 is slowly rotated, and you stare at the tiny dot in the center, the three fixed surrounding dots can disappear completely from view. The effect is weak when the grid is the same color as the dots, but quite strong when they are different; a blue grid and yellow dots work well. The effect is also present over a wide range of rotation speeds and dot sizes.

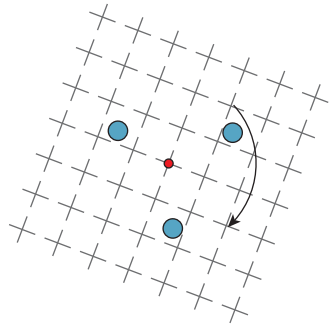


Figure 5.15: The grid of crosses is rotated about its center at a speed approaching ten seconds per complete rotation. The user is instructed to fixate on the center of rotation. After a moment, one or more of the other dots in the image seem to disappear.

As mentioned at the beginning of this chapter, visual perception is not the only mode of computer-to-human communication; sound and touch are also in frequent use. When both hearing and vision are used, and they contradict each other, which one dominates? Shams et al. [SKS02] report an interesting instance in which *sound* dominates; they "report a visual illusion which is induced by sound: when a single flash of light is accompanied by multiple auditory beeps, the single flash is perceived as multiple flashes." What about touch and vision? Randy Pausch [personal communication] reported that when a display shows an apparently dented surface, and a haptic device is used to "touch it" and is guided by

data that hides the dent, users report being able to feel the dent regardless. A careful study has been carried out by Burns et al. [BWR+05], supporting the idea of visual dominance. These are, however, isolated and controlled instances of multimodal sensation. The degree to which these different modalities interact in more complex situations is still unmeasured.

5.8 Exercises

Exercise 5.1: Write a program that displays an image consisting of parallel stripes, sitting above another image that's pure gray. Make the gray level adjustable (by slider, buttons, keystrokes, or any other means you like). Stand far enough away that the stripes are indistinguishable from one another, and adjust (or have a friend adjust) the gray level of the solid rectangle until you say it matches the apparent gray of the stripes. Now move toward the display screen until you can detect the stripes individually; measure your distance from the display, and compute the angle subtended at your eye by a pair of parallel stripes. You should make sure that you're not fooling yourself by having the display (after the press/click of a button) show either vertical or horizontal stripes next to the gray rectangle (at random) and have the position of the stripes and the solid rectangle exchanged or not (at random).

Exercise 5.2: Implement the motion-induced-blindness experiment; include buttons to increase/decrease the speed of rotation and the size of the "disappearing" dots, and allow the user to choose the color of the grid and the dots. Experiment with which colors work best at making the dots disappear.

Exercise 5.3: Write a program that draws three black dots of radius 0.25 at $x = 0, 1, 2$ along the x-axis. Then display instead three black dots at positions $t, t + 1$, and $t + 2$ (using $t = 0.25$ initially). Make the display toggle back and forth between the two sets of dots, once every quarter-second. Do you tend to see the dots as moving? What if you increase t to 0.5? Include a slider that lets you adjust t from 0 to 3. Does the illusion of the dots moving ever weaken? When $t = 1$, you *could* interpret the motion as "the outer dot jumps back and forth from the far left ($x = 0$) to the far right ($x = 3$) while the middle two dots remain fixed." Can you persuade yourself that this is what you're seeing? The strong impression that the dots are moving as a *group* is remarkably hard to abandon, supporting the Gestalt theory.

Exercise 5.4: Write a program to imitate Figure 5.13, where a slider controls the position of the red ball along its trajectory. Include a set of radio buttons that lets you change the "shadow" of the ball from an ellipse to a disk to a square to a small airplane shape, and see how the change affects your perception of the red ball's position. You can write the program using the 3D test bed or the 2D test bed—there's no particular need to get the perspective projection exactly right, so merely mimicking the figure will suffice.

This page intentionally left blank

Chapter 6

Introduction to Fixed-Function 3D Graphics and Hierarchical Modeling

6.1 Introduction

You've been introduced to how a 3D scene is projected to 2D to produce a rendered image, and you know the basic facts (substantially clarified in later chapters) about light, reflectance, sensors, and displays. The other required ingredient for an understanding of graphics is mathematics. We've found that students understand mathematics better when they encounter it experimentally (as we saw with the order-of-transformations issue in Chapter 2). But performing experiments using 3D graphics requires either that you build your own graphics system, for which the preliminary mathematics is critical, or that you use something premade. WPF is a good example of the latter, providing an easy-to-use foundation for 3D experimentation.

In this chapter, you learn how to use WPF's 3D features (which we'll refer to as **WPF 3D**) to specify a 3D scene, configure lighting of the scene, and use a camera to produce a rendered image. WPF's classic fixed-function model of light and reflectance is not based on physics directly, and does not produce images of the quality needed for entertainment products like animated films; however, because of the enormous adaptability of the human visual system, it does make pictures that our minds perceive as a 3D scene. The fixed-function model also has the advantage of being widely used in other graphics libraries; it's a model that researchers in graphics should know due to its extensive use in early graphics research and commercial practice, even though it's being rapidly superseded. The desire to produce more realistic pictures motivates the extensive discussions of light, materials, and reflectance found throughout the remainder of this book.

6.1.1 The Design of WPF 3D

There are dozens of commonly used 3D graphics platforms, covering a wide variety of design goals. Some are focused on image quality/realism regardless of cost (e.g., systems used to compute the frames for high-quality 3D animated films), while others target real-time interactivity with more-or-less realistic simulation of physical properties (e.g., systems used for creating 3D virtual-reality environments or video games), and yet others make compromises on image quality in order to achieve reasonably fast performance across a wide variety of hardware platforms.

As described in Chapter 2, WPF is a retained-mode (RM) platform—the application uses XAML and/or the WPF .NET API to specify and maintain a hierarchical **scene graph** stored in the platform. (You'll learn in Section 6.6.4 why it's called a "graph"; for now, just think of it as a scene database.) The platform, in conjunction with the GPU, automatically keeps the rendered image in sync with the scene graph. This kind of platform is significantly different from immediate-mode platforms such as OpenGL or Direct3D, which do not offer any editable scene retention. For a comparison of these two different architectures in the context of 3D platforms, see Chapter 16.

WPF's primary goal is to bring 3D into the domain of interactive user interfaces, and as such it was designed to meet these requirements:

- Support a large variety of hardware platforms
- Support dynamics for low-complexity scenes with a real-time level of performance on hardware that meets basic requirements
- And provide an approximation of illumination and reflection sufficiently efficient for real-time creation of visually acceptable 3D scenes

Here we use WPF 3D to introduce some 3D modeling and lighting techniques by example, taking advantage of WPF's easily editable scene descriptions to give you a hands-on understanding.

6.1.2 Approximating the Physics of the Interaction of Light with Objects

Each object in a 3D scene reflects a certain portion of incident light, based on the reflection characteristics of the object's material composition. Moreover, each point on the surface of an object receives light both *directly* from light sources (those that are not blocked by other objects) and *indirectly* by light reflected from other objects in the scene. The complex physics-based algorithms that directly model the intrinsically recursive nature of interobject reflection (described in Chapters 29 through 32) require lots of processing; if real-time performance is the goal, they often require more processing power than today's commodity hardware can provide. Thus, real-time computer graphics is currently dominated by approximation techniques that range from loosely physics-based to eye-fooling "tricks" that are not based on physical laws in any way.

The approximation techniques that generate the highest level of realism demand the most computation. Thus, interactive game applications (for which animating at a high number of frames per second is essential for success) must rely on the fast algorithms that compromise on realism. On the other hand, movie production applications have the luxury of being able to devote hours to computing a single frame of animation.

Many classic approximation algorithms were developed decades ago, when the power of computing and graphics hardware was a tiny fraction of what is available today, to meet two key goals: minimizing processing and storage requirements, and maximizing parallelism (especially in GPUs). These algorithms had their roots in software implementations that began in the late 1960s, grew in generality through the 1970s and 1980s, and were then implemented in increasingly more powerful commercial GPU hardware starting in the 1990s.

A particular sequence of the most successful of these algorithms, commonly called the **fixed-function 3D graphics pipeline,** has been in use for three decades and was dominant in GPU design until the late 1990s. This pipeline renders triangular meshes, approximating both polyhedral objects and curved surfaces, using simple surface lighting equations (for calculating reflected intensity at triangle vertices, as described in Sections 6.2.2 and 6.5) and shading rules (for estimating reflected intensity at interior points, as described in Sections 6.3.1 and 6.3.2). An application uses the fixed-function pipeline via software APIs of the type found in classic commodity 3D packages such as earlier versions of OpenGL and Direct3D. WPF is one of the newer APIs providing a fixed-function pipeline, and as we present its basic feature set throughout the rest of this chapter, we will provide a brief introduction to the classic approximation techniques, how well they "fool the eye," and what their limitations are.

Although the fixed-function pipeline is an excellent way to start experimenting with the use of 3D graphics platforms (thus our choice of WPF here), it is no longer de rigueur in modern graphics applications, for which the programmable pipeline (introduced in Sections 16.1.1 and 16.3) is now the workhorse. As GPU technology continues its rapid evolution, higher-quality approximations become more feasible in real time, and the ability to simulate the physics of light–object interaction in real time becomes ever more feasible.

6.1.3 High-Level Overview of WPF 3D

WPF's 3D support is closely integrated with the 2D feature set described in Chapter 2, and is accessed in the same way. XAML can be used to initialize scenes and implement simple animation, and procedural code can be used for interactivity and runtime dynamics. To include a 3D scene in a WPF application, you create an instance of `Viewport3D` (which acts as a rectangular canvas on which 3D scenes are displayed) and use a layout manager to integrate it into the rest of your application (e.g., alongside any panel of UI controls).

A `Viewport3D` is similar to a WPF `Canvas` in that it is blank until given a scene to display. To specify and render a scene, you must create and position a set of geometric objects, specify their appearance attributes, place and configure one or more lights, and place and configure a camera.

The 3D equivalent of the 2D abstract application coordinate system is the **world coordinate system,** with x-, y-, and z-axes in a right-handed orientation (as explained later in Figure 7.8). The unit of measurement is abstract; the application designer can choose to use a physical unit of measurement (such as millimeter, inch, etc.), or to assign no semantics to the coordinates. The scene's objects, the camera, and the lights are placed and oriented using world coordinates.

The use of physical units is optional, but can be helpful to accurately emulate some kind of physical reality (e.g., meters for modeling an actual neighborhood's houses and streets, or millimicrons for modeling molecules). The WPF platform itself is not informed of any semantics the application might attach to the units.

The scene is rendered to the display device via the pipeline represented at a very high level in Figure 6.1. The camera is positioned in the modeled world using world coordinates, and it is configured by specification of several parameters (e.g., field of view) that together describe a **view volume**—the pyramid-shaped object shown in the middle subfigure. (You'll learn a great deal about camera specification and view volumes in Chapter 13, and we'll examine the camera specification from the OpenGL perspective in Chapter 16.) The portion of the scene captured in the view volume is then projected to 2D, resulting in the rendering shown in the viewport, which will appear in the application's window.

As is the case in WPF 2D, the platform automatically keeps the rendering in sync with the modeled world. For example, making a change to the scene or to the camera's configuration automatically causes an update to the rendering in the viewport. Thus, animation is performed by editing the scene at runtime, performing actions such as the following:

- Adding or removing objects
- Changing the geometry of an object (e.g., editing its mesh specification)
- Transforming (e.g., scaling, rotating, or translating) objects, the camera, or geometric (in-scene) light sources
- Changing the properties of a material
- Changing the characteristics of the camera or lights

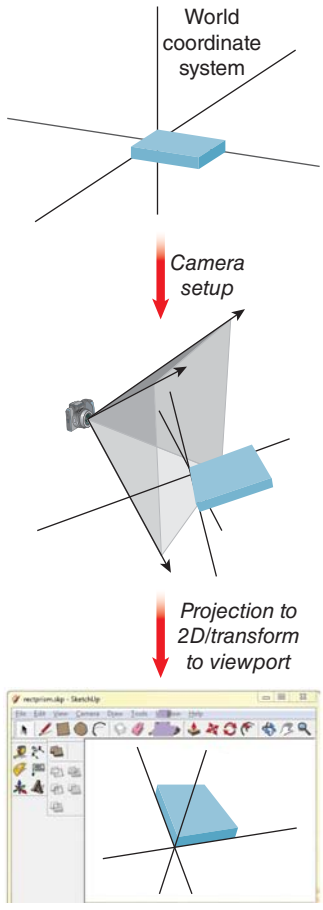


Figure 6.1: Very high-level overview of WPF's 3D geometry pipeline.

6.2 Introducing Mesh and Lighting Specification

In this section, we will use XAML to build a four-sided, solid-color pyramid, depicted atop a sandy desert floor in Figure 6.2 from the point of view of a low-flying helicopter. In this section, we focus on the construction and lighting of just the pyramid (ignoring the sky and desert floor).

6.2.1 Planning the Scene

Let's assume our desert floor is coplanar with the xz ground plane of the right-handed 3D coordinate system, as shown in Figure 6.3. To honor the great Mesoamerican Pyramid of the Sun (75 meters in height) near Mexico City, we'll give our pyramid that height, with a base of 100 m^2. As such, we'll choose meters as our unit of measurement. We will place our pyramid so that its base is on the xz ground plane, with its center located at the origin $(0, 0, 0)$, its four corners located at $(\pm 50, 0, \pm 50)$, and its apex located at $(0, 75, 0)$.

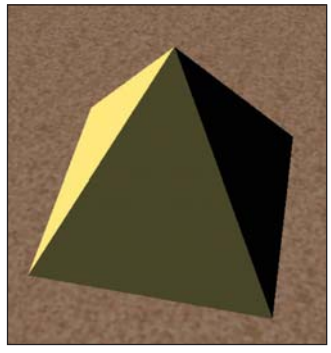


Figure 6.2: Overhead view of pyramid.

6.2.1.1 Preparing a Viewport for Content

To be visible, the viewport must live inside a WPF 2D structure such as a window or a canvas. For this example, we chose to use a WPF `Page` as the 2D container for the viewport, as it simplifies the use of interpreted development environments such as Kaxaml. Here we create a `Page` and populate it with a viewport of size 640×480 (measured in WPF canvas coordinates, as described in Chapter 2):

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
>
  <Page.Resources>
     Materials and meshes will be specified here.
  </Page.Resources>
  <Viewport3D Width="640" Height="480">
     The entire 3D scene, including camera, lights, model, will be specified here.
  </Viewport3D>
</Page>
```

Note: Here again, as in Chapter 2, some of XAML's "syntactic vinegar" will be obvious and may inspire questions. However, this chapter is not intended to be an XAML reference or to replace .NET's documentation; our focus is on semantics, not syntax.

The camera, the lights, and the scene's objects are specified inside the `Viewport3D` tag. The basic template of a viewport and its content looks like this:

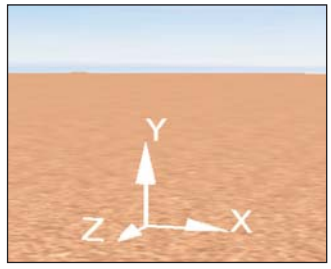


Figure 6.3: WPF's 3D right-handed coordinate system situated in a desert scene.

```
<Viewport3D ... >

    <Viewport3D.Camera>
      <PerspectiveCamera described below />
    </Viewport3D.Camera>

    <!- The ModelVisual3D wraps around the scene's content ->
    <ModelVisual3D>
      <ModelVisual3D.Content>
        <Model3DGroup>
            Lights and objects will be specified here.
        </Model3DGroup>
      </ModelVisual3D.Content>
    </ModelVisual3D>
 </Viewport3D>
```

We want the camera to be initially placed so that it lies well outside the pyramid, but is close enough to ensure that the pyramid dominates the rendered image. So we will position the camera at $(57, 247, 41)$ and "aim" it toward the pyramid's center point.[1]

```
  <PerspectiveCamera
    Position="57, 247, 41"
    LookDirection="-0.2, 0, -0.9"
    UpDirection="0, 1, 0"
    NearPlaneDistance="0.02" FarPlaneDistance="1000"
    FieldOfView="45"
        />
```

The camera is a geometric object, placed in the scene's world coordinate system (via the `Position` attribute) and oriented via two vectors.

1. Determining the numeric values that make a scene "look right" is often the result of trial and error; thus, scene design is greatly facilitated by interactive 3D development environments that offer instant feedback while a designer experiments with the placement and orientation of objects, cameras, and lights.

- `LookDirection` is a vector specifying the direction of the camera projection, in world coordinates. Think of the `LookDirection` as the center line of the barrel of the lens that you are pointing toward the central object.
- `UpDirection` rotates the camera about the look-direction vector to specify what will constitute the "up" direction to the viewer. In our example, in which the ground plane is the *xz*-plane, the up-direction vector $[0, 1, 0]^{\mathbf{T}}$ simulates a stationary tripod on the desert sand set up to photograph the pyramid, with the image plane perpendicular to the ground plane.

Additionally, the width of the camera's field of view can be specified as an angle in degrees; for example, a wide-angle lens can be simulated with a wide field of view such as 160°. Also, two **clipping planes** can be specified to prevent anomalies that occur when objects are too close to the camera (`NearPlaneDistance` property), and to reduce computation expense by ignoring objects that are very distant (`FarPlaneDistance` property) and thus too small to resolve due to perspective foreshortening.

Next we light the scene with nondirectional **ambient light** that applies a constant amount of illumination on all surfaces regardless of location or orientation. (We'll supplement this with more realistic lighting later.) Ambient light ensures that each surface is illuminated to some degree, preventing unrealistic pure-black regions on surfaces facing away from light sources. (Such regions would, in the "real world," be subjected to at least some level of interobject reflection.) The amount of ambient light is kept to a minimum when used in combination with other lighting, but in this initial scene, ambient is the only lighting type, so we'll use full-intensity white to ensure a bright rendering. We specify this light by adding an `AmbientLight` element inside the `Model3DGroup` element:

```
<AmbientLight Color="white"/>
```

Inline Exercise 6.1: At this point, we recommend that you start running module #1 ("Modeling Polyhedra. . . ") of the laboratory software for this chapter, available in the online resources. We will refer to this module throughout this section.

6.2.1.2 Placing the First Triangle

It is no coincidence that we have selected a pyramid as our first example object, since the triangular mesh is the only 3D primitive type currently supported by WPF (and is the most common format generated by interactive modeling applications). The first step in creating a 3D object is to define a resource object of type `MeshGeometry3D` by providing a list of 3D `Positions` (vertices) and a list of triangles. The latter is specified via the `TriangleIndices` property, in which we specify each triangle via a sequence of three integer indices into the zero-based `Positions` array. In this case, we are specifying a mesh containing just one triangle, the first face of our pyramid. Figure 6.4 shows a tabular representation of the mesh.

It is the programmer's responsibility to identify the front side of each triangle, because the front/back distinction is important, as we will soon discover. Thus, when specifying a vertex triplet in the `TriangleIndices` array, list the vertex indices in a counterclockwise order from the point of view of someone

Positions			
Index	*X*	*Y*	*Z*
0	0	75	0
1	–50	0	50
2	50	0	50
TriangleIndices			
0,1,2			

Figure 6.4: Tabular representation of the geometric specification of a single-triangle mesh.

facing the front side. For example, consider how we are presenting the vertices of the single triangle of our current model. In the `TriangleIndices` array, the three indices into the `Positions` array are in the order $0, 1, 2$. Thus, the vertices are in the sequence $(0, 75, 0)$, $(-50, 0, 50)$, $(50, 0, 50)$, which is a counterclockwise ordering, as shown in Figure 6.5.

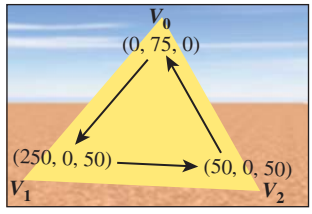


Figure 6.5: Identification of the front side of a mesh triangle via counterclockwise ordering of vertices.

The XAML representation of this mesh is as follows:

```
1   <MeshGeometry3D x:Key="RSRCmeshPyramid"
2      Positions="0,75,0 -50,0,50 50,0,50"
3      TriangleIndices="0 1 2" />
```

This mesh specification appears in the resource section of the XAML, and thus is similar to a WPF 2D template resource in that it has no effect until it is used or instantiated. So the next step is to add the 3D object to the viewport's scene by creating an XAML element of type `GeometryModel3D`, whose properties include at least the following.

- The geometry specification, which will be a reference to the geometry resource we created above.
- The material specification, which is usually also a reference to a resource. The material describes the light-reflection properties of the surface; WPF's materials model provides approximations of a variety of material types, as we shall soon see in Section 6.5.

Let's keep things basic for now, and define a basic solid-yellow material resource, earmarked for the front side of each surface, giving it a unique key for later referencing:

```
1   <!- Front material uses a solid-yellow brush ->
2   <DiffuseMaterial x:Key="RSRCmaterialFront" Brush="yellow"/>
```

We now are ready to create the element that will add this single-triangle mesh to our scene. We place this XAML as a child of the `Model3DGroup` element:

```
1       <GeometryModel3D
2           Geometry="{StaticResource RSRCmeshPyramid}"
3           Material="{StaticResource RSRCmaterialFront}"/>
```

Our image of the model now appears as shown in Figure 6.6.

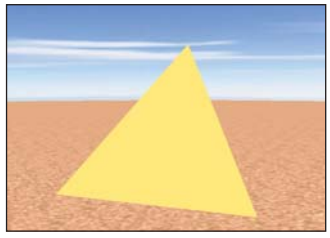


Figure 6.6: First triangle's front side rendered using a uniformly yellow material.

Inline Exercise 6.2: In the lab, select the "Single face" option in the model drop-down list. If you wish, click on the XAML tab to examine the source code generating the scene. Activate the turntable to rotate this triangle around the y-axis.

If we were to rotate this triangular face 180° around the y-axis, to examine its "back side," we would obtain the puzzling image shown in Figure 6.7.

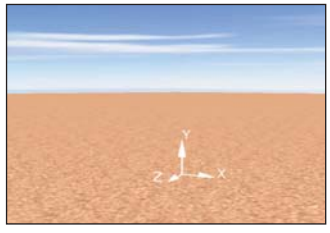


Figure 6.7: First triangle's back side, invisible due to lack of specification of a material for the back side.

The triangle disappears due to a rendering optimization: WPF by default does not render the back sides of faces. This behavior is satisfactory for the common case of a "closed" object (such as the pyramid we intend to construct) whose exterior is composed of the front sides of the mesh's triangles. For such a closed figure, the back sides of the triangles, whose surface normals point toward the object's interior, are invisible and need not be rendered.

For our current simple model—a lone triangle—it is useful to disable this optimization and show the back face in a contrasting color, by setting the `BackMaterial` property to refer to a solid-red material that we will add to the resource section with the key `RSRCmaterialBack`:

```
1  <GeometryModel3D
2    Geometry="{StaticResource RSRCmeshPyramid}"
3    Material="{StaticResource RSRCmaterialFront}"
4    BackMaterial="{StaticResource RSRCmaterialBack}"/>
```

As a result, the back face now is visible when the front faces away from the camera, as shown in Figure 6.8.

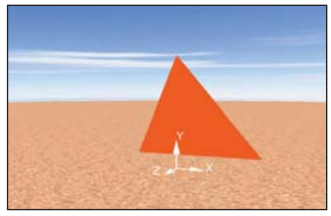


Figure 6.8: First triangle's back side, rendered using a uniformly red material.

Inline Exercise 6.3: In the lab, check the box labeled "Use back material" and keep the model spinning.

With the first face of the pyramid now in place, let's add the second face, using the strategy represented in a tabular form in Figure 6.9. Notice that the vertices shared by the two faces (V_0 and V_2) have separate entries in the `Positions` array, effectively being listed redundantly.

```
1  <MeshGeometry3D x:Key="RSRCmeshPyramid"
2     Positions="0,75,0  -50,0,50  50,0, 50
3                0,75,0   50,0,50  50,0,-50"
4     TriangleIndices="0 1 2    3 4 5" />
```

Positions			
Index	*X*	*Y*	*Z*
0	0	75	0
1	–50	0	50
2	50	0	50
3	0	75	0
4	50	0	50
5	50	0	–50

TriangleIndices
0,1,2
3,4,5

Figure 6.9: Tabular representation of geometric specification of a two-triangle mesh.

Inline Exercise 6.4: In the lab, select the "Two faces" model and keep the model spinning.

The result appears in two snapshots of the spinning model shown in Figure 6.10.

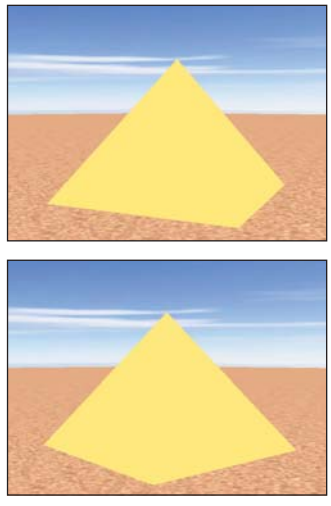


Figure 6.10: Renderings of the partial pyramid in two distinct orientations, in an environment containing only ambient light.

6.2.2 Producing More Realistic Lighting

There is an obvious problem with this rendering: A single constant color value is being applied to both faces of the model, regardless of orientation. But in a daytime desert scene, one would expect variation in the brightness of the pyramid's faces, with a bright reflection from those facing the sun and lesser reflection from those facing away from the sun.

The use of the artificial construct of nondirectional ambient lighting as the sole light source produces this unrealistic appearance. In the real world, lights are part of the scene and the light energy hitting a point P on a surface has a direction (a vector, represented by the symbol ℓ, from the light source to point P). Moreover, the energy reflected toward the camera from P is not a constant, but instead is based on a number of variables such as the camera's location, the surface's orientation at P, the direction ℓ, the reflection characteristics of the object's material, and others. In Section 6.5 we will examine a lighting equation that takes many of these kinds of variables into consideration, but here let's take a high-level look at one example of a more realistic light source: the **point light,** which is a **geometric light,** having a position in the scene and radiating light in all directions equally (as shown in Figure 6.11). A point light's presence can introduce a great deal of variation into a scene via its infinite set of values for ℓ, which ensures that each point on a surface facing the light receives its energy from a unique ℓ direction.

We'll examine the characteristics and impact of point and other geometric light sources in more detail in Section 6.5, but here let's start with a simplification: the "degenerate case" of a point light source that is located at an infinite distance from the scene. WPF distinguishes this kind of light source from geometric ones, calling this a **directional light.** Its rays are parallel (with a constant ℓ, as shown in Figure 6.12), providing an approximation of light from an infinitely distant sun.

So, let's replace the ambient light source with a directional one. We'll specify its color as full-intensity white, and its direction ℓ as $[1, -1, -1]^{\mathbf{T}}$ to simulate the sun's position being behind the viewer's left shoulder:

```
<DirectionalLight Color="white" Direction="1, -1, -1" />
```

The direction ℓ for this light (shown as a scene annotation in the lab and in Figure 6.13) is at a 45° angle relative to all three axes, and when projected onto the xz ground plane, it is a vector that travels from the $(-x, +z)$ quadrant to the $(+x, -z)$ quadrant.

Inline Exercise 6.5: A static 2D image is not the best way to depict 3D information like our light's ℓ value, so we recommend that you use the lab to follow along with this section's discussion. Select directional lighting, note the "Light direction" annotation, and use the trackball-like mouse interaction within the viewport to move around in the scene.

As introduced in Section 1.13.2, for a completely diffuse surface like that of our pyramid, the light is reflected with equal brightness in all viewer directions and is therefore view-angle-independent. The brightness of the reflected light is only dependent on how directly the incident light hits the surface. Figure 6.14 demonstrates how this directness is measured, by determining the angle θ between ℓ and the surface normal **n.** The larger the value of θ, the more oblique the light is, and thus the less energy reflected.

Given the angle θ and the incident light's intensity I_{dir}, the reflected intensity is calculated by Lambert's cosine rule, which was introduced in Section 1.13.2:

$$I = I_{\text{dir}} \cos\theta. \tag{6.1}$$

We've described light with the word "intensity" without a precise definition. Intensity is a vague term, not even defined in the international standard for units. Precisely defining "how much light is arriving here" turns out to be a bit tricky. Chapter 14 gives some initial ideas, and Chapter 26 gives full detail.

It does seem as if getting "intensity" wrong ought to bring our work to a stop, but the human visual system is coming to our rescue. It's mostly sensitive to changes in light (either over time, or between nearby arriving-light directions), and exact magnitudes don't seem to matter much. In fact, if you take a grayscale image with values between 0 and 1 and you replace each gray-value g with g^2 or g^3 and redisplay, the image is still perfectly understandable.

By the way, the word "brightness" is used to describe the *perception* of light; it's a psychophysical measurement rather than a physical one. Many papers in graphics have nonetheless used it as a proxy for "intensity."

For now, treat "intensity" as meaning "some sort of measurement of light, where bigger intensity means more light," and wait until Chapter 26 to get the whole story.

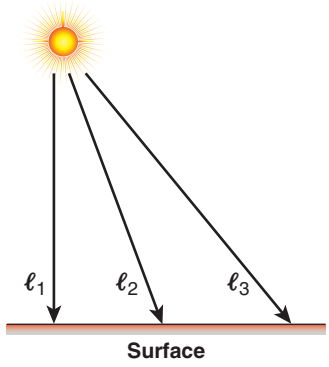


Figure 6.11: Rays emanating from a point light source in the scene, striking points on a planar surface at an infinite variety of angles.

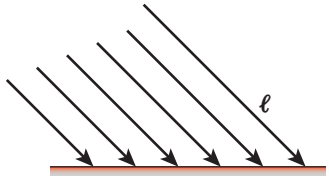


Figure 6.12: Rays emanating from a directional light source, infinitely distant from the planar surface, striking the surface's points at identical angles.

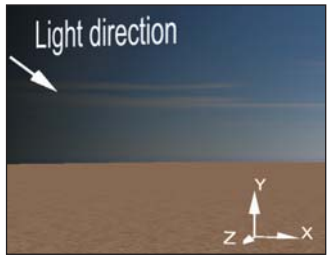


Figure 6.13: Our desert scene's coordinate system with annotation showing the direction of the rays emanating from the directional light source.

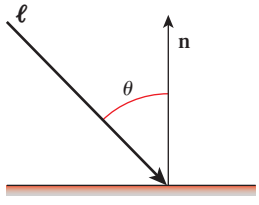


Figure 6.14: The angle θ, defined as the angle between the incoming light direction ray ℓ and the surface normal **n**.

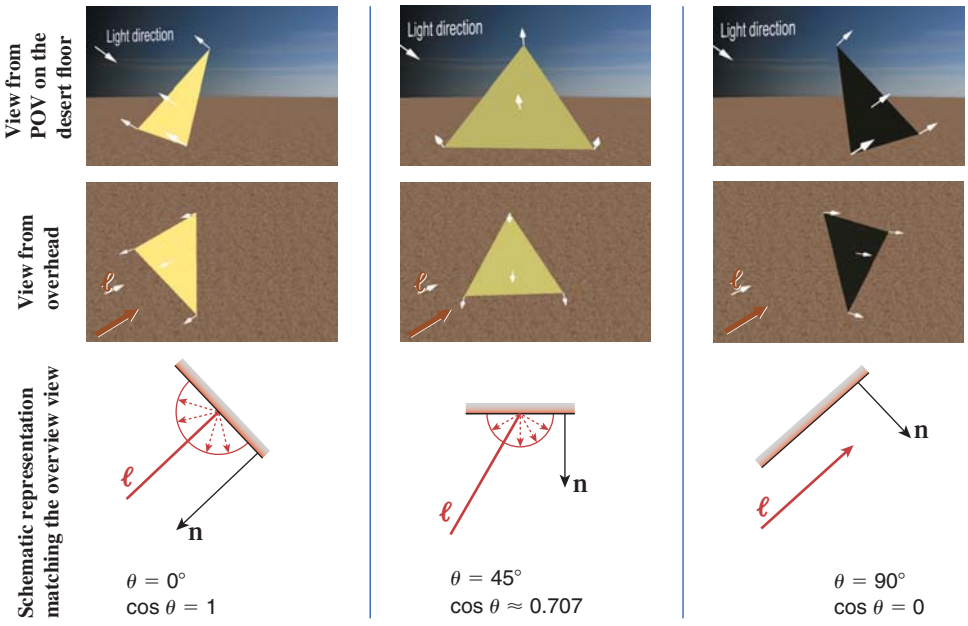


Figure 6.15: Brightness computed by Lambert's cosine law for three values of θ.

Figure 6.15 demonstrates this equation's effect on a single-triangle model, for various values of θ achieved by rotating the pyramid on an invisible turntable. In the figure, the length of each dashed red vector depicts the intensity of the reflected energy in its indicated direction, for the given value of θ. With perfectly diffuse reflection, light is reflected with equal intensity in all directions, and therefore, that length is a constant for any given value of θ. The locus of the endpoints of the reflection vectors for all possible reflection angles is thus a perfect hemisphere in the case of diffuse reflection. (In our 2D figure, of course, this envelope appears as a semicircle.)

This equation, being independent of viewing angle, cannot simulate glossy materials such as metals and plastic that exhibit highlights at certain viewing angles. Another oversimplification in the equation is an unrealistic lossless reflection of all incoming light energy when $\theta = 0°$. In reality, some amount of light energy is absorbed by the material and thus is not reflected. Section 6.5 describes a more complete model that corrects these and other problems.

With directional lighting replacing ambient lighting, processed by the Lambert lighting model, the results are more realistic, as you can see in the images in Figures 6.16 and 6.17.

Inline Exercise 6.6: In the lab, activate the directional lighting by selecting "directional, over left shoulder" and enabling turntable rotation. Observe the dynamic nature of the lighting of the yellow front face; it may help to occasionally pause/resume the turntable's motion. Observe the display of the value of θ and $\cos\theta$, and note how the yellow face approaches zero illumination as θ approaches and passes 90°. Select different models and examine the two-face and full four-face models in this new lighting condition.

Figure 6.16: Rendering of the pyramid with directional lighting, with θ close to 90° for the rightmost visible face.

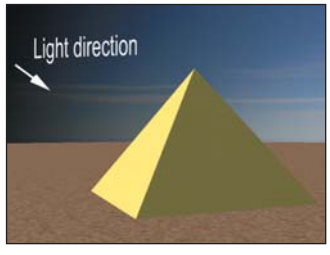


Figure 6.17: Rendering of the pyramid with directional lighting, with θ approximately 70° for the rightmost visible face.

Lambert's cosine rule for idealized diffuse reflection has two key characteristics: (1) The reflected intensity is independent of view angle and (2) it depends only on the cosine of the angle between the incoming light direction ℓ and the normal at a point on the surface. To gain an intuitive understanding of these phenomena, find a matte surface such as a clean chalkboard or a painted wall without any sheen, and point a bright light at the surface. Now pick a bright spot on the illuminated area and look at it from a variety of locations through a tube with a diameter so tiny that all you see is a uniformly lit "dot" (simulating what a radiometer would detect). Note that as you move your point of view, the dot's apparent brightness will remain constant, while it would vary if you performed this same experiment with a shiny surface. Varying the angle of incidence of the light source, however, will cause the reflected brightness to vary with the cosine of the angle. If you are curious about the math behind this rule, consult Section 7.10.6.

6.2.3 "Lighting" versus "Shading" in Fixed-Function Rendering

The Lambert equation presented above, and the more complete equation presented in Section 1.13.1, are examples of functions that compute the amount of light energy that is reflected from a given surface point P toward the specified camera position.

A lighting equation, like the Lambert equation, is an algorithmic representation of the way a surface's material reflects light. From a theoretical point of view, a renderer processes a given visible surface by "loading" the lighting equation for its material, and "executing" it for points on that surface. (Interestingly, in programmable-pipeline hardware, this abstraction isn't too far from the truth!) For which surface points should the equation be executed? A reasonable approach is to perform the calculation once for each pixel covered by the surface's rendered image, executing the equation for a representative surface point for each such pixel. Offline rendering systems use an approach like this, but that strategy is too computationally expensive for real-time rendering systems running on today's commodity hardware. The approach taken by many such systems—including fixed-function pipelines such as WPF's, as well as programmable pipelines—is to compute the lighting only at key points on the surface, and to use lower-cost shading rules[2] to determine values for surface points lying between the key points.

For example, let's examine the simplest shading technique, known as **flat shading** or **constant shading,** in which one vertex of each triangle is selected as the key vertex for that triangle. The lighting equation is executed to compute the illumination value for that vertex, and the entire triangle is filled with a copy of that value. An example rendered image using flat shading is shown in Figure 6.18, in which we've highlighted three triangles and the key vertex that was the determinant for each.

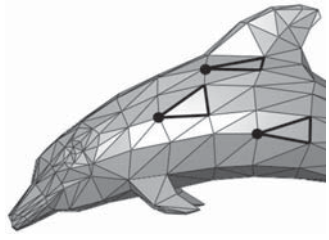


Figure 6.18: Flat-shaded rendering of a dolphin mesh model, with three triangles highlighted to demonstrate the concept of the key vertex.

Flat shading may be appropriate for pyramids, but as Figure 6.18 shows, when the triangular mesh is approximating a curved surface a more sophisticated

2. As discussed in Section 27.5.3, this classic use of the term "shading" to refer to efficient determination of lighting at interior points conflicts with modern uses of the terms "shading" and "shader."

shading technique is clearly needed. In the next section, we describe a popular real-time shading technique designed to address the problem of rendering curved surfaces.

6.3 Curved-Surface Representation and Rendering

Survey the room around you and you'll find that most objects have some curved surfaces or rounded edges. A purely faceted polyhedron like our simple pyramid is quite rare in the "real world." Thus, in most cases, a triangular mesh in a 3D scene is not being used to represent an object exactly, but rather is being used to *approximate* an object.

For example, we can approximate a circular cone using a many-sided pyramid. With just 16 faces, flat shading produces a fairly good approximation of a cone (as seen in Figure 6.19), but it doesn't really "fool the eye" into accepting it as a curved surface.

Increasing the number of facets (e.g., to 64 sides, as shown in Figure 6.20) does help improve the result, but the approximation is still apparent. Attempting to solve the problem merely by increasing the mesh's resolution is not only expensive (in terms of storage/processing costs) but also ineffective: If the camera's position is moved toward the mesh, at some point the faceting will become apparent.

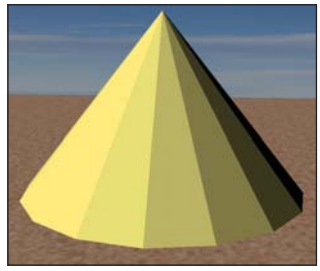


Figure 6.19: Flat-shaded rendering of a cone with 16 sides.

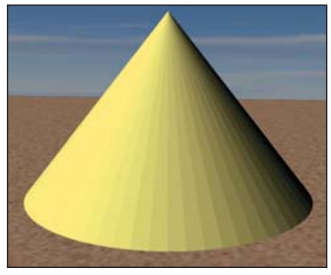


Figure 6.20: Flat-shaded rendering of a cone with 64 sides, reducing (but not eliminating) the obvious faceting.

Inline Exercise 6.7: You might want to visit the "Modeling Curved Surfaces" module of the laboratory to see the effect of changes in the facet count when flat shading is in effect. You can zoom in/out by dragging the mouse (when the cursor is within the viewport) while holding down the right mouse button. Note how increasing the number of facets can only fool the eye at a distance—zooming in exposes the fraud easily. Note also that motion of the object makes the approximation even more obvious, especially at the bottom edge.

6.3.1 Interpolated Shading (Gouraud)

The task of finding an efficient way to produce acceptable images of curved surfaces from low-resolution mesh approximations was particularly urgent in the early days of computer graphics, when computer memory was measured in kilobytes and processors were many orders of magnitude less powerful than they are today. Per-vertex lighting with flat shading was widely used, but there was an obvious need for a shading technique that would fool the eye and allow the rendered image to approximate the curved surface represented by the mesh, even for a low-resolution mesh, at minimal processor and memory cost. In the early 1970s, University of Utah Ph.D. student Henri Gouraud refined a shading technique based on interpolation of intensity values at mesh vertices, using algorithms like those described in the opening sections of Chapter 9. To appreciate the difference in quality between flat shading and Gouraud shading, compare the two renderings of the Utah teapot in Figures 6.21 and 6.22.

Figure 6.21: Flat-shaded rendering of the classic "Utah" teapot model.

Figure 6.22: Gouraud-shaded rendering of the same teapot model.

Let's first examine Gouraud interpolation in two dimensions. In Figure 6.23, the curved 2D surface is shown in yellow, the approximation mesh of 2D line segments in black, and the vertices in green. At each vertex, the lighting model (in this case, diffuse Lambert illumination) has computed a color for that

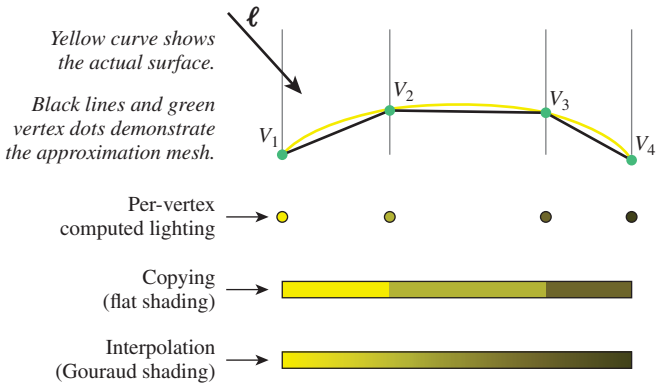


Figure 6.23: Comparison of flat shading and Gouraud shading, two different techniques for determining intensity values between the vertices at which lighting calculations were performed.

vertex. The result of the process of shading (to compute the color across the interior points) is shown for both flat shading and Gouraud interpolated shading.

As we have seen, the Lambert lighting equation depends on the value of **n**, the surface normal. Thus, to produce the color value at vertex V, the renderer must determine what we call the **vertex normal**—that is, the surface normal at the location of V. How should this be determined?

If the curved surface is analytical, for example, a perfect sphere, the equation used to generate the surface can provide the surface normal for any point. However, the approximation mesh itself is often the only information known about the surface's geometry. This limitation is alleviated by use of Gouraud's simple strategy for determining the vertex normal via averaging.

In 2D, the vertex normal is computed by averaging the surface normals of the adjacent line segments, as shown in Figure 6.24. For example, the vertex normal for V_2 is the average of the surface normals for the line segments $\overline{V_1V_2}$ and $\overline{V_2V_3}$.

In 3D, the vertex normal is computed by averaging the surface normals of all adjacent triangles, as depicted in Figure 6.25 for a scenario in which four triangles share the vertex.

The success of this technique lies in the fact that, for a mesh that is sufficiently fine-grained, the vertex normal computed via averaging is typically a very good approximation of the surface normal of the actual surface being approximated. (Chapter 25 discusses some limitations of this approximation.) For example, in the 2D representation shown in Figure 6.24, note that $\mathbf{n}_{v_2}$ looks like a very good estimate of the normal to the yellow surface at the location of V_2. The accuracy of the computed normal is of course dependent on the granularity of the mesh, and the granularity requirement increases in areas of discontinuity.

Inline Exercise 6.8: We suggest that you return to the curved-surface module of the lab, and select "Gouraud shading." Note the success of the interpolation even with a minimal number of facets. You will notice that, if the granularity is extremely low (e.g., 4 or 8) and/or the model is rotating, the silhouette of the cone—its bottom edge where it meets the ground—unfortunately continues to exhibit the mesh's structure, reducing the effectiveness of the "trick."

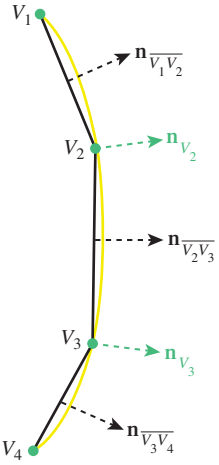


Figure 6.24: Calculating a vertex normal in 2D, as an average of the normals of the two adjacent line segments.

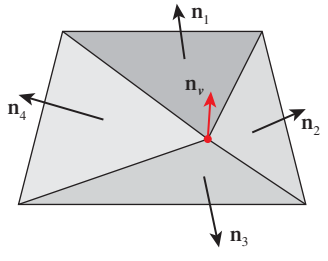


Figure 6.25: Calculating a vertex normal in 3D, as an average of the surface normals of all triangles sharing the vertex.

6.3.2 Specifying Surfaces to Achieve Faceted and Smooth Effects

WPF's rendering engine uses Gouraud shading unconditionally; in other words, WPF does not offer a rendering "mode" allowing the application to select between flat shading and smooth shading. Yet, in Section 6.2 we were able to create a pyramid with a faceted appearance, and in Figure 6.21 we showed a WPF-generated flat-shaded depiction of the teapot. How were we able to force WPF to generate a flat look?

Examine Figure 6.9 to review how we specified the first two pyramid faces in Section 6.2. Each shared vertex was placed redundantly in the `Positions` list to ensure that each was referenced by only one face. This causes the calculation of the vertex normal to involve no averaging, since each vertex is attached to only one triangle. The normal at each vertex will thus match the triangle's surface normal. Since Gouraud smoothing at edges and vertices is based on the averaging of vertex normals, the use of an unshared vertex effectively disables smoothing at that vertex.

Now, suppose that these two faces are part of a pyramid approximating a circular cone. In this case, we *do* want smooth shading. So, let's specify the same two triangular faces, but in the `TriangleIndices` list, let's "reuse" the vertex data for the shared apex (V_0 in Figure 6.27) and the shared basepoint (V_2):

```
1  <MeshGeometry3D x:Key="RSRCmeshPyramid"
2    Positions="0,75,0  -50,0,50  50,0, 50  50,0,-50"
3    TriangleIndices="0 1 2  0 2 3" />
```

In this specification (shown in Figure 6.26), vertices V_0 and V_2 are shared by two triangles, resulting in their vertex normals being computed via the averaging technique. The result is shown in Figure 6.28, with a smoothing of the edge between V_0 and V_2 being quite apparent.

In summary, WPF mesh specification requires following a simple rule: You should share vertices that need to participate in Gouraud smoothing, and you should duplicate vertices (i.e., so that each is referenced by only face) that need to be points of discontinuity in the rendered image.

You will find a need for both of these techniques, because typically, complex objects are a hybrid of both smooth curved surfaces and discontinuities where a crease or seam is located and needs to be visible in the rendering. Examples of discontinuities include the location on a teapot where the spout joins the body, and the seam on an airplane where wings join the fuselage. By using vertex sharing appropriately, you can easily represent such hybrid surfaces.

Positions			
Index	*X*	*Y*	*Z*
0	0	75	0
1	–50	0	50
2	50	0	50
3	50	0	–50

TriangleIndices
0,1,2
0,2,3

Figure 6.26: Tabular representation of geometric specification of a two-triangle mesh with reuse of shared vertices (the apex and the shared base vertex).

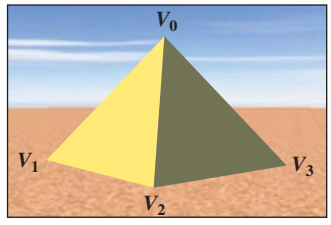


Figure 6.27: Assignment of indices to the vertices in our pyramid model.

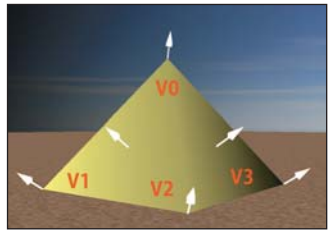


Figure 6.28: Gouraud shaded pyramid, produced in WPF by specifying that the two triangles share vertices V_0 and V_2, causing their vertex normals to be the average of the surface normals of the two triangles.

6.4 Surface Texture in WPF

We posit that any computer graphics professional confronted with the question, "What was the single most effective reality-approximation trick in the early history of real-time rendering?" will scarcely skip a beat before exclaiming, "Texture mapping!" When faced with the need to display a "rough" or color-varying material such as gravel, brick, marble, or wood—or to create a background such as a grassy plain or dense forest—it is not advisable to try to create a mesh

representing every detail of the fine-grained structure of the material. Consider the complexity of the mesh that would be required to model the dimples and crannies of the rough-hewn stone of an ancient pyramid—our simple four-triangle mesh would balloon into a mesh of millions of triangles, exploding the memory and processing requirements of our application.

Through the "trick" of **texture mapping** (wrapping a 3D surface with a 2D decal), complex materials (such as linen or asphalt) and complex scenes (such as farmland viewed from an airplane) can be roughly simulated with no increase in mesh complexity. (Chapters 14 and 20 discuss this idea in detail.) For example, the desert sand in our scene was modeled as a square (two adjacent coplanar right triangles) wrapped with the image shown in Figure 6.29.

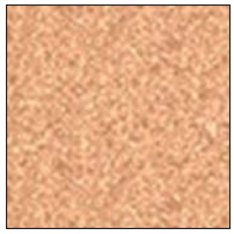


Figure 6.29: Square 64 × 64 image of a tan-hued pattern to simulate a sandy desert floor.

"Texturing" a 3D surface in WPF corresponds to the act of covering an object with a stretchable sheet of decorated contact paper. Theoretically, we must specify, for each point *P* on the surface, exactly which point on the paper should touch point *P*. In practice, however, we specify this mapping only for each *vertex* on the surface, and interpolation is used to apply the texture to the interior points.

This specification requires a coordinate system for referring to positions within the texture image. By convention, instead of using exact integer pixel coordinates, we refer to points on the image using the floating-point texture coordinate system shown in Figure 6.30, whose axes *u* and *v* have values limited to the range 0 to 1.

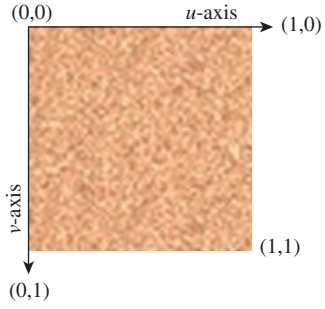


Figure 6.30: Floating-point texture coordinate system applied to the sand-pattern image, with the origin located at the upper-left corner.

In XAML, the first step is to register the image as a diffuse material in the resource dictionary. We have used a solid-color brush previously, but here we create an image brush to define the material:

```
<DiffuseMaterial x:Key="RSRCtextureSand">
    <DiffuseMaterial.Brush>
       <ImageBrush ImageSource="sand.gif" />
    </DiffuseMaterial.Brush>
</DiffuseMaterial>
```

The next step is to register into the resource database the simple two-triangle mesh representing the ground, using the same technique as before, but adding a new attribute to specify the corresponding texture coordinate for each vertex in the `Positions` array:

```
<MeshGeometry3D x:Key="RSRCdesertFloor"
    Positions="-9999, 0, -9999
                9999, 0, -9999
                9999, 0,  9999
               -9999, 0,  9999"
    TextureCoordinates=" 0,0   1,0   1,1   0,1 "
    TriangleIndices="0 1 3   1 2 3"
/>
```

Since this is a mapping from a square (the two coplanar triangles in the 3D model) to a square (the texture image), we declare texture coordinates that are simply the corners of the unit-square texture coordinate system, as shown in Figure 6.31.

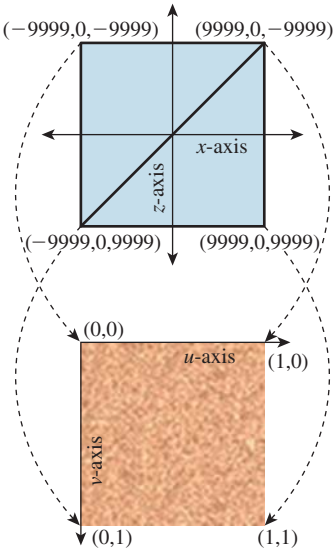


Figure 6.31: Mapping world-coordinate vertices on the two-triangle model of the desert floor to corresponding texture coordinates.

With the material and geometry registered as resources, we are ready to instantiate the desert floor:

```
<GeometryModel3D
      Geometry="{StaticResource RSRCdesertFloor}"
      Material="{StaticResource RSRCtextureSand}"/>
```

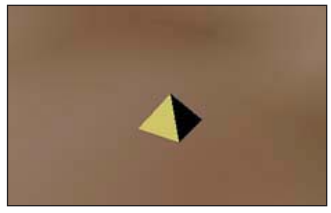


Figure 6.32: Sand texture overstretched to cover the entire desert floor.

The result is shown in Figure 6.32, from a point of view high above the pyramid. The result is not acceptable; there is some subtle variation in the color of the desert floor, but the color patches are huge (in comparison with the pyramid).

The problem is that our tiny 64 × 64-pixel sand decal (which was designed to represent about one square inch of a sandy floor) has been stretched to cover the entire desert floor. The result looks nothing like sand even though our decal provides fairly good realism when viewed unscaled.

Our failure to simulate desert sand here is a case of a reasonable texture image being applied to the model incorrectly. Implementing texturing in WPF requires choosing between two mapping strategies: tiling and stretching.

Figure 6.33: Square image of a brick pattern.

6.4.1 Texturing via Tiling

If the texture is being used to simulate a *material* with a consistent look and no obvious points of discontinuity (e.g., sand, asphalt, brick), the texture image is replicated as needed to cover the target surface. In this case, the texture is typically a small sample image (either synthetic or photographic) of the material, which has been designed especially to ensure that adjacent tiles fit together seamlessly. As an example, consider the texture image of Figure 6.33 showing six rows of red brick.

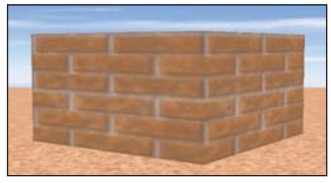


Figure 6.34: Result of stretching one copy of the brick texture onto each wall.

Applying it to each face of a rectangular prism without tiling produces a decent image (Figure 6.34) but the number of rows is insufficient for representing a tall brick fortress. Tiling allows the number of apparent brick rows to be multiplied, producing an image (Figure 6.35) that is more indicative of a tall fortress.

Consult the texture-mapping module of the lab for details on how to enable and configure tile-based texturing in WPF.

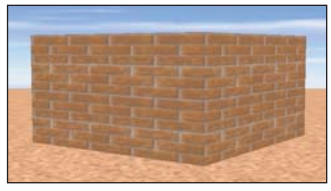


Figure 6.35: Result of tiling multiple copies of the brick texture onto each wall.

6.4.2 Texturing via Stretching

If a texture is being used as a substitute for a highly complex model (e.g., a city as seen from above, or a cloudy sky), the texture image is often quite large (to provide sufficiently high resolution) and may be either photographic or original artwork (e.g., if being used to represent a landscape in a fantasy world). Most importantly, this kind of texture image is a "scene" that would look unnatural if tiled. The correct application of this kind of texture image is to set the mesh's texture coordinates in such a way as to stretch the texture image to cover the mesh.

For example, in our desert scene, the background sky (as seen often in Figures 6.5 through 6.17) is modeled as a cylinder whose interior surface is stretch-textured with the actual sky photograph shown in Figure 6.36.

Figure 6.36: Image of a sky image.

Consult the texture-mapping module of the lab for details on how to enable and configure stretch-based texturing in WPF. More information, including algorithms for computing texture coordinates for curved surfaces and a discussion of common texture-mapping problems, is presented in Section 9.5 and in Chapter 20.

6.5 The WPF Reflectance Model

In Section 6.2.2, we presented Lambert's simple cosine rule for calculating reflected light from a diffuse surface. That simple equation is just one part of the complete WPF reflectance model, which is based on a classic approximation strategy that provides results of acceptable quality, without complex physics-based calculations, on a wide variety of commodity graphics hardware.

6.5.1 Color Specification

The word "color" is used to describe multiple things: the spectral distribution of wavelengths in light, the amount of light of various wavelengths that a surface will reflect, and the perceptual sensation we experience on seeing an object. Representing color precisely is a serious matter to which Chapter 28 is dedicated. Indeed, specifying color via RGB triples—the common approach in graphics APIs and drawing/painting applications—may well be the grossest of all the approximations made in computer graphics practice!

When describing a scene, we specify the colors of the light sources and of the objects themselves. In WPF, specifying the former is straightforward (see Section 6.2.2), but describing the latter is far more complex, requiring the separation of the material into three distinct components. Section 6.5.3 is dedicated to describing this specification technique and the effects it can achieve.

6.5.2 Light Geometry

The two WPF lighting types we have used thus far (ambient and directional) are useful approximations but decidedly unrealistic: They are not considered to be emanating from a specific point in the scene, and their brightness is uniform throughout the entire scene.

A **geometric light source** adds realism in that it is located in the scene and is **attenuated**—that is, the amount of energy reaching a particular surface point P is dependent on the distance from P to the light source. WPF offers two geometric light source types.

- A **point light** emanates energy equally in all directions, simulating a naked bulb suspended from a ceiling without any shade or baffles. Specification parameters include its position and the attenuation type/rate (constant, linear, or quadratic, as described in Section 14.11.9). The lab software for this chapter allows experimentation with this type of light source.
- A **spotlight** is similar, but it simulates a theatre spotlight in that it spreads light uniformly but restricts it to a cone-shaped volume.

Geometric lights are useful but should still be considered only approximations, since real physical light sources (described in Section 14.11.6) have volume and surface area, and thus do not emit light from a single point.

6.5.3 Reflectance

In our modeling of desert scenes throughout this chapter, we have applied material to our meshes by specifying either a solid color or a texture image. However, there is more to a material than its color. If you've shopped for interior-wall paint, you

know that you must also choose the finish (flat, eggshell, satin, semigloss), which describes how the painted surface will reflect light.

The physics of how light is reflected from a surface is extremely complex, so for decades, the fixed-function pipeline has relied on a classic approximation strategy called the **Phong reflectance (lighting) model** that yields an effective simulation of reflection at very little computational cost.[3] In the Phong model, a material is described by configuring three distinct components of reflection: ambient (a small constant amount of light, providing a gross simulation of inter-object reflection), diffuse (representing viewer-independent light reflected equally in all directions), and specular[4] (providing glossy highlights on shiny surfaces when the viewpoint is close to the reflection ray). The values calculated for the three components—Figure 6.37, (a) through (c)—are summed to produce the final appearance, shown in part (d) of that figure.

The independent nature of the diffuse and specular components allows us to generate the approximate appearance of materials having multiple layers with distinct reflectance characteristics. Consider a polished red apple: On top of its diffuse red layer lies a colorless waxy coating that provides glossy highlights based on the color of the light source (not of the apple). This same pattern of reflection is also very common in plastics, although it is not generated by a multilayer reflectance, but by the nature of the plastic material itself. Our blue plastic teapot (in Figure 6.37) shows this: Its glossy highlights have the colorless hue of the incoming white light, while the diffuse reflections have the blue hue of the plastic. In Section 6.5.3.3 we'll provide more detail on how to produce this effect. Of course, this simplistic technique of summing noninteracting layers is inadequate for complex materials such as human skin; Section 14.4 presents an introduction to richer, more accurate material models, and Chapter 27 gives full details.

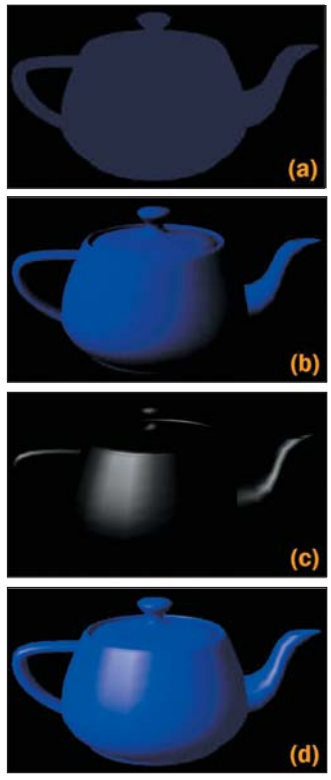


Figure 6.37: Renderings of a teapot, showing the contribution of each of the three components generated by the Phong lighting equation: (a) ambient, (b) diffuse, (c) specular, and (d) result generated by summing the contributions.

In this section, we describe the lighting equation for WPF's reflectance model, which is heavily based on, but not completely identical to, the Phong model. Let's first examine the equation's inputs that are specified as properties of the material resources (e.g., in WPF elements such as `DiffuseMaterial` and others enumerated in this chapter's online materials):

Symbol	Description	Format
C_d	Innate color of "diffuse layer"	$(C_{d,R}, C_{d,G}, C_{d,B})$
C_s	Innate color of "specular layer"	$(C_{s,R}, C_{s,G}, C_{s,B})$
k_a	Efficiency of diffuse layer at reflecting ambient light	$(k_{a,R}, k_{a,G}, k_{a,B})$
k_d	Efficiency of diffuse layer at reflecting directional/geometric light	$(k_{d,R}, k_{d,G}, k_{d,B})$
k_s	Efficiency of specular layer at reflecting directional/geometric light	$(k_{s,R}, k_{s,G}, k_{s,B})$

3. This non-physics-based reflectance model was invented early in the history of raster graphics and rendering research in the 1970s initially by University of Utah Ph.D. student Bui Tuong Phong and then slightly modified by Blinn, and has been remarkably long-lived, especially in real-time graphics.
4. For this chapter only, we are following the convention that "specular" refers to *somewhat* concentrated reflections rather than to perfect mirror reflection. Elsewhere specular means "mirrorlike," while sort-of-specular reflection is called "glossy." The use of "specular" for glossy follows both Phong's original paper and the WPF convention, but conflicts with its ordinary meaning of "having the properties of a mirror."

For a solid-color material, the innate colors for the diffuse and specular layers are constant across a surface. For a textured material, the texture image and texture algorithm together determine the diffuse-layer color at each individual surface point.

The three efficiency factors are each expressed as an RGB triple, with each entry being a number between 0 and 1, with 0 meaning "no efficiency" and 1 meaning "full efficiency." For example, we would specify $k_{a,R} = 0.5$ for a diffuse layer that reflects exactly half of the red component of the ambient light in the scene.

What we've called "reflection efficiency" here is closely related to the physical notion of reflectivity, which we'll examine in detail in Chapter 26.

Next, let's examine the inputs that are specified by or derived from the lights that have been placed in the scene, via any of WPF's light-specification elements, such as `DirectionalLight`:

Symbol	Description	Format
I_a	Color/intensity of the scene's ambient light	$(I_{a,R}, I_{a,G}, I_{a,B})$
I_{dir}	Color/intensity of a directional light source	$(I_{dir,R}, I_{dir,G}, I_{dir,B})$
I_{geom}	Color/intensity of a geometric light source	$(I_{geom,R}, I_{geom,G}, I_{geom,B})$
F_{att}	Attentuation factor for geometric lights	a single real number

A geometic light's actual contribution is subject to attenuation. The attenuation factor F_{att} is calculated for each surface point P, based on the light's characteristics and distance from P. Thus, the actual light arriving from at the surface point P from the geometric light source is

$$(F_{att}I_{geom,R};\ F_{att}I_{geom,G};\ F_{att}I_{geom,B}).$$

Now that we have enumerated all of the inputs, we are ready to examine the WPF lighting equation. Here is the equation that computes the intensity of the red light that reaches the camera from a specific surface point (we examine each component in detail below):

$$I_R = \tag{6.2}$$

$$\left(I_{a,R}\ \ k_{a,R}\ \ C_{d,R}\right) \tag{6.3}$$

$$+ \sum_{\text{directional lights}} \left(I_{dir,R}\ \ k_{d,R}\ \ C_{d,R}\ \ (\cos\theta)\right) \tag{6.4}$$

$$+ \sum_{\text{geometric lights}} \left(F_{att}\ \ I_{geom,R}\ \ k_{d,R}\ \ C_{d,R}\ \ (\cos\theta)\right) \tag{6.5}$$

$$+ \sum_{\text{directional lights}} \left(I_{dir,R}\ \ k_{s,R}\ \ C_{s,R}\ \ (\cos\delta)^s\right) \tag{6.6}$$

$$+ \sum_{\text{geometric lights}} \left(F_{att}\ \ I_{geom,R}\ \ k_{s,R}\ \ C_{s,R}\ \ (\cos\delta)^s\right) \tag{6.7}$$

The sums in the equations above are over all lights of various kinds, which we'll describe shortly.

If the scene contains multiple lights, and/or if the material uses multiple components (e.g., both ambient and diffuse) with high reflection efficiency coefficients, the computed result may be greater than 100% intensity, which has no

meaning since the pixel's range of red values is limited to the range of 0% to 100% illumination. Simple lighting models simply clamp excessive values to 100%. The "extra" illumination is thus discarded, which can have a negative impact on renderings, including unintended changes in hue or saturation. More sophisticated techniques for dealing with this situation—primarily the use of physical units—are discussed in Chapters 26 and 27. This failure is a consequence of the ill-defined nature of "intensity" that we discussed earlier, and presents a practical and widespread incidence of the failure of the Wise Modeling principle. The model of intensity as a number that varies from 0 to 1 was clearly a bad choice as the scale of scenes and complexity of lighting grew.

Inline Exercise 6.9: As we present the various components of the WPF reflectance model below, you may want to experiment with the effects of the configurable terms by using the lighting/materials laboratory software, and accompanying list of suggested exercises, available in the online resources for this chapter.

6.5.3.1 Ambient Reflection

Ambient light is constant throughout the scene, so the computation of the ambient reflection component is extremely simple and devoid of geometric dependencies. In the WPF reflectance model, there is no ambient innate color, so the material's diffuse color is used. The red component of the ambient reflection is computed via:

$$I_{a,R}\, k_{a,R}\, C_{d,R} \tag{6.8}$$

We encourage you to immediately perform the ambient-related experiments suggested in the lighting exercises presented online.

6.5.3.2 Diffuse Reflection

Directional light appears in the diffuse term of the reflectance model, computed by Lambert's cosine rule described in Section 6.2.2. Here is the red portion of this term, which takes into account all the directional lights in the scene:

$$\sum_{\text{directional lights}} I_{dir,R}\ \ k_{d,R}\ \ C_{d,R}\ \ (\cos\theta) \tag{6.9}$$

The sum here is over all directional lights in the scene. The angle θ will generally be different for each one, as will the intensity $I_{dir,R}$.

A similar equation sums over the set of *geometric* light sources, to take into account their attenuation characteristics. This equation is shown as part of the diffuse term in the full equation shown in the table above.

We encourage you to immediately perform the diffuse-related experiments suggested in the lighting exercises presented online.

Note that for solid-color materials, the distinction between the two terms C_d and k_d is unnecessary, as far as the math is concerned; you can think of them as a single term. That is, you can fix $k_{d,R}$ at 1.0 and use $C_{d,R}$ to achieve any effect, and conversely you could fix $C_{d,R}$ and specify only $k_{d,R}$. However, the distinction between the two terms is meaningful when the innate color C_d is being provided via texture mapping; in that case, there is a need for a $k_{d,R}$ factor affecting the reflection of the varying color C_d specified by the texture.

6.5.3.3 Specular Reflection

The specular reflectance term is the sum of a computed intensity for each directional and geometric light in the scene. Let's examine this sum for the directional lights:

$$\sum_{\text{directional lights}} I_{\text{dir}} \; k_{\text{s}} \; C_{\text{s}} \; (\cos\delta)^s \tag{6.10}$$

Most materials produce a specular reflection that is some mixture of its diffuse color and the light source's color, but the ratio of the former to the latter varies. You may have noticed that some shiny materials show a specular highlight that is essentially a brighter version of the diffuse color of the material. For example, the shiny highlights on a brass kettle illuminated by a bright light source are a "tinted" version of the light's color, highly affected by the innate brass color. But, as explained earlier, for plasticlike materials, specular highlights take on primarily the color of the light source rather than of the diffuse color. To achieve this plasticlike appearance, ensure that the product of k_{s} and C_{s} is a value not biased toward any component color (red, green, or blue) so as to preserve the hue of the incoming light.

This computation also includes a cosine-based attenuation factor, differing from that of Lambert's law in two ways. First, Lambert's law compares incoming light to the orientation of the surface alone, and is therefore viewpoint independent. However, specular reflection is highly viewpoint dependent, and thus relies on a different value δ, which in Phong's original formulation measures the angle between the reflection vector **r** (computed via the "angle of reflection equals angle of incidence" rule mentioned in Section 1.13.1) and the surface-to-camera vector **e**, as shown in Figures 6.38 and 6.39. The use of $\cos\delta$ ensures that the specular effect is strongest when the viewpoint lies on vector **r**, and weaker as the surface-to-camera vector varies more from vector **r**.

Second, whereas $\cos\delta$ ensures an intensity drop-off as **e** varies further from **r**, we also need to control how "fast" that drop-off is. For a perfect mirror, there is no gradual drop-off; rather, the reflection's intensity is at a maximum when the viewpoint is directly on vector **r**, and is zero if not. This binary situation doesn't occur in real-world materials; instead, there is a large variety in fall-off velocity among different materials. Thus, the equation provides for control of the amount of specularity through the variable *s*, known as the **specular exponent** (or **specular power**) of the material. Values of *s* for highly shiny surfaces are typically around 100 to 1000, providing a very sharp fall-off. A polished apple, on the other hand, might have an *s* of about 10, and thus a larger but dimmer area of measurable specular contribution. The lab software lets you experiment with different values of *s* to become familiar with its effect on specular appearance, and we encourage you to perform the specular-lighting exercises provided in the online material.

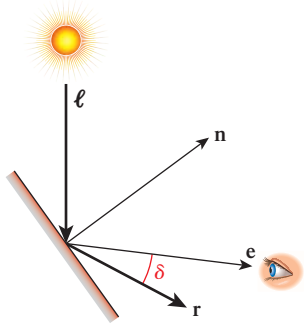


Figure 6.38: Phong's original technique for computing specular reflection, depicted in a context in which the camera position is very close to the reflection ray.

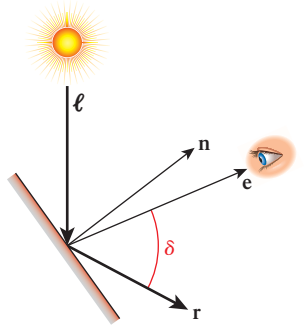


Figure 6.39: Phong's original technique for computing specular reflection, depicted in a context in which the camera position is not close to the reflection ray. The significant difference in the value of $\cos\delta$ makes an even greater difference when it's raised to a large power, so the specular term is nearly zero for this view.

The exponent *s* on the specular term is sometimes denoted *n,* which can conflict with the name of the normal vector, often written **n.** It's also sometimes denoted n_{s}, with the "s" denoting "specular." Our experience is that artists succeed better in adjusting the specularity when they're given a control that adjusts the *logarithm* of this exponent. As the artist moves a slider from 0 to 3, the specular exponent moves from 1 to 1000: the value 0 on the slider produces an appearance like latex paint, 1 like a polished apple, 2 like a shiny coin, and 3 like a mirror.

In this chapter, we describe the classic Phong definition of the angle δ. WPF and many popular fixed-function pipelines actually use a very similar, but computationally more efficient, variant known as the Blinn-Phong model, which uses a different approach for measuring δ, described in Section 14.9.3.

6.5.3.4 Emissive Lighting

Many rendering systems additionally offer the artificial notion of self-luminous **emissive lighting** that allows a surface to "reflect" light that is not actually present externally. Emission is independent of geometry and is not subject to any attenuation. The specification is simply a single color (solid or textured), which is added to the other three components to yield the final intensity value. Note that emission is most useful when emitting a texture—for example, to emulate a nighttime cityscape background or a star-filled sky—but it can also be used to model neon lights that have the particular "look" of a neon tube, although they do not illuminate anything else in the scene. We encourage you to perform the emissive-lighting exercises provided in the online material.

6.6 Hierarchical Modeling Using a Scene Graph

What's a desert without camels? In this section, we will design a simple articulated robotic camel with some pin joints (highlighted in Figure 6.40) supporting rigid-body rotations on a single axis. This section builds on the modeling and animation techniques you first encountered with the clock example in Chapter 2. While the XAML code examples below are particular to WPF, these techniques for composing and animating complex models are common to all scene-graph platforms.

Throughout this section we will refer you to activities in the "Hierarchical Modeling" module of this chapter's laboratory. We strongly recommended that you perform the lab activities while reading this section.

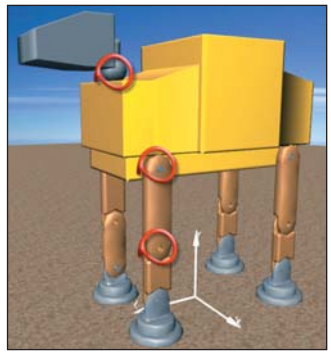


Figure 6.40: WPF's rendering of the camel constructed via hierarchical modeling, with joints for legs and neck animation.

6.6.1 Motivation for Modular Modeling

When designing any complex model, a developer should modularize the specification of the geometry by dividing the model into parts that we call **subcomponents.** There are many reasons for avoiding a monolithic (single-mesh) model.

- Materials are typically specified at the level of the mesh (e.g., the `GeometryModel3D` element in WPF). Thus, if you want the materials to vary (e.g., to render the camel's foot using a different material from its shin), you must use subcomponents.
- When a component appears at multiple places in the model (e.g., the camel's four legs), it is convenient to define it once and then instantiate it as needed. Reusability of components is as fundamental to 3D modeling as it is to software construction, and is a key optimization technique for complex scenes.
- The use of subcomponents facilitates the animation/motion of subparts. If a complex object is defined via a single mesh, movement of subparts requires editing the mesh. But if the design is modular, a simple transformation (of

the kind we used in animating the clock in Chapter 2) can be applied to a subcomponent to simulate a motion such as the bending of a knee.

- **Pick correlation** (the identification of which part of the model is the target of a user's click/tap action) is more valuable if your model is modularized well. If the user clicks on a single-mesh camel, the result of correlation is simply the identity of the camel as a whole. But if the camel is modularized, the result includes more detail; for example, "the shin of the front-left leg."
- Editing a single-mesh model is difficult due to interdependencies among the various parts—for example, extending the height of the camel's legs would, as a side effect, require revising all the vertices in the camel's head and torso. But when a model is defined using a hierarchy of subparts, the geometry of a subcomponent can be edited in isolation in its own coordinate system, and the assembly process can use transformations to integrate the parts into a unified whole.

These reasons are so compelling that we abstract them into a principle:

✓ **THE HIERARCHICAL MODELING PRINCIPLE:** Whenever possible, construct models hierarchically. Try to make the modeling hierarchy correspond to a functional hierarchy for ease of animation.

6.6.2 Top-Down Design of Component Hierarchy

One strategy for designing a complex model for animation is to analyze the target object to determine the locations of joints at which movement might be desired. For example, as depicted in Figure 6.40, we might want our camel to have knee and hip joints for leg movements, and a neck joint for head movement.[5] The joint locations, along with other requirements such as variations in materials, are then used to determine the necessary component breakdown. Let's focus first on just the camel's leg: We need to implement hip and knee joints, and we'd like the option of a distinct material for the foot.

The hierarchy shown in Figure 6.41 fits our needs. In the figure, we distinguish between primitive nodes (meshes with associated materials) and higher-level grouping nodes that combine subordinate grouping nodes and/or primitive nodes. Also, on the lines connecting components, we distinguish between two different types of modeling transformations. As you may recall from Chapter 2, we identify two slightly different uses of modeling transformations.

- An **instance transform** is used to position, resize, and orient a subcomponent in order to position it properly into a scene or into a higher-level composite object. In our clock application in Chapter 2, we used instance transforms to position the clock hands relative to the clock face, and to reshape a stencil clock hand to form the distinctive shapes of the hour

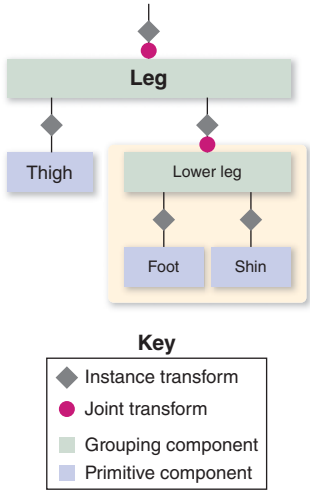


Figure 6.41: Scene graph of the camel-leg model. Here, and below, we use a beige background to highlight a portion of the graph that is being used as a component or submodel.

5. Here, our use of the term "joint" is informal and simply identifies locations at which we might want to implement an axis of rotation on a subcomponent to simulate a biological joint or a construction hinge. In sophisticated animation technologies, a joint is far more complex, may support more than one axis, and is often an actual object (distinct from the model's subcomponents) with structure, appearance, and behaviors derived from principles of physics and biomechanics.

and minute hands. Since the need for proper placement of a subcomponent may be present anytime instantiation is performed, we tend to include an instance transform on each subcomponent in our hierarchical design.

- A **joint transform** is used to simulate movement at a joint during animation. For example, the knee joint is implemented by a rotation transformation acting on the lower leg, and the hip joint is implemented by a rotation transformation acting on the entire leg. In our clock application, we used this to implement movement of the clock hands.

6.6.3 Bottom-Up Construction and Composition

Now we demonstrate how XAML can be used to construct the model. The order is bottom-up: first generating the primitive components (foot, shin, etc.) and then composing the parts to create the higher-level components.

The activities involved in bottom-up construction are summarized in this table:

Intended Goal	Where	WPF Element/Properties
Specify the geometry of a primitive component	Resource section	`MeshGeometry3D` element
Instantiate a primitive component	Inside the content of a viewport, as a direct child of the `Model3DGroup` representing its parent in the hierarchy	`GeometryModel3D` element `Name` property provides a unique ID useful for animation and pick correlation `Geometry` property points to the corresponding `MeshGeometry3D` resource `GeometryModel3D.Transform` property can be used to specify an instance transform and/or joint transform, often in the form of a `TransformGroup`
Construct a composite component	Inside the content of a viewport, as a direct child of the `Model3DGroup` representing its parent in the hierarchy	`Model3DGroup` element `Name` and `Model3DGroup.Transform` properties as described above

6.6.3.1 Defining Geometries of Primitive Components

The design of each primitive component should be an independent task, with its geometry specified in its own coordinate system, as we did for the clock hand in Chapter 2. The abstract coordinate system in which an object is specified is sometimes called the **object coordinate system**. For convenience, the component should be at a canonical position and orientation—for example, at the origin, centered on one of the coordinate axes, resting on one of the three coordinate planes.

Choosing a physical unit of measurement is optional, but composing the parts is simpler if the dimensions of components are *consistent.* For example, we have designed the foot as 19 units high (Figure 6.42) and the shin as 30 units high

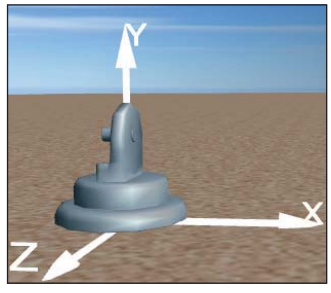


Figure 6.42: Rendering of the foot model, at its canonical position at the origin.

(Figure 6.43) to ensure that composing the two (to form the lower leg) requires only translation of the shin, as shown in Figure 6.45. (We'll work through the details of the lower leg construction in the next section.) No stretching/compressing (scaling) or rotation actions are necessary. Similarly, the thigh is consistently sized so that the full leg can be built by translating the thigh to connect it to the top of the lower leg.

Note that a typical interactive 3D modeling environment makes it very easy to build canonical, consistent atomic components, via features such as ruler overlays, templates of common volumes, and snap-to-grid editing assistance.

Naturally, if your design incorporates subcomponents obtained from third parties, inconsistencies can be expected, and additional transformations (e.g., resizing or reshaping via scaling) may be required to facilitate composing them with the components you designed. Similar transformation-based adjustments may be necessary when incorporating a completed composite model into an existing scene. For example, if we wish to place our completed camel model (which is well over 100 abstract units in height) into the pyramid scene we constructed previously, we will have to take into account that our scene's world coordinate system is a physical one with each unit representing 1 meter. Our camel, if placed in that scene without scale adjustment, would be 100 meters tall, towering over our 75-meter pyramid!

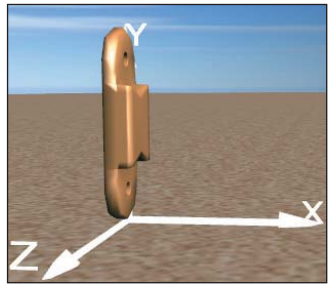


Figure 6.43: Rendering of the shin model, at its canonical position at the origin.

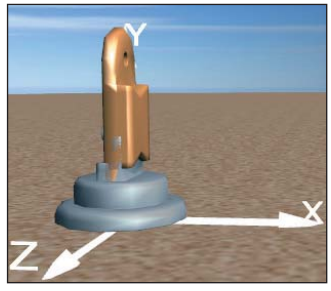


Figure 6.44: Rendering of a first draft of a lower-leg model, constructed by composing the two subcomponents without moving them from their canonical positions at the origin of the coordinate system.

As a little hint for future work, it's always nice if you can build the parts of your model in a way that makes it simple to place them in your scene. Aligning them with coordinate axes or planes is a good start, but making their proportions correct is also a good idea. It means that you can place the parts in the scene using only translation, rotation, and *uniform* scaling (i.e., scaling by the same amount in each axis). Such transformations turn out to be much easier to work with than more general scaling transformations.

6.6.3.2 Instantiating a Primitive Component

Once the primitive components have been designed and their meshes stored in the resource dictionary, each one can be "test-viewed" by instantiating it alone in the viewport, via creation of a `GeometryModel3D` element. The XAML shown below adds an instance of the foot primitive to our desert-scene viewport:

```
1 <ModelVisual3D.Content>
2   <Model3DGroup>
3       Lights will be specified here.
4      <GeometryModel3D Geometry="{StaticResource RSRCmeshFoot}"
5            Material=... />
6   </Model3DGroup>
7 </ModelVisual3D.Content>
```

Note that the instantiated foot is not being transformed, so it will appear at the origin of the world coordinate system of the scene, as you can see in Figure 6.42.

Inline Exercise 6.10: Use the Model listbox, along with the turntable feature, to examine the various primitive components of the camel in their canonical positions at their local origins. For example, the shin in its canonical position appears as shown in Figure 6.43.

6.6.3.3 Constructing a Composite Component

A composite node is specified via the instantiation of subcomponents within a `Model3DGroup` element; the subcomponents are accumulated into the composite node's own coordinate system.

6.6.3.4 Creating the Lower Leg

Here is a first draft of our camel's lower leg:

```
1 <Model3DGroup x:Name="LowerLeg">
2   <GeometryModel3D Geometry="{StaticResource RSRCmeshFoot}"
3         Material=... />
4   <GeometryModel3D Geometry="{StaticResource RSRCmeshShin}"
5         Material=... />
6 </Model3DGroup>
```

Testing this composite by instantiating it into the viewport yields the rendering shown in Figure 6.44.

Inline Exercise 6.11: Back in the lab, select the model "Lower leg (shin + foot)".

This unsatisfactory result, in which the two models co-inhabit space causing the foot's ankle region to intersect the shin, occurs because each component is, by design, positioned at the origin of its local coordinate system. When composing parts, we must use instance transforms to properly position the subcomponents relative to one another. Our goal is for the bottom of the shin to be connected to the top (ankle) part of the foot.

Thus, we need to translate the shin in the positive y direction; an offset of 13 units is satisfactory. Note that the foot is already properly positioned for its role in the lower-leg composite and thus needs no transform.

Below is our second draft of the XAML specification for this composite component (with the new lines of code highlighted). A couple of views of the result are shown in Figures 6.45 and 6.46.

```
1  <Model3DGroup x:Name="LowerLeg">
2
3    <GeometryModel3D Geometry="{StaticResource RSRCmeshFoot}"
4       Material=... />
5
6    <GeometryModel3D Geometry="{StaticResource RSRCmeshShin}"
7       Material=... >
8       <GeometryModel3D.Transform>
9          <TranslateTransform3D OffsetY="13"/>
10      </GeometryModel3D.Transform>
11   </GeometryModel3D>@</Model3DGroup>
```

Inline Exercise 6.12: Return to the lab, and use the hierarchy viewer/editor to add a transform to the shin to repair the lower-leg composite.

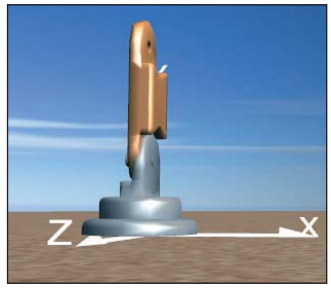


Figure 6.45: Rendering of the lower-leg model, now corrected via application of a modeling transformation on the shin subcomponent.

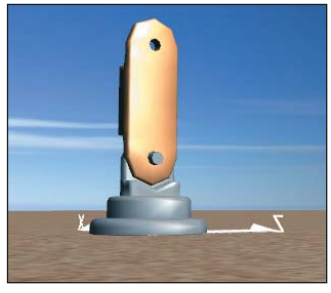


Figure 6.46: Rendering of the lower-leg model from a second point of view.

6.6.3.5 Creating the Full Leg

Let's continue our bottom-up implementation by going up to the next level. The "whole leg" is a composition of the lower leg (itself a composite) and the thigh (a primitive). Let's first compose these two components to form a rigid locked object, and then we'll attack the challenge of adding a knee joint.

As was the case for the lower leg, one of the subcomponents needs an instance transform (i.e., the thigh needs to be raised 43 units in the *y* direction) and the other is already at a suitable location. The resultant image is shown in Figure 6.47; the XAML code is as follows:

```
<Model3DGroup x:Name="Leg">

  <!- Build the lower-leg composite (same XAML shown earlier). ->
  <Model3DGroup x:Name="LowerLeg"> . . . </Model3DGroup>

  <!- Instantiate and transform the thigh. ->
  <GeometryModel3D Geometry="{StaticResource RSRCmeshThigh}"
      Material=. . . >
      <GeometryModel3D.Transform>
          <TranslateTransform3D OffsetY="43"/>
      </GeometryModel3D.Transform>
  </GeometryModel3D>

</Model3DGroup>
```

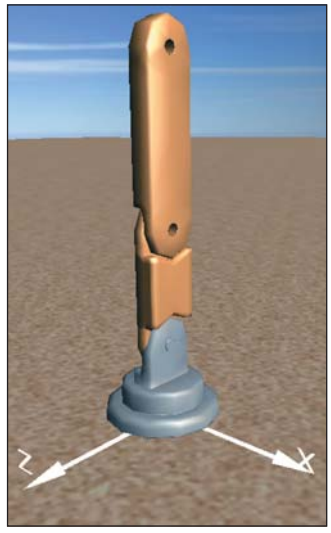


Figure 6.47: Rendering of the complete leg model.

Inline Exercise 6.13: Return to the lab, and select "Thigh" from the list of models to examine that component in isolation. Then select the model "Whole leg". The undesired merging of the two subcomponents will be obvious. Repair by adding an instance transform to the thigh to translate it on the *y*-axis. (If you wish, you can jump straight to our solution by choosing "Whole leg auto-composed" from the list of models.)

6.6.3.6 Adding the Knee Joint

The leg is currently locked in a straight position. But, by adding a rotation transformation to the lower leg, we can provide a "hook" that animation logic can use to simulate bending at the knee.

Figure 6.48 shows the leg in its canonical location at the origin, but with a 37° rotation at the knee. (The invisible axis of rotation has been added to this rendered image for clarity.)

The WPF element for expressing 3D rotation requires specification of the axis and the rotation amount. The axis is configured via two parameters: an arbitrary directional vector (e.g., $\begin{bmatrix}1 & 0 & 0\end{bmatrix}^{\mathbf{T}}$ representing any line that is parallel to the *x*-axis) and a center point lying on that vector (e.g., the center of the "knee" part of the lower leg, which is $(0, 50, 0)$ in the lower leg's coordinate system).

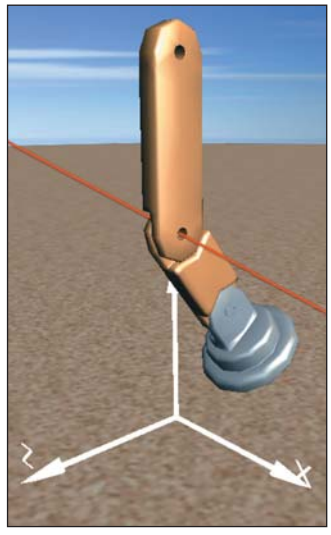


Figure 6.48: Result of specifying a 37° rotation at the knee joint, annotated with a red line through the joint, parallel to the x-axis, showing the axis of rotation.

Inline Exercise 6.14: Return to the whole-leg model in the lab. Implement the simulated knee joint by adding a rotation transform to the lower-leg component. Set the rotation axis as needed, and then use the numeric spinner to change the rotation amount to produce a knee-bend animation.

The XAML for the lower leg—with the joint transformation in effect to bend the knee—now appears as follows:

```
<!- Construct the lower-leg composite (same XAML shown earlier) ->
  <Model3DGroup x:Name="LowerLeg">
    <Model3DGroup.Transform>
      <!-- Joint transform for the knee -->
      <RotateTransform3D CenterX="0" CenterY="50" CenterZ="0">
        <RotateTransform3D.Rotation>
          <AxisAngleRotation3D x:Name="KneeJointAngle" Angle="37" Axis="1 0 0"/>
        </RotateTransform3D.Rotation>
      </RotateTransform3D>
    </Model3DGroup.Transform>
    <GeometryModel3D Geometry="{StaticResource RSRCmeshFoot}" Material=... />
    <GeometryModel3D Geometry="{StaticResource RSRCmeshShin}" Material=... >
      <GeometryModel3D.Transform>
        <TranslateTransform3D OffsetY="14"/>
      </GeometryModel3D.Transform>
    </GeometryModel3D>
  </Model3DGroup>
```

Manipulation of the joint's rotation amount can be performed by procedural code, or by the declarative animation features introduced in Section 2.5.1.

6.6.4 Reuse of Components

With our leg designed, let's move up one more level and consider how we might compose the entire camel. The leg is the first component that is going to be multiply instantiated into its parent. Reusing components is fundamental to hierarchical modeling; however, there are two types of reuse, each achieving different goals and appropriate for different scenarios.

First, consider a scenario in which we must show the camel walking or galloping in a realistic way. To achieve this, we need to be able to individually control the amount of rotation at each of the four hip joints and each of the four knee joints. To achieve hip and knee rotation, we construct the camel as a tree of nodes, with each node being used only once, as shown in Figure 6.49. Thus, there are four hip joints and four knee joints, each independently manipulable.

Is any "reuse" going on in this model? Well, you could say that we are reusing the *design* of the leg, since the model has four copies of the leg component hierarchy we designed above. But there is no reuse of the *actual components* in the constructed model. The front-left leg is a tree of components dedicated to representing only the front-left leg; there is a completely independent tree of components for each of the four legs. The advantage of this approach is that each leg is independent of the others, and the effect of manipulating one leg's joints is limited in scope to just that leg.

It is useful to search a model for rigid subcomponents that do not have any internal joints; they are candidates for being extracted and turned into reusable components. In the model discussed above, the lower-leg component is indeed rigid, because our design does not include an ankle joint that would allow the foot to rotate relative to the shin. Thus, we can reuse the lower-leg component hierarchy without any loss of animation flexibility, by constructing the camel using the structure shown in Figure 6.50. Here, our model is no longer a tree—it is a directed acyclic graph (DAG) now that component reuse is in effect; this topology is the reason this kind of structure is known as a "scene graph."

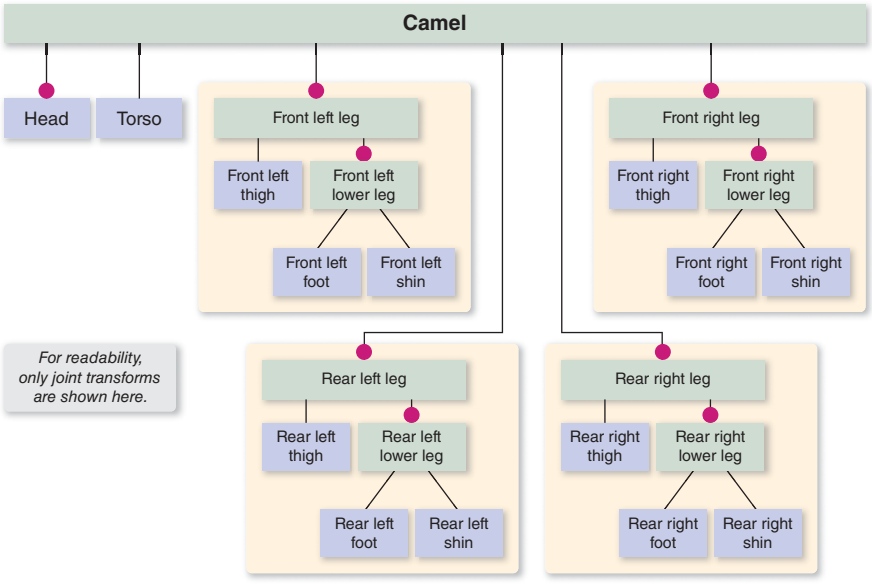


Figure 6.49: Scene graph of a camel constructed without reusable components, allowing individual control of each joint.

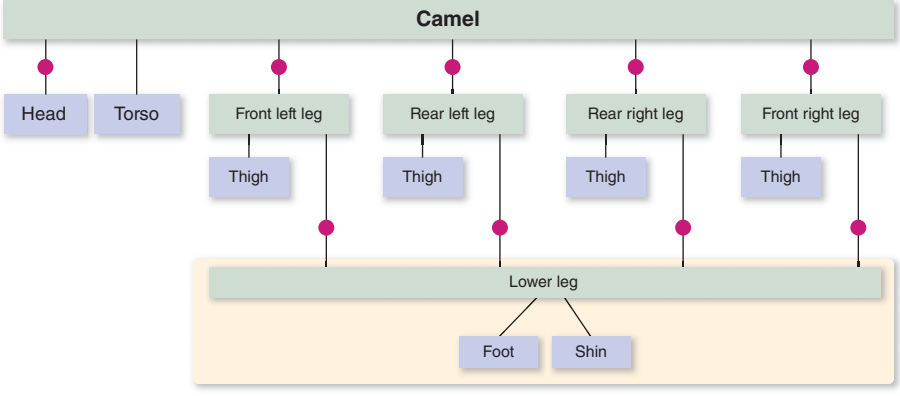


Figure 6.50: Reducing the storage cost by reusing a lower-leg submodel, with no loss of flexibility in joint control.

As the figure makes clear, each instantiation of the reusable `LowerLeg` has its own knee joint transform, so there is no loss in flexibility—each knee joint is still individually controllable. This DAG-based design is identical in functionality to the tree-based design above.

Thus far, we have been focused on supporting a high-fidelity animation of the camel's movement, but there other scenarios of interest. Consider a desert scene in which hundreds of camels, seen from afar, are crossing dunes in a caravan. The amount of processing needed to animate each hip and knee joint individually might be considered not worth the cost, especially if the caravan is so far from the viewpoint that such details would not be apparent to the viewer. In such a case, we might choose a lower-fidelity motion in which the camels move in unison,

with all left legs, both front and back, in the scene sharing a particular knee-bend animation, and all right legs sharing a different knee-bend animation.

A model using this approach is depicted in Figure 6.51. Here we have our first reusable components that include internal joint transforms: The reusable left leg has a left-knee joint, and the reusable right leg has its own right-knee joint. If we were to do a test instantiation of just one camel designed in this way, we would be able to control both left knees by manipulating just the one left-knee joint transform, and both right knees via the one right-knee joint transform.

But the processing advantage gained by reuse becomes much more apparent if you increase the number of camels to simulate the entire caravan. Consider Figure 6.52, which depicts a model of a caravan featuring reuse of the entire camel model.

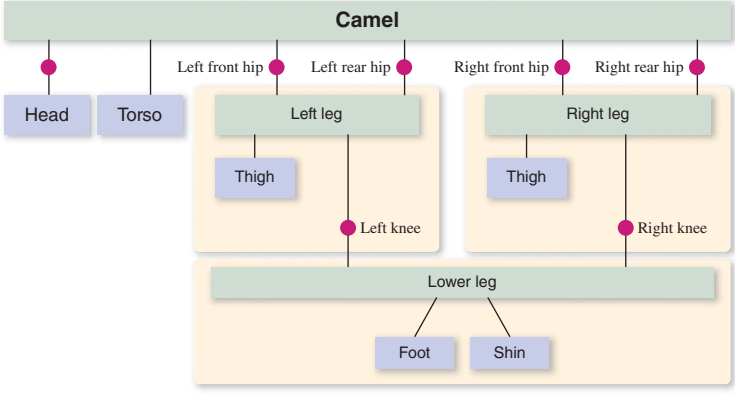


Figure 6.51: Reducing the storage cost by reusing a model for the left-side legs and a separate model for the right-side legs, with great loss of flexibility in joint control.

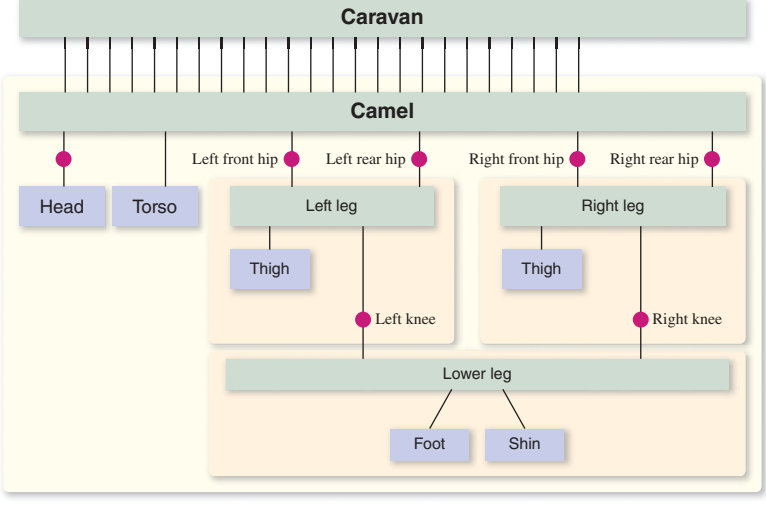


Figure 6.52: Modeling a caravan by reusing a single camel model, a highly scalable approach at the cost of excessive synchronized leg movement.

With this new scene graph, we have a caravan that is extremely scalable. No matter how many camels we place in this caravan, the number of joint transforms requiring manipulation for the animation of the legs is constant. By manipulating just four hip joints and two knee joints, the entire caravan's leg motion will be affected. Of course, this scalability comes at a cost: the uncanny and unnatural perfect synchronization of the entire caravan. At a distance, this type of low-fidelity caravan might be perfectly adequate, but if a bit more variety is desired, we could choose to have a handful of different limb animation sequences, create a distinct reusable camel "stencil" for each such sequence, and have each camel in the caravan be an instance of one of the stencils, chosen at random. Without much loss in scalability, we could thus achieve animation that is not quite so unnatural.

Of course, our caravan needs to be moving across the desert too; otherwise, the leg motions will look awfully silly. Thus, our animation logic must perform time-based manipulation of instance transforms on the camel objects simultaneously with the manipulation of the joint transforms.

Inline Exercise 6.15: How can we make the caravan's movement across the desert scalable? Is there a scene graph that would allow a single instance transform to move the entire caravan, without the loss of the scalable knee/hip control we just designed? What loss in realism would occur with such a plan?

If you are interested in knowing more about how to implement reusable components in XAML and WPF, consult the online materials for this chapter.

WPF is, of course, just one of many scene-graph platforms, and all implement reusability as part of their support for controlling scene complexity. For more information on scene-graph platforms, see Chapter 16.

6.7 Discussion

We have demonstrated the techniques common to most real-time fixed-function 3D platforms that are useful for displaying simple scenes composed of triangle-mesh objects, covered with solid or texture-mapped materials, and rendered using the classic Phong reflectance model with interpolated or flat shading.

Our focus on WPF as the example platform is designed to allow you to experiment and build prototype scenes using XAML, so you can exercise these techniques without the need to work with a procedural language and compile/build cycles.

It is important to note that a platform-resident scene graph is only applicable to projects for which "the picture is the thing." For most nontrivial applications, where the image is meant to be a visualization of some application data, there is an application model (database) storing both geometric and nongeometric information, acting as the source from which a scene graph is derived for display purposes. This topic is discussed more extensively in Chapter 16.

This page intentionally left blank

Chapter 7

Essential Mathematics and the Geometry of 2-Space and 3-Space

7.1 Introduction

Unlike other chapters in this book, this one covers a great deal of material with which you may already be familiar in some form. The goals of this organization of the material are

- To have it all in one place, where you can easily refer to it
- To present it in a way that has a somewhat different approach from what you may have seen in the past, one that's particularly useful for graphics

Much of the chapter will be easy reading; the material will look familiar. To be sure it really is familiar, we include a number of exercises in the chapter itself; you should work through these exercises to be certain you really are understanding what you're reading. We assume that you've encountered some linear algebra, and are familiar with vectors, matrices, linear transformations, and notions like "basis" and "linear independence."

One test, as you read this material, is to ask yourself, "Could I write code to implement this idea?" If the answer is no, you should spend more time understanding the concept. This is sufficiently important that we embody it in a principle, which one of us first heard expressed by Hale Trotter of Princeton in the 1970s.

✓ **THE IMPLEMENTATION PRINCIPLE:** If you understand a mathematical process well enough, you can write a program that executes it.

We add to this the idea that often, writing such a program removes the necessity of understanding the details in the future: If you've done a good job, you can reuse your program.

7.2 Notation

We'll use conventional mathematical notation, in which most variables appear in italics; vectors will be written in roman boldface (e.g., $\mathbf{u}$), as will matrices. In general, vectors will be lowercase, matrices uppercase. When a variable has a subscript used for indexing, it's italic, as in the i in $\sum_i x_i$. When a subscript or superscript is mnemomic, as in ρ_{dh}, the "directional hemispherical reflectance," it's in roman font.

Certain special sets have predefined names and are written in boldface font: $\mathbf{R}$ is the set of real numbers; $\mathbf{C}$ is the set of complex numbers; $\mathbf{R}^+$ (pronounced "R plus") is the set of positive real numbers; and $\mathbf{R}_0^+$ (pronounced "R plus zero") is the set of non-negative reals.

7.3 Sets

Sets are generally denoted by capital letters. The *Cartesian product* of the sets B and C is the set

$$B \times C = \{(b, c) : b \in B, c \in C\}^1, \tag{7.1}$$

which is pronounced "B cross C"; despite this, it is called a "Cartesian" product rather than a "cross product"; that term is reserved for the cross product of vectors discussed in Section 7.6.4.

The product $\mathbf{R} \times \mathbf{R}$ is denoted $\mathbf{R^2}$; higher order products are $\mathbf{R^3}$, $\mathbf{R^4}$, etc., with the n-fold product being $\mathbf{R}^n$.

The **closed interval** $[a, b]$ is the set of all real numbers between a and b, inclusive, that is,

$$[a, b] = \{x : a \leq x \leq b\}. \tag{7.2}$$

If $b < a$, then the interval is empty; if $b = a$, the interval contains just the number b. We'll also occasionally use intervals that contain just one of their endpoints (i.e., **half-open intervals**):

$$[a, b) = \{x : a \leq x < b\}, \tag{7.3}$$
$$(a, b] = \{x : a < x \leq b\}. \tag{7.4}$$

We also define the following two notational conventions:

$$[a, \infty) = \{x : a \leq x\}, \tag{7.5}$$
$$(-\infty, b] = \{x : x \leq b\}. \tag{7.6}$$

1. This notation means "the set of all pairs (b, c) such that b is in B and c is in C." That is, the colon is read "such that."

7.4 Functions

The notion of a *function* is already familiar to you from both mathematics and programming. We'll use a particular notation to express functions; an example is

$$f : \mathbf{R} \to \mathbf{R} : x \mapsto x^2. \tag{7.7}$$

The name of the function is f. Following the colon are two sets. The one to the left of the arrow is called the **domain;** the one to the right is called the **codomain** (some books use the term "range" for this; however, "range" is also used in a similar but different sense, leading to confusion). Following the second colon is a description of the rule for associating to an element of the domain, x, an element of the codomain.

This corresponds closely to the definition of a function in many programming languages, which tends to look like this:

```
1 double f(double x)
2 {
3    return x * x;
4 }
```

Once again, the function is named; the domain is explicitly defined ("x can be any double"), and the codomain is explicitly defined ("this function produces doubles"). The rule for associating the typical domain element, x, with the resultant value is given in the body of the function.

Mathematics allows somewhat subtler definitions than do most programming languages. For example, we can define

$$g : \mathbf{R} \to \mathbf{R}_0^+ : x \mapsto x^2 \tag{7.8}$$

in mathematics, but most languages lack a data type which is a "non-negative real number." The distinction between f and g is important, however: In the case of g the set of values produced by the function (i.e., the set $\{x^2 : x \in \mathbf{R}\}$) turns out to be the entire codomain, while in f it is a proper subset of the codomain. The function g is said to be **surjective,** while f is not. (Some books say that "g is onto.").

If we define

$$h : \mathbf{R}_0^+ \to \mathbf{R}_0^+ : x \mapsto x^2, \tag{7.9}$$

we get yet a different function. The function h is not only surjective, it has another property: No two elements of the domain correspond to the same element of the codomain; that is, if $h(a) = h(b)$, then a and b must be equal. Such a function is called **injective.**[2] A function like h that's both injective and surjective has an **inverse,** denoted h^{-1}, a function that "undoes" what h does. The domain of h^{-1} is the codomain of h, and vice versa. In the case of our particular function, the inverse is

$$h^{-1} : \mathbf{R}_0^+ \to \mathbf{R}_0^+ : x \mapsto \sqrt{x}. \tag{7.10}$$

2. Some books use the term "one-one" or "one-to-one," but others use the same term to mean both injective and surjective.

More generally, if

$$f : C \to D \tag{7.11}$$

is an injective and surjective (or **bijective**) function, its inverse,

$$f^{-1} : D \to C, \tag{7.12}$$

is the unique function satisfying

$$f^{-1}(f(x)) = x, \qquad \text{for all } x \in D, \text{ and} \tag{7.13}$$

$$f(f^{-1}(y)) = y, \qquad \text{for all } y \in C. \tag{7.14}$$

Figure 7.1 illustrates these three classes of functions.

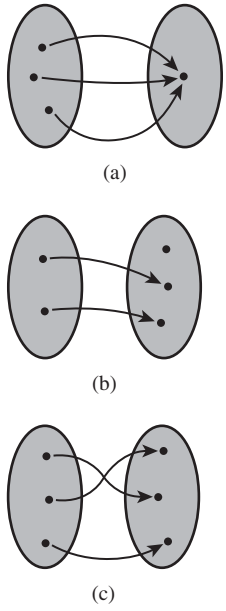


Figure 7.1: Three different functions: (a) is surjective but not injective; (b) is injective but not surjective; and (c) is bijective.

Inline Exercise 7.1: Which of the following functions have inverses? Describe the inverses when they exist.
(a) The negation function $N : \mathbf{R} \to \mathbf{R} : x \mapsto -x$.
(b) $q_1 : \mathbf{R} \to \mathbf{R} : x \mapsto \arctan(x)$.
(c) $q_2 : \mathbf{R} \to [-\pi/2, \pi/2] : x \mapsto \arctan(x)$.

In describing functions, we'll always describe the domain, the codomain, and the rule that associates elements in the first to elements in the second. Sometimes these rules may involve cases, as in

$$u : \mathbf{R} \to \mathbf{R} : x \mapsto \begin{cases} 1 & -1 \le x \le 1 \\ 0 & \text{otherwise} \end{cases}, \tag{7.15}$$

just as the code for a function may involve an `if` statement. We will also always speak of functions by name (e.g., "the function f is continuous") rather than saying "the function $f(x)$ is continuous," because $f(x)$ denotes the *value* of the function at a point x; this value is usually *not* a function. If we need to include the variable name for some reason, we will write "the function $x \mapsto f(x)$ is continuous at $x = 0$, but not elsewhere," for instance. This careful distinction becomes important when we discuss functions like the Fourier transform, $\mathcal{F}$, which operate on *functions,* producing other functions. If we speak of "the function $f \mapsto \mathcal{F}(f)$," we are referring to the Fourier transform $\mathcal{F}$; if we speak of "the function $\mathcal{F}(f)$," we are referring to its value on a particular function f.

7.4.1 Inverse Tangent Functions

Mathematicians tend to define **arctan,** the inverse tangent function, from $\mathbf{R}$ to the open interval $(-\pi/2, \pi/2)$. We'll sometimes use this, and denote it $u \mapsto \tan^{-1}(u)$. A frequent use of the inverse tangent is to find the angle, θ, in the situation shown in Figure 7.2: We have the point with coordinates (x, y) and want to know θ. The usual answer is that when $x > 0$, it's $\tan^{-1}(y/x)$, followed by several special cases for when $x < 0, y > 0$, or $x < 0, y = 0$, etc. These special cases have been built into a single function, atan2, which takes a *pair* of arguments, rather than a single one. It's almost always used in the form $\theta = \text{atan2}(y, x)$, which does exactly what you would expect: It returns the angle between the x-axis and the ray from $(0, 0)$ to (x, y). The returned angle is between $-\pi$ and π. In the case where x

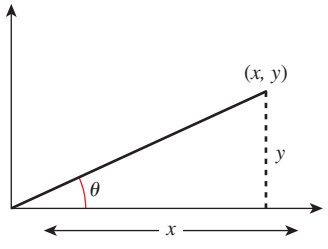


Figure 7.2: How is θ related to x and y?

and y are both zero, the returned value is zero. This makes atan2 *discontinuous at the origin,* as well as along the negative x-axis. On the negative x-axis, the IEEE version of atan2(y, x) returns either $+\pi$ or $-\pi$ depending on whether $y = +0$ or -0. The only tricky part is remembering that y comes first. You can remember this by knowing that if $-\pi < \theta < \pi$, then

$$\text{atan2}(\sin\theta, \cos\theta) = \theta. \tag{7.16}$$

We'll use atan2 not only in programs, but also in equations.

7.5 Coordinates

The Cartesian plane shown in Figure 7.3 is a model for Euclidean geometry: All of the axioms of geometry hold in the Cartesian plane, and we can use our geometric intuition to reason about it. A tabletop (or to be more accurate, an infinite tabletop) is also a model for Euclidean geometry. The difference between the two is that each point of the Cartesian plane has a pair of real numbers—its **coordinates**—associated to it. This lets us transform geometric statements like "the point P lies on the lines ℓ_1 and ℓ_2" to algebraic statements, like "the coordinates of the point P satisfy these two linear equations." We can, of course, draw two perpendicular lines on the infinite tabletop, declare them to be the x- and y-axes, place equispaced tick marks along each, and use perpendicular projection onto these lines to define coordinates. But the choices we made—which line to call the x-axis, which to call the y-axis, what point to use as the origin, etc.—were arbitrary.[3] It's important to distinguish between the properties of a point or a line, and the properties of its coordinates; the underlying geometric properties don't change when we change coordinate systems, while numerical properties of the coordinates *do* change. In Figure 7.4, you can see that the point P is on the line ℓ—that's a *geometric* property; it's true independent of any coordinate system.

As an example of coordinate-dependent properties, the coordinates of P in the black coordinate system are $(3, 5)$, while in the blue coordinate system they are $(2, 2)$. Similarly, the equation of the line ℓ in the black coordinate system is $y = 5$, while in the blue coordinate system it's $x + y = 4$. Thus, the point's coordinates and the line's equation are coordinate-dependent. But the fact that the point is on the line is coordinate-independent: Although P's coordinates and ℓ's equation are different in the two systems, the black coordinates of P satisfy the black equation for ℓ, and similarly for the blue.

From now on when we speak of "the coordinates of a point," it will always be with respect to some coordinate system; much of the time the coordinate system will be obvious and we won't mention it. For example, in $\mathbf{R}^2$, the set of ordered pairs of real numbers, the "standard" coordinates of the point (x, y) are just x and y.

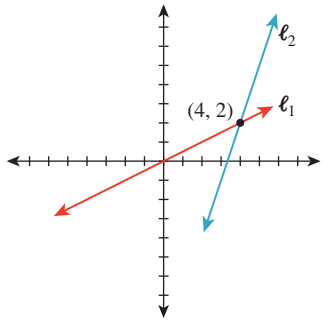


Figure 7.3: The Cartesian plane, in which points are specified by x- and y-coordinates.

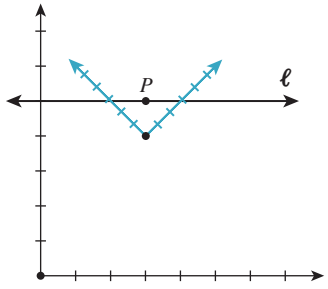


Figure 7.4: The Cartesian plane with multiple coordinate systems.

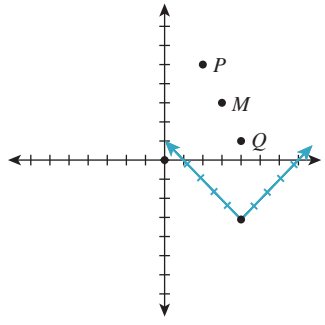


Figure 7.5: The coordinates of M in each coordinate system are the average of the coordinates of P and Q in that coordinate system; thus, the geometric *operation of finding the midpoint of a segment corresponds to the* algebraic *operation of averaging coordinates,* independent of what coordinate system we use.

7.6 Operations on Coordinates

Suppose (see Figure 7.5) we have the points $P = (2, 5)$ and $Q = (4, 1)$ in the plane, where the coordinates are with respect to the coordinate system drawn in horizontal and vertical black lines. If we *average* the coordinates of these points

3. Indeed, Descartes did not even require that the two axes be perpendicular, although we now always choose them so.

(averaging the x-coordinates and then averaging the y-coordinates) we get $M = (3, 3)$, which turns out to be the midpoint of the segment between them. This is an interesting situation: The midpoint is defined purely geometrically, independent of coordinates. But we've got a formula, in coordinates, for computing it. Suppose we look at the coordinates of P and Q in the blue coordinate system, which is rotated 45° from the black one, and has its origin to the right and below, and average those: The coordinates are (2, 2) and (2, 4); the average is (2, 3). In the blue coordinate system, the point $(2, 3)$ is exactly at the same location as the black coordinate system point $(3, 3)$. In short, while the coordinate computations differ, the underlying geometric result is the same.

Contrast this with the operation "divide the coordinates of a point by two" (Figure 7.6). Under this operation, the point P with black-line coordinates $(2, 5)$ becomes the point P' with coordinates $(1, 2.5)$. But if we apply the same operation in blue-line coordinates, where P has coordinates $(4, 7)$, the new blue-line coordinates are $(2, 3.5)$, and the point P'' corresponding to those coordinates is far from P'. What's the difference between the averaging and the divide-by-two operations? Why does "averaging coordinates" give the same result in any two coordinate systems, while "dividing coordinates by two" gives different ones? We'll answer this in greater detail in Section 7.6.4. For now, let's just examine the distinction algebraically: Let's write down the average of the coordinates of points (x_1, y_1) and (x_2, y_2). It's just

$$M = \left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2} \right). \tag{7.17}$$

If we agree to temporarily define a "multiplication" of a point's coordinates by a number with the rule

$$s(x, y) = (sx, sy), \tag{7.18}$$

and addition of points by

$$(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2), \tag{7.19}$$

then this averaging can be written

$$M = \frac{1}{2}(x_1, y_1) + \frac{1}{2}(x_2, y_2), \tag{7.20}$$

while the "dividing coordinates by two" operation can be written

$$\frac{1}{2}(x, y). \tag{7.21}$$

The key difference, it turns out, is that the first operation involves summing up terms where the coefficients sum to one (because $\frac{1}{2} + \frac{1}{2} = 1$), while the second does not. A combination where the coefficients sum to one is called an **affine combination** of the points, and it is combinations like these that are invariant when we change coordinate systems. (You should try a few others to convince yourself of this.)

Since affine combinations have the property of being "geometrically meaningful," we'll now examine them more closely. Suppose that instead of averaging, we took a $\frac{1}{3} - \frac{2}{3}$ combination of the points, that is, we computed (for the points P and Q in Figure 7.5)

$$\frac{1}{3}P + \frac{2}{3}Q. \tag{7.22}$$

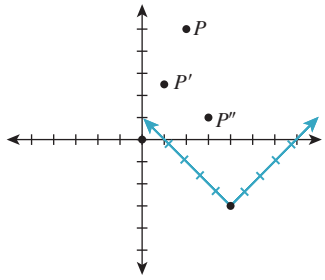


Figure 7.6: The operation "divide the coordinates of a point by two" produces different results (P' and P'') in the two coordinate systems—this simple algebraic operation is not *independent of the coordinate system, so it doesn't correspond to any geometric operation.*

We'd get the point $(\frac{10}{3}, \frac{7}{3})$, which also lies on the line between P and Q, but is closer to Q. In fact, we can compute

$$(1-\alpha)P + \alpha Q \tag{7.23}$$

for any number α: With $\alpha = 1$ we get Q; with $\alpha = 0$ we get P; with $\alpha = \frac{1}{2}$ we get M; and with any value of α between 0 and 1 we get points on the line segment between P and Q.

What happens when we consider values of α that are less than 0? Those correspond to points on the line beyond P; similarly, ones with $\alpha > 1$ are beyond Q. In summary:

> As α ranges over the real numbers, the points $(1-\alpha)P + \alpha Q$ range over the line containing P and Q, with $\alpha = 1$ corresponding to P and $\alpha = 0$ corresponding to Q, and values of α between 0 and 1 corresponding to points *between* P and Q.

With this in mind, we can define a function

$$\gamma : \mathbf{R} \to \mathbf{R^2} : t \mapsto (1-t)P + tQ. \tag{7.24}$$

The image of this function is the line between P and Q; if we restrict the domain to the interval $[0, 1]$, then the image is the line *segment* between P and Q. We call this the **parametric form of the line between P and Q**, where the argument t is the **parameter.** (In Section 7.6.4 we'll justify this particular use of the multiply-by-scalars-and-add operation applied to points.)

> **Inline Exercise 7.2:** We discussed that certain coordinate constructions are invariant under changes in coordinate systems. If two people place coordinate systems on the same tabletop and compute lengths, angles, and areas, will they always get the same results? In other words, are lengths, angles, and areas invariant under changes of coordinates? If not, can you think of particular conditions on the coordinate systems under which these *are* invariant? Note: The *length* of the segment from (x_1, y_1) to (x_2, y_2) in a Cartesian coordinate system is defined to be $\sqrt{(x_2-x_1)^2 + (y_2-y_1)^2}$; you'll need to figure out similar definitions of angle and area to answer this question.

7.6.1 Vectors

We'll return to lines presently; before we do so, however, we'll codify some of the ideas above by relating them to vectors. The term "vector" gets used in many fields to mean many things. For now, we'll content ourselves with one particular type of vector, a **coordinate vector,** which is simply a list of real numbers. An n-vector is a list of n numbers; we'll write these between square brackets, organized vertically:

$$\begin{bmatrix} 1 \\ -4 \\ 0 \end{bmatrix}, \tag{7.25}$$

for example, is a 3-vector. A generalization of this notion is that of **matrices,** which are doubly indexed lists of real numbers, named by the number of rows and the number of columns. Thus,

$$\begin{bmatrix} 1 & 2 \\ -4 & 0 \\ 0 & 6 \end{bmatrix} \tag{7.26}$$

is a 3 by 2 (often written 3×2) matrix. Entries of the matrix are indicated by subscripts, with the row index first; if A is a matrix, then a_{ij} denotes the entry in row i, column j. An n-vector can be considered an $n \times 1$ matrix. An important operation on matrices is **transposition:** An $n \times k$ matrix $\mathbf{A}$ becomes a $k \times n$ matrix whose ij entry is the ji entry of $\mathbf{A}$. Thus, the transpose of the matrix above is

$$\begin{bmatrix} 1 & -4 & 0 \\ 2 & 0 & 6 \end{bmatrix}. \tag{7.27}$$

The transpose of the matrix $\mathbf{A}$ is written $\mathbf{A}^{\mathbf{T}}$. Because horizontal lists fit typography better than do vertical ones, we'll often describe a vector by its transpose. Thus, if $\mathbf{v}$ is the vector above, we might write "Let $\mathbf{v} = \begin{bmatrix} 1 & -4 & 0 \end{bmatrix}^{\mathbf{T}} \ldots$" as a way of introducing it into the discussion.

7.6.1.1 Indexing Vectors and Arrays

In mathematics, vectors and matrices use one-based indexing: If $\mathbf{v}$ is a vector, its first element is written v_1, its second v_2, etc. If $\mathbf{M}$ is a matrix, the element in row i and column j is denoted m_{ij}; when i and j are particular integers, these are sometimes separated by commas, as in $m_{1,2}$.

7.6.1.2 Certain Special Vectors

In $\mathbf{R^2}$, any vector $\begin{bmatrix} a & b \end{bmatrix}^{\mathbf{T}}$ can be expressed in the form

$$\begin{bmatrix} a \\ b \end{bmatrix} = a \begin{bmatrix} 1 \\ 0 \end{bmatrix} + b \begin{bmatrix} 0 \\ 1 \end{bmatrix}; \tag{7.28}$$

the two vectors on the right are called $\mathbf{e}_1$ and $\mathbf{e}_2$. In $\mathbf{R^3}$, we have a similar set of vectors; the names are reused:

$$\mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \qquad \mathbf{e}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \text{and} \qquad \mathbf{e}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}. \tag{7.29}$$

In general, in $\mathbf{R}^n$, the name $\mathbf{e}_i$ refers to a vector whose entries are zeroes, except the ith, which is 1.

The vector whose entries are all zero (in any dimension) is denoted $\mathbf{0}$; note the boldface font.

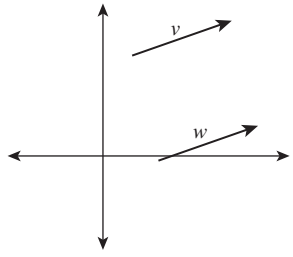


Figure 7.7: The arrows labeled $\mathbf{v}$ *and* $\mathbf{w}$ *are often considered to be "the same," even though they are clearly different entities. If we think of each of them as representing a displacement of the plane (i.e., a motion of all points of the plane up and to the right), then although the arrows themselves are distinct, they represent the same displacement.*

7.6.2 How to Think About Vectors

It's common for students to say, "A vector is an arrow." So, when asked if the vectors $\mathbf{v}$ and $\mathbf{w}$ in Figure 7.7 are the same, they answer "yes," although the two arrows, being in different places, are clearly different. A better way to think of a

vector is as a **displacement**—it represents an amount by which you must move to get from one place to another. For example, to get from the point $(3, 1)$ to the point $(5, 0)$, you must move by 2 in the x-direction and by -1 in the y-direction. This displacement is represented by the vector $\begin{bmatrix} 2 & -1 \end{bmatrix}^{\mathbf{T}}$. It's exactly the same as the displacement needed to move from $(4, 1)$ to $(6, 0)$. With this interpretation, addition of vectors makes sense: You add corresponding terms. And multiplication by a constant is similarly defined by multiplying each entry by that constant, thus increasing or decreasing the displacement.

If the word "displacement" is not satisfactory, you can also think of vectors as "differences between points," that is, as a description of the amount you would have to move the first point to reach the second. The same pair of points, moved to a different location by a shift in x and y, correspond to the same vector, because the difference between them is unchanged.

7.6.3 Length of a Vector

The length (or **norm**) of the vector $\mathbf{v}$, denoted $\|\mathbf{v}\|$, is the square root of the sum of the squares of the entries of $\mathbf{v}$. If $\mathbf{v} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}^{\mathbf{T}}$, then $\|\mathbf{v}\| = \sqrt{1^2 + 2^2 + 3^2} = \sqrt{14}$. This corresponds, when we think of $\mathbf{v}$ as a displacement, to the distance that we moved. A vector whose length is 1 is called a **unit vector.**

You can convert a nonzero vector $\mathbf{v}$ to a unit vector, which is called **normalizing** it, by dividing it by its length. We write

$$\mathcal{S}(\mathbf{v}) = \mathbf{v}/\|\mathbf{v}\| \tag{7.30}$$

for this, with the letter "S" being chosen for "sphere," since normalizing a vector in 3-space amounts to adjusting its length so that its tip lies on the unit sphere.

7.6.4 Vector Operations

We can add vectors and multiply a vector by a constant (called **scalar multiplication**); more generally, if we have several vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$, and numbers $c_1, c_2, \ldots, c_n$, we can form the **linear combination**

$$c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \ldots + c_n\mathbf{v}_n. \tag{7.31}$$

The set of all linear combinations of a single nonzero vector $\mathbf{v}$ is the line containing $\mathbf{v}$ (where here we are reverting temporarily to the notion of the vector $\mathbf{v}$ representing the endpoint of an arrow starting at the origin); the set of all linear combinations of two nonzero vectors $\mathbf{v}$ and $\mathbf{w}$ is, in general, the plane that contains both of them. One exception is when one vector is a multiple of the other, in which case the result is the *line* containing both.

Aside from addition and multiplication by a constant, there are two other operations on vectors that we'll often use: dot product and cross product.

7.6.4.1 Cross Product

The cross product is usually defined for pairs of vectors in 3-space as follows:

$$\begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \times \begin{bmatrix} w_x \\ w_y \\ w_z \end{bmatrix} = \begin{bmatrix} v_y w_z - v_z w_y \\ v_z w_x - v_x w_z \\ v_x w_y - v_y w_x \end{bmatrix}. \tag{7.32}$$

The cross product is *anticommutative,* that is, $\mathbf{v} \times \mathbf{w} = -\mathbf{w} \times \mathbf{v}$ (the labeling and ordering of the subscripts was chosen to make this self-evident). It's distributive over addition and scalar multiplication, but not associative. One of the main uses of the cross product is that

$$\|\mathbf{v} \times \mathbf{w}\| = \|\mathbf{v}\|\|\mathbf{w}\| |\sin\theta| \tag{7.33}$$

where θ is the angle between $\mathbf{v}$ and $\mathbf{w}$. That means that half the length of the cross product is the area of the triangle with vertices $(0,0,0)$, (v_x, v_y, v_z), and (w_x, w_y, w_z).

The cross product can be generalized to dimension n; in dimension n, it's a product of $n-1$ vectors (which explains why the $n = 3$ case is the most often used). Aside from dimension 3, our most frequent use will be in dimension 2, where it's a "product" of one vector. The cross product of the vector is

$$\times \begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} -v_y \\ v_x \end{bmatrix}. \tag{7.34}$$

This cross product has an important property: Going from $\mathbf{v}$ to $\times\mathbf{v}$ involves a rotation by 90° *in the same direction as the rotation that takes the positive x-axis to the positive y-axis*. Because of this, it's sometimes also denoted by $\mathbf{v}^{\perp}$.

In the same way, going from $\mathbf{v}$ to $\mathbf{w}$ to $\mathbf{v} \times \mathbf{w}$ describes a **right-handed coordinate system,** one in which placing your right-hand pinkie on the first vector and curling it toward the second makes your thumb point in the direction of the third (see Figure 7.8). In general, in n dimensions, the cross product $\mathbf{z}$ of $n-1$ vectors $\mathbf{v}_1, \ldots, \mathbf{v}_{n-1}$ lies in a line perpendicular to the subspace containing $\mathbf{v}_1, \ldots, \mathbf{v}_{n-1}$. The length of $\mathbf{z}$ is $(n-1)!$ times the $(n-1)$-dimensional volume of the pyramid-like shape whose vertices are the origin and the endpoints of $\mathbf{v}_i$. Assuming this volume is nonzero, $\mathbf{z}$ is oriented so that $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_{n-1}, \mathbf{z}$ is "positively oriented" in analogy with the right-hand rule in three dimensions.

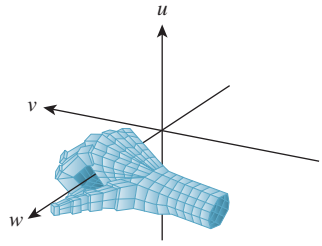


Figure 7.8: The uvw directions form a right-handed coordinate system.

7.6.4.2 Dot Product

From linear algebra, you're familiar with the **dot product** of two n-vectors $\mathbf{v}$ and $\mathbf{w}$, defined by

$$\mathbf{v} \cdot \mathbf{w} = v_1 w_1 + v_2 w_2 + \ldots + v_n w_n. \tag{7.35}$$

This is sometimes denoted $\langle \mathbf{v}, \mathbf{w} \rangle$; in this form, it's usually called the **inner product.** The dot product is used for measuring angles. If $\mathbf{v}$ and $\mathbf{w}$ are unit vectors, then

$$\mathbf{v} \cdot \mathbf{w} = \cos(\theta), \tag{7.36}$$

where θ is the angle between the vectors (Figure 7.9). This is most often used in the form

$$\theta = \cos^{-1} \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|} \tag{7.37}$$

which gives the angle between any two nonzero vectors, expressed as a number between 0 and π, inclusive.

Fix the vector $\mathbf{w} \in \mathbf{R^2}$ for a moment. The function

$$\phi_{\mathbf{w}} : \mathbf{R^2} \to \mathbf{R} : \mathbf{v} \mapsto \mathbf{w} \cdot \mathbf{v} \tag{7.38}$$

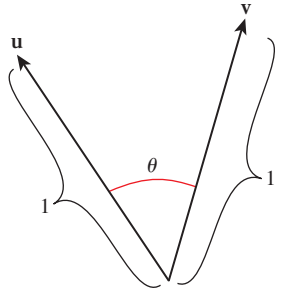


Figure 7.9: The dot product of unit vectors gives the cosine of the angle θ between them.

tells "how much **v** is like **w**" in the sense that as **v** ranges over all displacements of a fixed length—say, length 1—this function is most positive when **v** is parallel to **w**, most negative when it's a displacement in the direction opposite **w**, and zero for displacements that are in directions perpendicular to **w**.

The dot product is central to much of linear algebra; a surprising number of computations and simplifications can be done by working with vectors and their dot products rather than with their coordinates. This often leads to insight into the underlying operations.

7.6.4.3 The Projection of w on v

As an example of the use of dot products, suppose we want to write the displacement **w** as a sum,

$$\mathbf{w} = \mathbf{v}' + \mathbf{u}, \tag{7.39}$$

where $\mathbf{v}'$ is parallel to **v** and **u** is perpendicular to it (see Figure 7.10). How can we find $\mathbf{v}'$? First, we know it's a multiple $s\mathbf{v}$ of **v**; all we need to find is s. Consider the dot product of **v** with both sides of Equation 7.39:

$$\begin{aligned} \mathbf{v}\cdot\mathbf{w} &= \mathbf{v}\cdot\mathbf{v}' + \mathbf{v}\cdot\mathbf{u} && (7.40)\\ &= \mathbf{v}\cdot(s\mathbf{v}) + 0 && (7.41)\\ &= s(\mathbf{v}\cdot\mathbf{v}),\ \text{so} && (7.42)\\ s &= \frac{\mathbf{v}\cdot\mathbf{w}}{\mathbf{v}\cdot\mathbf{v}}. && (7.43) \end{aligned}$$

The projection is therefore

$$\mathbf{v}' = \frac{\mathbf{v}\cdot\mathbf{w}}{\mathbf{v}\cdot\mathbf{v}}\mathbf{v}. \tag{7.44}$$

And **u** is just

$$\mathbf{u} = \mathbf{w} - \mathbf{v}'. \tag{7.45}$$

When **v** is a unit vector, the expression for $\mathbf{v}'$ simplifies to

$$\mathbf{v}' = (\mathbf{v}\cdot\mathbf{w})\mathbf{v}. \tag{7.46}$$

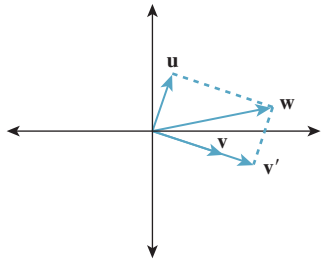


Figure 7.10: Decomposing a displacement **w** *as a sum of two displacements, one parallel to a given vector* **v** *and one perpendicular to* **v**.

7.6.4.4 Operations on Points and Vectors

With the notion of a vector as a difference between points, or as a displacement, we can write down a number of operations.

- The *difference* between points P and Q, denoted by $P - Q$, is a vector.
- The *sum* of a point P and a vector **v** is another point. In particular, $P + (Q - P) = Q$.
- The *sum* or *difference* of vectors is defined by elementwise addition or subtraction.
- The *sum* of two points isn't defined at all.

Thus, although for both points and vectors in the plane we represent each with a pair of real numbers, we use the typographical convention that points are denoted

by tuples like $(3, 6)$, while vectors are denoted by 2×1 matrices in square brackets. In *software* (see Chapter 12), it's helpful to also make vectors and points have different *types*. In an object-oriented language, we would define the classes `Point` and `Vector`. The first would contain an `AddToVector` operation, but not an `AddToPoint` operation; the second would have both (with results of types `Vector` and `Point`, respectively). Such a distinction allows the compiler to save us from mistakes in our mathematical treatment of points and vectors.

If we, for a moment, use E^2 to denote the set of points, but $\mathbf{R}^2$ to denote vectors, then we have defined

$$\text{difference} : E^2 \times E^2 \to \mathbf{R}^2 : (P, Q) \mapsto P - Q, \text{ and} \tag{7.47}$$

$$\text{sum} : E^2 \times \mathbf{R}^2 \to E^2 : (P, \mathbf{v}) \mapsto P + \mathbf{v}. \tag{7.48}$$

(Note that $E^2 \times E^2$ is the Cartesian product of E^2 with itself, i.e., the set of all pairs of points.) These definitions generalize in a natural way to $\mathbf{R}^n$ and E^n. In general, using these operations, we can define an **affine combination of points.** Even though we said we could not add points, we saw earlier that computing the midpoint between P and Q could be expressed nicely if we allowed ourselves to write things like

$$\frac{1}{2}P + \frac{1}{2}Q. \tag{7.49}$$

We'll now make sense of that. The expression

$$\alpha P + \beta Q, \tag{7.50}$$

which we call an **affine combination** of P and Q, will be defined only when $\alpha + \beta = 1$. If we pretend for a moment that all sorts of arithmetic on points is well defined, then we can add βP and subtract it, to get

$$\alpha P + \beta Q = \alpha P + \beta P + \beta Q - \beta P \tag{7.51}$$

$$= (\alpha + \beta)P + \beta(Q - P) \tag{7.52}$$

$$= P + \beta(Q - P). \tag{7.53}$$

With this cavalier bit of algebra as motivation, we *define* $\alpha P + \beta Q$ to mean

$$P + \beta(Q - P), \tag{7.54}$$

which makes sense because $Q - P$ is a vector, and hence $\beta(Q - P)$ is as well; this vector can be added to the point P to get a new point. This definition naturally generalizes to an affine combination of more than two points; for instance, $\alpha P + \beta Q + \gamma R$, where $\alpha + \beta + \gamma = 1$,

$$P + \beta(Q - P) + \gamma(R - P). \tag{7.55}$$

We'll frequently encounter such affine combinations of points, especially when we discuss splines in Chapter 22. Mann et al. [MLD97] make a case for treating all of graphics in terms of such "affine geometry," and abandoning coordinates to the degree possible. In fact, it makes sense to include an `AffineCombination` method for `Points`, to avoid errors or code that can be baffling. Listing 7.1 shows a possible implementation.

Listing 7.1: Code in C# for creating an affine combination of points, with a special case for two points.

```
public Point AffineCombination(double[] weights, Point[] Points)
{
  Debug.assert(weights.length == Points.length);
  Debug.assert(sum of weights == 1.0);
  Debug.assert(weights.length > 0);

  Point Q = Points[0];
  for (int i = 1; i < weights.length; i++) {
    Q = Q + weights[i]* (Points[i] - Points[0]);
  }
  return Q;
}

public Point AffineCombination(Point P, double wP,
                               Point Q, double wQ)
{
  Debug.assert( (wP + wQ) == 1.0);

  Point R = P;
  R = R + wQ * (Q - P);
  return R;
}
```

7.6.5 Matrix Multiplication

The product of two matrices **A** and **B** is defined only when the number of columns of **A** matches the number of rows of **B.** If **A** is $n \times k$ and **B** is $k \times p$, then the product **AB** is an $n \times p$ matrix. If we let the ith row of **A** be the transpose of the vector $\mathbf{r}_i$, and the jth column of **B** be the vector $\mathbf{c}_i$, then the ijth entry of **AB** is just $\mathbf{r}_i \cdot \mathbf{c}_j$. The following picture may help you remember this:

$$\begin{array}{c} \\ \\ \cdot \end{array} \overbrace{\begin{bmatrix} & \begin{pmatrix} \cdot \\ \cdot \\ \cdot \end{pmatrix} \mathbf{c}_j & \end{bmatrix}}^{p} \Bigg\} k$$

$$n \left\{ \underbrace{\begin{bmatrix} \\ \left(\cdot \quad \cdot \quad \cdot \right) \\ \mathbf{r}_i \\ \\ \end{bmatrix}}_{k} \right. \quad \begin{bmatrix} & \downarrow & \\ \rightarrow & \mathbf{r}_i \cdot \mathbf{c}_j & \\ & & \end{bmatrix} \tag{7.56}$$

Thus, we see that the dot product of the vectors **v** and **w** is just the matrix product $\mathbf{v}^\mathbf{T}\mathbf{w}$:

$$\mathbf{v} \cdot \mathbf{w} = \mathbf{v}^\mathbf{T}\mathbf{w} = \mathbf{w}^\mathbf{T}\mathbf{v}. \tag{7.57}$$

This leads to an interpretation of the product of the matrix **A** (with rows $\mathbf{a}_i$) and a vector **v**: If $\mathbf{w} = \mathbf{A}\mathbf{v}$, then the ith entry of **w** tells "how much **v** looks like $\mathbf{a}_i^\mathbf{T}$" (in the sense described when we discussed Equation 7.38).

There's another equally useful interpretation of $\mathbf{Av}$. If we let $\mathbf{b}_i$ denote the *columns* of **A,** then

$$\mathbf{Av} = v_1\mathbf{b}_1 + v_2\mathbf{b}_2 + \ldots + v_n\mathbf{b}_n, \tag{7.58}$$

that is, the product of **A** with **v** is a linear combination of the columns of **A.**

One particularly useful consequence of the definition of $\mathbf{Av}$ is that if we want to multiply **A** by several vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k$, we can express this product by concatenating the vectors $\mathbf{v}_i$ into a matrix **V** whose columns are $\mathbf{v}_1, \mathbf{v}_2$, etc.; the product

$$\mathbf{AV} \tag{7.59}$$

is then a matrix **W** whose *i*th column is $\mathbf{Av}_i$. This is no more computationally efficient than multiplying **A** by each individual vector. The key application of this idea is when we have one set of vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k$ and a second set of vectors $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_k$, and we wish to find a matrix with $\mathbf{Av}_i = \mathbf{w}_i$ for $i = 1, \ldots, k$, that is, we wish to have

$$\mathbf{AV} = \mathbf{W}. \tag{7.60}$$

Often it's impossible to achieve exact equality, but it will turn out that we can find the "best" such matrix **A** by straightforward operations on the matrices **V** and **W,** but we'll wait to present the main ideas in context in Section 10.3.9.

In general, matrix multiplication is not commutative: $\mathbf{AB} \neq \mathbf{BA}$.

Inline Exercise 7.3: (a) If **A** is a 2×3 matrix and **B** is a 3×1 matrix, show that **AB** makes sense, but **BA** does not.
(b) Let $\mathbf{A} = \begin{bmatrix}1 & 2 & 3\end{bmatrix}^{\mathbf{T}}$ and $\mathbf{B} = \begin{bmatrix}0 & 1 & 1\end{bmatrix}$. Compute both **AB** and **BA.**

7.6.6 Other Kinds of Vectors

The spaces $\mathbf{R^2}$ and $\mathbf{R^3}$ that we've been discussing, and $\mathbf{R}^n$ in general, have certain properties. There's a notion of addition (which is commutative and associative), and of scalar multiplication (again associative, and distributive over addition). There's also a zero vector with the property that $\mathbf{0} + \mathbf{v} = \mathbf{v} + \mathbf{0} = \mathbf{v}$ for any vector **v**. And there are additive inverses: Given a vector **v**, we can always find another vector **w** with $\mathbf{w} + \mathbf{v} = \mathbf{0}$. These properties, taken together, make $\mathbf{R}^n$ a vector space. There are a few other vector spaces we'll encounter in graphics; some of these will arise in our discussion of images, others in our discussion of splines, and still others in our discussion of rendering. Most have a common form: They are spaces whose elements are all *functions*.

Recall that, if we had a vector $\mathbf{w} \in \mathbf{R^2}$, we built a function

$$\phi_{\mathbf{w}} : \mathbf{R^2} \to \mathbf{R} : \mathbf{v} \mapsto \mathbf{w} \cdot \mathbf{v}. \tag{7.61}$$

This is a linear function on $\mathbf{R^2}$, and in fact, *any* linear function from $\mathbf{R^2}$ to **R** has this particular form.

Inline Exercise 7.4: Let $f : \mathbf{R^2} \to \mathbf{R}$ be a linear function. Let $a = f(\mathbf{e}_1)$, $b = f(\mathbf{e}_2)$, and $\mathbf{w} = \begin{bmatrix} a & b \end{bmatrix}^{\mathbf{T}}$. Show that for any vector $\mathbf{v}$, we have $f(\mathbf{v}) = \phi_{\mathbf{w}}(\mathbf{v})$. You'll need to use the fact that f is linear.

The collection of *all* such functions, that is,

$$\mathbf{R^{2^\star}} = \{\phi_{\mathbf{w}} : \mathbf{w} \in \mathbf{R^2}\}, \tag{7.62}$$

forms a vector space: The sum of any two linear functions is again linear; the zero element of the vector space is $\phi_{\mathbf{0}}$; and the additive inverse of $\phi_{\mathbf{w}}$ is $\phi_{-\mathbf{w}}$. Scalar multiplication deserves a brief comment. What does it mean to multiply a function from $\mathbf{R^2}$ to $\mathbf{R}$ by, say, 11? If f is such a function, then $g = 11f$ is the function defined by

$$g : \mathbf{R^2} \to \mathbf{R} : \mathbf{v} \mapsto 11f(\mathbf{v}), \tag{7.63}$$

that is, one multiplies the output of f by 11.

Inline Exercise 7.5: Suppose that $\mathbf{w} \in \mathbf{R^2}$. Explain why $3\phi_{\mathbf{w}} = \phi_{3\mathbf{w}}$.

This space of linear functions from $\mathbf{R^2}$ to $\mathbf{R}$ is called the **dual space** of $\mathbf{R^2}$, and its elements are sometimes called **dual vectors** or **covectors.** This same idea generalizes to $\mathbf{R^3}$ or even $\mathbf{R}^n$. There's an obvious correspondence between elements of $\mathbf{R^2}$ and elements of $\mathbf{R^{2^*}}$, namely we can associate the vector $\mathbf{w}$ with the covector $\phi_{\mathbf{w}}$. So why not just call them "the same"? It will turn out that treating them distinctly has substantial advantages; in particular, we'll see that when we transform all the elements of a vector space by some operation like rotation, or stretching the y-axis, the covectors transform *differently* in general.

In coordinate form, if $\mathbf{w} = \begin{bmatrix} a & b \end{bmatrix}^{\mathbf{T}}$, then the function $\phi_{\mathbf{w}}$ can be written

$$\phi_{\mathbf{w}} : \mathbf{R^2} \to \mathbf{R} : \begin{bmatrix} x \\ y \end{bmatrix} \mapsto \begin{bmatrix} a & b \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}. \tag{7.64}$$

Some books therefore identify covectors with row vectors, and ordinary vectors with column vectors.

Note that in designing software, just as it made sense to distinguish `Point` and `Vector`, it makes sense to have a `CoVector` class as well.

Covectors are particularly useful in representing triangle normals (the normal to a triangle is a nonzero vector perpendicular to the plane of the triangle, and is used in computing things like how brightly the triangle is lit by light coming in a certain direction). Although people often talk about normal vectors, such vectors are almost always used as *covectors*. To be precise, when we have a vector $\mathbf{n}$ normal to a triangle, we will almost never add $\mathbf{n}$ to another vector or a point, but we'll often use it in expressions like $\mathbf{n} \cdot \boldsymbol{\ell}$ (where $\boldsymbol{\ell}$ might be, for instance, the direction that light is arriving from). Thus, it's really the covector

$$\mathbf{u} \mapsto \mathbf{n} \cdot \mathbf{u} \tag{7.65}$$

that's of interest.

7.6.7 Implicit Lines

We've already encountered the parametric form of a line (Equation 7.24). There's another way to describe the line between P and Q: Instead of writing a function $t \mapsto \gamma(t)$ whose value at each real number t is a point of the line, we can write a different kind of function—one that tells, for each point (x, y) of $\mathbf{R^2}$, *whether* the point (x, y) is on the line. Such a function is said to define the line *implicitly* rather than parametrically. Such implicit descriptions are frequently useful; we'll see shortly that computing the intersection of two parametrically described lines is more difficult than computing the intersection of a parametrically defined line and an implicitly defined one. Since the operation of intersecting lines (or rays) with objects is one that arises frequently in graphics (we're very interested in where light, which travels in rays, hits objects in a scene!), we'll examine such implicit descriptions more fully.

If $F : \mathbf{R^2} \to \mathbf{R}$ is a function, then for each c, we can define the set

$$L_c = \{(x, y) : F(x, y) = c\}, \tag{7.66}$$

which is called the **level set** for F at c. As an example, consider an ordinary weather map. To each point (x, y) on the map there's an associated temperature $T(x, y)$. The set of all points where the temperature is 80°F is a level set, as are the sets where the temperature is 70°F, 60°F, etc. These sets are typically drawn as curves on the map; each of them is a level set for the temperature function. Similarly, a contour map like the one in Figure 7.11 typically has contour lines for various heights—these curves represent the level sets of the height function. In graphics, we often build a function F whose level set for $c = 0$ constitutes some shape. This set is called the **zero set** of F.

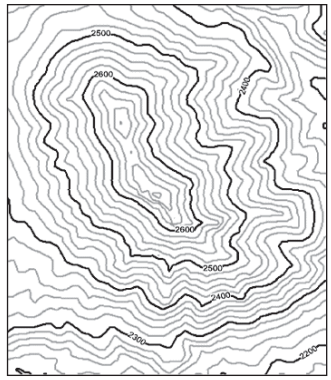


Figure 7.11: A contour map shows the height above sea level with contour lines.

Inline Exercise 7.6: Can two temperature-contour curves on a weather map ever cross? Why or why not?

7.6.8 An Implicit Description of a Line in a Plane

How can we find a function F whose value, on points of the line containing two distinct points P and Q, is zero, but whose value elsewhere is nonzero, that is, how can we find an implicit description of the line? Ponder that question briefly, then read on.

First,[4] let $\mathbf{n} = \times(Q-P) = (Q-P)^{\perp}$; then the vector $\mathbf{n}$ is perpendicular to the line (see Figure 7.12). A nonzero vector with this property is said to be a **normal vector** or simply a **normal** to the line.

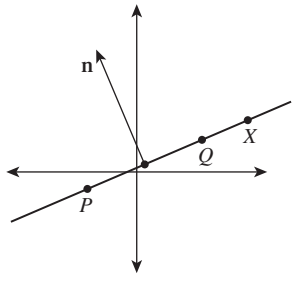


Figure 7.12: The vector $\mathbf{n} = (Q-P)^{\perp}$ *is perpendicular to the line through P and Q. A typical point X of this line has the property that* $(X - P)$ *is also perpendicular to* $\mathbf{n}$*. Indeed, a point X is on the line if and only if* $(X - P) \cdot \mathbf{n} = 0$*.*

Inline Exercise 7.7: The vector $\mathbf{n}$ is called *a* normal rather than *the* normal; show that this is justified by explaining why $2\mathbf{n}$ and $-\mathbf{n}$ are also normals to the line.

If X is a point of the line, then the vector $X - P$ points along the line, and so is also perpendicular to $\mathbf{n}$. If X is not on the line, then $X - P$ does *not* point along the line, and hence is not perpendicular to $\mathbf{n}$. Thus,

4. Note that we're using the 2D cross product defined in Equation 7.34 here.

$$(X - P) \cdot \mathbf{n} = 0 \tag{7.67}$$

completely characterizes points X that lie on the line. We can therefore define

$$F(X) = (X - P) \cdot \mathbf{n}, \tag{7.68}$$

which serves as an implicit description of the line. We'll call this the **standard implicit form for a line.**

Inline Exercise 7.8: What are the domain and codomain of the function F just defined?

Inline Exercise 7.9: Our discussion assumes that P and Q are distinct. What set does the function F implicitly define if P and Q are identical?

As a concrete example, if $P = (1, 0)$ and $Q = (3, 4)$, then $Q - P = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$ and $\mathbf{n} = \begin{bmatrix} -4 \\ 2 \end{bmatrix}$. Letting X have coordinates (x, y), we have

$$F(x, y) = \begin{bmatrix} x - 1 \\ y - 0 \end{bmatrix} \cdot \begin{bmatrix} -4 \\ 2 \end{bmatrix} = 0 \tag{7.69}$$

as the implicit form of our line; expressed in coordinates, this says

$$-4(x - 1) + 2y = 0 \tag{7.70}$$

or

$$-4x + 2y - 1 = 0, \tag{7.71}$$

which is the familiar $Ax + By + C = 0$ form for defining a line.

Both the implicit and parametric descriptions of lines generalize to three dimensions: Given two points in 3-space, the parametric form of Equations 7.28 given above will determine a line between them. The implicit form (i.e., $(X - P) \cdot \mathbf{n} = 0$) determines a *plane* passing through P and perpendicular to the vector $\mathbf{n}$; to determine a single line, we must take two such plane equations with two nonparallel normal vectors.

7.6.9 What About $y = mx + b$?

In graphics we generally avoid the "slope-intercept" formulation of lines (an equation of the form $y = mx + b$; m is called the *slope* and the point $(0, b)$ is the *y-intercept,* i.e., the point where the line meets the y-axis), because we cannot use it to express vertical lines; the "two-point" implicit and parametric forms above are far more general, and formulas involving them generally don't need special-case handling.

7.7 Intersections of Lines

We now have two forms for describing lines in the plane: implicit and parametric. (The parametric form extends to lines in $\mathbf{R}^n$.) With the help of the exercises,

you can easily convert between them. If we have two lines whose intersection we need to compute, we could do an implicit-implicit, parametric-parametric, or parametric-implicit computation. The implicit-implicit version is messy enough that it's better to convert one line to parametric form and use the implicit-parametric intersection approach. We'll begin by computing the intersection of two lines in parametric form, mostly so that you see the benefit of the later implicit-parametric form. In general, we find that intersections between implicit and parametric forms tend to produce the simplest algebra.

7.7.1 Parametric-Parametric Line Intersection

Suppose we have two lines in parametric form, that is, two functions

$$\gamma : \mathbf{R} \to \mathbf{R^2} : t \mapsto tA + (1-t)B, \tag{7.72}$$

$$\eta : \mathbf{R} \to \mathbf{R^2} : s \mapsto sC + (1-s)D, \tag{7.73}$$

whose images are (respectively) the line AB and the line CD (see Figure 7.13). We'd like to find the point P where these lines intersect. That point will lie on the line determined by γ, that is, it will be $\gamma(t_0)$ for some real number t_0; similarly, it'll be $\eta(s_0)$ for some real number s_0. Equating these gives

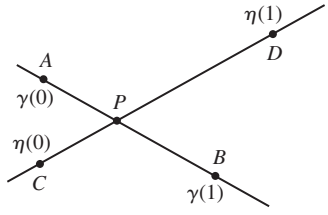


Figure 7.13: The lines AB and CD are the images of the parametric functions γ and η; they intersect at the unknown point P.

$$t_0A + (1-t_0)B = s_0C + (1-s_0)D, \tag{7.74}$$

which can also be written

$$B - D = -t_0(A-B) + s_0(C-D), \tag{7.75}$$

which is a nice form because it involves only vectors (i.e., differences between points).

If we write out Equation 7.74 in terms of the known coordinates of the points A, B, C, and D, we get two equations in the two unknowns s_0 and t_0 and can solve for them. We can then compute P by computing either $t_0A + (1-t_0)B$ *or* $s_0C + (1-s_0)D$.

An alternative, and preferable, approach is to use the vector form shown in Equation 7.75. Letting $\mathbf{v} = A - B$ and $\mathbf{u} = C - D$, we have

$$B - D = -t_0\mathbf{v} + s_0\mathbf{u}. \tag{7.76}$$

Taking the dot product of both sides with $\times\mathbf{v}$, we get

$$(B-D)\cdot(\times\mathbf{v}) = -t_0\mathbf{v}\cdot(\times\mathbf{v}) + s_0\mathbf{u}\cdot(\times\mathbf{v}) \tag{7.77}$$

$$(B-D)\cdot(\times\mathbf{v}) = s_0\mathbf{u}\cdot(\times\mathbf{v}) \tag{7.78}$$

$$\frac{(B-D)\cdot(\times\mathbf{v})}{\mathbf{u}\cdot(\times\mathbf{v})} = s_0 \tag{7.79}$$

where the simplification arises because $\times\mathbf{v}$ is perpendicular to $\mathbf{v}$, so their dot product is zero. The last trick—eliminating a term from an equation by taking the dot product of both sides with a vector orthogonal to that term—is often useful.

7.7.2 Parametric-Implicit Line Intersection

Now suppose we're given a parametric form for one line and an implicit form for a second line. How can we compute their intersection? We need to find the value t_0 of t with the property that

$$\gamma(t_0) = (1 - t_0)P + t_0Q \tag{7.80}$$

is on the line defined by $\ell = \{X : (X - S) \cdot \mathbf{n} = 0\}$.

For this to be true, we need that

$$(\gamma(t_0) - S) \cdot \mathbf{n} = 0, \tag{7.81}$$

that is

$$((1 - t_0)P + t_0Q - S) \cdot \mathbf{n} = 0. \tag{7.82}$$

Once again the vector form becomes useful as we simplify:

$$(P + t_0(Q - P) - S) \cdot \mathbf{n} = 0, \text{so} \tag{7.83}$$

$$(t_0(Q - P) + (P - S)) \cdot \mathbf{n} = 0. \tag{7.84}$$

Writing $\mathbf{u} = Q - P$ and $\mathbf{v} = P - S$, we have

$$t_0\mathbf{u} \cdot \mathbf{n} + \mathbf{v} \cdot \mathbf{n} = 0 \tag{7.85}$$

$$t_0\mathbf{u} \cdot \mathbf{n} = -\mathbf{v} \cdot \mathbf{n} \tag{7.86}$$

$$t_0 = \frac{-\mathbf{v} \cdot \mathbf{n}}{\mathbf{u} \cdot \mathbf{n}}. \tag{7.87}$$

Plugging this into the formula for $\gamma(t_0)$ yields the xy-coordinates of the intersection point P.

Inline Exercise 7.10: Carry out this final computation to find the coordinates of P. Try to do all your computations using dot products and vector operations rather than explicit coordinate computations.

Note that this computation gives us *two* things. It tells us where along the line determined by γ the intersection lies, by giving us t_0; if t_0 is between 0 and 1, for instance, we know that the intersection lies between P and Q. It also tells us the actual coordinates of the intersection point (x_0, y_0), that is, it provides an explicit point on the line that's determined implicitly by $Ax + By + C = 0$. If we only care about intersections between P and Q, but find t_0 is not between 0 and 1, we can avoid the second part of the computation.

7.8 Intersections, More Generally

In talking about intersections of lines, we've advocated the use of vectors and inner products. There are several advantages to this approach.

- Rather than writing out everything two (in two dimensions) or three (in three dimensions) times, once per coordinate, we write things just once, reducing the chance of error.

- Frequently the vector form of a computation generalizes naturally to n dimensions, while the generalization of the coordinate form may not be obvious (a nice example is the decomposition of a vector into a part parallel to a given vector $\mathbf{u}$ and a part that's perpendicular to $\mathbf{u}$; our formula works in 2-space, 3-space, 4-space, etc.).
- Our programs become easier to read and debug when we write less code. Programs written to reflect the vector description of computations are usually simpler.

We'll consider two more examples of the vector form of computation of intersections: the intersection of a ray with a plane and with a sphere.

7.8.1 Ray-Plane Intersection

Let's intersect a ray, given by its starting point P and direction vector $\mathbf{d}$ (so that a typical ray point is $P + t\mathbf{d}$, with $t \geq 0$), with a plane, specified by a point Q of the plane and a normal vector $\mathbf{n}$. A point X of the plane therefore has the property that

$$(X - Q) \cdot \mathbf{n} = 0, \tag{7.88}$$

which generalizes the standard implicit form of a line in $\mathbf{R^2}$.

We seek a value, $t \geq 0$, with the property that the point $P + t\mathbf{d}$ is on the plane, for this point will be on both the plane and the ray, that is, at their intersection. Let's proceed by assuming that such an intersection point exists and is unique; we'll return to this assumption shortly.

For $P + t\mathbf{d}$ to lie on the plane, it must satisfy

$$((P + t\mathbf{d}) - Q) \cdot \mathbf{n} = 0. \tag{7.89}$$

We can now simplify this considerably and solve for t as follows:

$$\begin{aligned}
((P + t\mathbf{d}) - Q) \cdot \mathbf{n} &= 0 && (7.90)\\
(P - Q + t\mathbf{d}) \cdot \mathbf{n} &= 0 && (7.91)\\
(P - Q) \cdot \mathbf{n} + t\mathbf{d} \cdot \mathbf{n} &= 0 && (7.92)\\
t\mathbf{d} \cdot \mathbf{n} &= -(P - Q) \cdot \mathbf{n} && (7.93)\\
t\mathbf{d} \cdot \mathbf{n} &= (Q - P) \cdot \mathbf{n} && (7.94)\\
t &= \frac{(Q - P) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}} && (7.95)
\end{aligned}$$

The reader will notice that this is essentially the same computation we did in Equation 7.87 earlier; once again, the vector form generalizes nicely.

Inline Exercise 7.11: In the algebra shown above, why did we not write $P \cdot \mathbf{n} + t\mathbf{d} \cdot \mathbf{n} - Q \cdot \mathbf{n} = 0$ as a simplification of the first equation?

Knowing t, we can now compute the intersection point $P + t\mathbf{d}$. Two small concerns remain:

- The possibility of division by zero in the expression for t
- The assumption that an intersection exists and is unique

These two concerns, it turns out, are identical! If $\mathbf{d} \cdot \mathbf{n} = 0$, then the ray is parallel to the plane; that means it's either contained in the plane (in which case there are infinitely many intersections) or disjoint from the plane, in which case there are none. In the first case, both the numerator and denominator are zero; in the second, only the denominator is.

Inline Exercise 7.12: What happens if we solve for t and find that it's negative? That can happen if either the numerator or denominator is negative, and the other is positive. Describe each of these situations geometrically, as in "In the first case, P is on the positive side of the plane (i.e., the half-space into which $\mathbf{n}$ points), and . . ."

Let's carry out the computation again, in this case with a plane specified by a point Q that lies in the plane, and two vectors, $\mathbf{u}$ and $\mathbf{v}$, that lie in the plane and are linearly independent. Now, points in the plane can be written in the form

$$Q + \alpha\mathbf{u} + \beta\mathbf{v} \tag{7.96}$$

for some real numbers α and β. This version of the problem seems distinctly more difficult, in the sense that a solution will give us t, α, and β; that impression is partially correct, as we'll see.

We want to find a value $t \geq 0$ with the property that

$$P + t\mathbf{d} = Q + \alpha\mathbf{u} + \beta\mathbf{v} \tag{7.97}$$

for *some* values α and β. The first algebraic steps are similar. Move the points around so that a difference of points appears, and we'll be working only with vectors:

$$P + t\mathbf{d} = Q + \alpha\mathbf{u} + \beta\mathbf{v} \tag{7.98}$$

$$P - Q + t\mathbf{d} = \alpha\mathbf{u} + \beta\mathbf{v} \tag{7.99}$$

$$P - Q = \alpha\mathbf{u} + \beta\mathbf{v} - t\mathbf{d}. \tag{7.100}$$

Letting $\mathbf{h} = P - Q$, we can see that now the problem is "Express the vector $\mathbf{h}$ as a linear combination of $\mathbf{u}, \mathbf{v}$, and $\mathbf{d}$." We could let $\mathbf{M}$ be the matrix whose columns are these three vectors, and the solution becomes

$$\begin{bmatrix}\alpha & \beta & -t\end{bmatrix} = \mathbf{M}^{-1}(P - Q). \tag{7.101}$$

To implement this, we'd need to invert a 3×3 matrix, which is not difficult, but masks some of the essential features of the problem. First, it's possible that $\mathbf{M}$ is not invertible, but even if this happens, there may be a solution to

$$\mathbf{M}\begin{bmatrix}\alpha & \beta & -t\end{bmatrix} = (P - Q). \tag{7.102}$$

($\mathbf{M}$ is noninvertible when the ray is parallel to the plane; the solution exists when the ray lies *in* the plane, and indeed, in this case there are infinitely many solutions.) Second, the inversion must be redone for each new ray we want to intersect with the plane; that makes ray-plane intersection computationally intensive, which is bad.

An alternative is to say, "Let's just compute $\mathbf{n} = \mathbf{u} \times \mathbf{v}$," and use the previous solution; that's an excellent choice, because the cross-product computation can be performed once, and stored for later reuse. The point here is that computing a ray intersection with the implicit form is computationally far easier than doing so with the parametric form of the plane.

7.8.2 Ray-Sphere Intersection

Once again, suppose we have a ray whose points are of the form $P + t\mathbf{d}$, with $t \geq 0$. We want to compute its intersection with a sphere given by its center, Q, and its radius, r.

The implicit description of this sphere is that the point X is on the sphere if the distance from X to Q is exactly r; this is equivalent to the *squared* distance (which is almost always easier to work with) being r^2, that is,

$$(X - Q) \cdot (X - Q) = r^2, \tag{7.103}$$

where we've used the fact that the squared length of a vector $\mathbf{v}$ is just $\mathbf{v} \cdot \mathbf{v}$.

Now let's ask, "Under what conditions on t is $P + t\mathbf{d}$ on the sphere?" It must satisfy

$$((P + t\mathbf{d}) - Q) \cdot ((P + t\mathbf{d}) - Q) = r^2. \tag{7.104}$$

Because the vector $P - Q$ is going to come up a lot, we'll give it a name, $\mathbf{v}$; now we can simplify the expression above:

$$((P + t\mathbf{d}) - Q) \cdot ((P + t\mathbf{d}) - Q) = r^2 \tag{7.105}$$

$$((P - Q) + t\mathbf{d}) \cdot ((P - Q) + t\mathbf{d}) = r^2 \tag{7.106}$$

$$(\mathbf{v} + t\mathbf{d}) \cdot (\mathbf{v} + t\mathbf{d}) = r^2 \quad (\text{because } \mathbf{v} = P - Q) \tag{7.107}$$

$$(\mathbf{v} \cdot \mathbf{v}) + 2(t\mathbf{d} \cdot \mathbf{v}) + (t\mathbf{d} \cdot t\mathbf{d}) = r^2 \tag{7.108}$$

$$(\mathbf{v} \cdot \mathbf{v} - r^2) + t(2\mathbf{d} \cdot \mathbf{v}) + t^2(\mathbf{d} \cdot \mathbf{d}) = 0. \tag{7.109}$$

This last equation is a quadratic in t, which therefore has zero, one, or two real solutions.

Inline Exercise 7.13: Describe geometrically the conditions under which this equation will have zero, one, or two solutions, as in "If the ray doesn't intersect the sphere, there will be no solutions; if . . . "

Fortunately, it's easy to tell whether a quadratic $c + bt + at^2 = 0$ has zero, one, or two real solutions. The quadratic formula tells us that the solutions are

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, \tag{7.110}$$

which are real if $b^2 - 4ac \geq 0$; the two solutions are identical precisely if the square root is zero, that is, if $b^2 = 4ac$. In our case, this turns out to mean that there are two solutions if

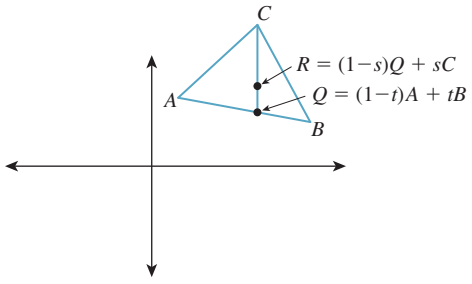


Figure 7.14: The point $Q = (1 - t)\,A + tB$ is on the edge between A and B (provided $t \in [0, 1]$), and $(1 - s)\,Q + sC$ is on the line segment from Q to C (when $s \in [0, 1]$).

$$(\mathbf{d} \cdot \mathbf{v})^2 > (\mathbf{v} \cdot \mathbf{v} - r^2)(\mathbf{d} \cdot \mathbf{d}) \tag{7.111}$$

and one solution if the two sides are equal.

Note that if we choose to represent our ray with a vector $\mathbf{d}$ that has unit length, this computation simplifies somewhat, since $\mathbf{d} \cdot \mathbf{d} = 1$.

The examples above—line-line intersection, line-plane intersection, ray-sphere intersection—all serve to illustrate the following general principle.

✓ **THE PARAMETRIC/IMPLICIT DUALITY PRINCIPLE:** There's a duality between parametric and implicit forms for shapes. In general, it's easy to find an intersection between shapes where one is described implicitly and the other parametrically, and harder when either both are implicit or both are parametric.

7.9 Triangles

Triangles, which are familiar from geometry, are the building blocks of much of computer graphics. If a triangle has vertices A, B, and C, then a point of the form $Q = (1 - t)A + tB$, where $0 \leq t \leq 1$ is on the edge between A and B (see Figure 7.14). Similarly, a point of the form $R = (1 - s)Q + sC$ is on the line segment between Q and C if $0 \leq s \leq 1$. Expanding this out, we get

$$R = (1 - s)(1 - t)A + (1 - s)tB + sC. \tag{7.112}$$

Equation 7.112 is worth examining carefully in several ways. First, we can define a function,

$$F : [0, 1] \times [0, 1] \to \mathbf{R^2} : (s, t) \mapsto (1 - s)(1 - t)A + (1 - s)tB + sC, \tag{7.113}$$

whose image is exactly the triangle ABC (see Figure 7.15). F sends the entire right edge ($s = 1$) of the square to the single point C. Vertical lines ($s = constant$) are sent to lines parallel to AB. Horizontal lines ($t = constant$) are sent to lines through C and a point of the edge AB. This **parameterization** of the triangle (the variables s and t are the parameters) is often used in graphics.

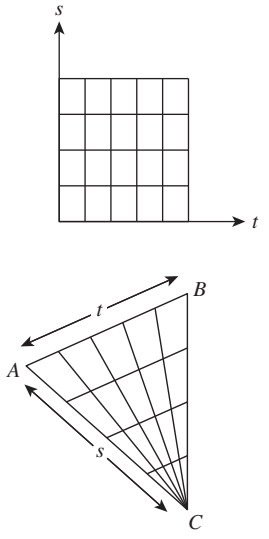


Figure 7.15: The function $F : [0, 1] \times [0, 1] \to \mathbf{R^2} : (s, t) \mapsto (1 - s)\,(1 - t)A + (1 - s)tB + stC$, from the unit square to the triangle ABC, sends the entire $s = 1$ edge to the point C. All other lines in the square are sent to lines in the triangle as shown.

7.9.1 Barycentric Coordinates

Let's also look at the coefficients in Equation 7.112: They are $(1-s)(1-t)$, $(1-s)t$, and s. It's easy to see that for $0 \leq s, t \leq 1$, all three are positive; summing them up we get

$$(1-s)(1-t) + (1-s)t + s = (1-s)((1-t)+t) + s \tag{7.114}$$
$$= (1-s) + s \tag{7.115}$$
$$= 1. \tag{7.116}$$

(This is good: Combinations of multiple points are only defined when the coefficients sum to one.) So we can say that the points of the triangle are of the form

$$\alpha A + \beta B + \gamma C, \tag{7.117}$$

where $\alpha + \beta + \gamma = 1$, and $\alpha, \beta, \gamma \geq 0$. Points where $\alpha = 0$ lie on the edge BC; points where $\beta = 0$ lie on the edge AC; points where $\gamma = 0$ lie on the edge AB. If $P = \alpha A + \beta B + \gamma C$, then the numbers α, β, and γ are called the **barycentric coordinates** of P with respect to the triangle ABC.

Inline Exercise 7.14: (a) What are the barycentric coordinates of the midpoint of the edge AB in the triangle ABC? (b) What about the centroid?

Inline Exercise 7.15: Suppose $A = (1,0,0), B = (0,1,0)$, and $C = (0,0,1)$, and the point P of triangle ABC has barycentric coordinates α, β, and γ. What are the 3D coordinates of P?

Two other descriptions of the barycentric coordinates of a point in a triangle are often useful.

- In a nondegenerate triangle ABC, the A-coordinate of a point P is the perpendicular distance of P from the edge BC, scaled so that the A-coordinate of the point A is exactly one. There are two ways to see this. The first is to simply write everything out in terms of coordinates. The other is to realize that the "perpendicular distance" function and the "A-coordinate" function are both affine functions on the plane, and they agree at three points (A, B, and C), and this suffices to determine them uniquely.
- From the preceding description, it's easy to see that the area of the triangle PCB, being half the product of the length of BC and the length of the perpendicular from P to BC, is proportional to that perpendicular length. So the A-coordinate of P is proportional to the area of triangle PBC. The proportionality constant is exactly the area of ABC, that is, the A-coordinate of P is

$$\frac{\text{Area}(\triangle PBC)}{\text{Area}(\triangle ABC)}. \tag{7.118}$$

  The same proof works for this case. In short, the point P provides a natural partition of the triangle ABC into three subtriangles; the fractions of the area of ABC represented by each of these triangles are the barycentric coordinates of P (Figure 7.16).

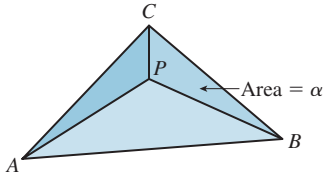


Figure 7.16: The point P divides triangle ABC into three smaller triangles, whose areas are fractions α, β, and γ of the whole; the barycentric coordinates of P are α, β, and γ, that is, $P = \alpha A + \beta B + \gamma C$.

7.9.2 Triangles in Space

The formulas given above—the parameterization of the triangle by a square, and the barycentric coordinates—don't depend on the dimension. They work just as well when the points A, B, and C are in 3-space as in the plane! There is one aspect of a triangle in 3-space that's different from the planar case: A triangle in 3-space is contained in a particular plane defined by an implicit equation of the form $F(X) = (X - P) \cdot \mathbf{n} = 0$. For P, we can use any of the three vertices of the triangle; for $\mathbf{n}$, we can use the cross product

$$\mathbf{n} = (B - A) \times (C - B). \tag{7.119}$$

If the angle at B is *very* near zero or π, this computation can be numerically unstable (see Section 7.10.4).

With this in mind, let's solve a frequently occurring problem: finding the intersection of a ray $t \mapsto P + t\mathbf{d}$ with a triangle ABC in 3-space. There are several cases. The intersection might occur where $t < 0$; the line containing the ray might not intersect the triangle at all; or the ray and the triangle might be in the same plane, and their intersection could be empty, a point, or a line segment. These last few cases have the property that a tiny numerical error—a slight change to one of the coordinates—can change the answer entirely. Such instabilities make the answers we compute almost useless. So, if the direction vector $\mathbf{d}$ is sufficiently close to perpendicular to the normal vector $\mathbf{n}$, we'll return a result of "UNSTABLE" rather than computing an intersection. Our strategy is to first find the parameter t at which the ray intersects the plane of the triangle, then find $Q = P + t\mathbf{d}$, the intersection point in the plane, and finally find the barycentric coordinates of Q, which in turn tell us whether Q is inside the triangle.

For the most common situation—we'll make many ray-intersect-triangle computations for a single triangle—it's worth storing some additional data, per triangle, to speed the computation. We'll precompute the normal vector, $\mathbf{n}$, and two vectors $AB^\perp$ and $AC^\perp$, in the plane of ABC with the property that (a) $AB^\perp$ is perpendicular to AB, and $(C - A) \cdot AB^\perp = 1$, and similarly for $AC^\perp$. If X is the point $X = \alpha A + \beta B + \gamma C$ in the plane of ABC, then we can compute γ easily as $\gamma = AB^\perp \cdot (X - C)$, and a similar computation gives us β. Finally, we find $\alpha = 1 - (\beta + \gamma)$.

With these ideas in mind, Listing 7.2 shows the actual structure.

If you consider any plane equation of the form $f(X) = (X - A) \cdot \mathbf{u} = 0$, then the value

$$f(P + t\mathbf{d}) = (P + t\mathbf{d} - A) \cdot \mathbf{u} \tag{7.120}$$

is a linear function of t. For instance, if f is the equation of the plane of the triangle, we can solve for t to find the intersection of the ray with that plane. But suppose that f is the equation for the plane containing the edge AB and the normal vector $\mathbf{n}$. Then f, when restricted to the triangle plane, is zero on the line AB, and nonzero on the point C (assuming the triangle is nondegenerate). That means that some multiple of f—namely $X \mapsto f(X)/f(C)$—gives the barycentric coordinate at C. Now when we look at $f(P + t\mathbf{d})$, we can see how fast the C-coordinate of the projection of $P + t\mathbf{d}$ into the triangle plane is changing. When we find the t-value at which the intersection occurs, we can therefore easily find the barycentric coordinate for the intersection point, as long as we know this plane equation.

Listing 7.2: Intersecting a ray with a triangle.

```
// input: ray P + td; triangle ABC
// precomputation

n = (B − A) × (C − A)
AB⊥ = n × (B − A)
AB⊥ /= (C − A) · AB⊥
AC⊥ = n × (A − C)
AC⊥ /= (B − A) · AC⊥


// ray-triangle intersection
u = n · d
if (|u| < ε) return UNSTABLE

t = ((A−P)·n)/u
if t < 0 return RAY_MISSES_PLANE

Q = P + td
γ = (Q − C) · AC⊥
β = (Q − B) · AB⊥
α = 1 − (β + γ)
if any of α,β,γ is negative or greater than one
   return (OUTSIDE_TRIANGLE, α, β, γ)
else
   return (INSIDE_TRIANGLE, α, β, γ)
```

Section 15.4.3 applies this idea in developing an alternative ray-triangle intersection procedure—one which, at its core, is very similar to the one we've given, but which, when you read it, may seem completely opaque. Why have more than one method? Ray-triangle intersection testing is at the heart of a great deal of graphics code. It's one of those places where the slightest gain in efficiency has an impact everywhere. We wanted to show you two different approaches in hopes that you might find another, even faster approach, and to show you some of the optimization tricks that might help you improve your own inner-loop code.

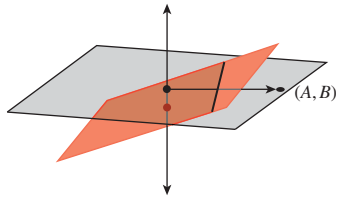


Figure 7.17: The graph of a linear function on $\mathbf{R}^2$ *defined by* $F(x,y,z) = Ax+By+C$ *is a plane shown in red, which is tilted relative to the large pale gray* $z = 0$ *plane as long as either* A *or* B *is nonzero; it intersects the* $z = 0$ *plane in a line* ℓ *(a portion of which is shown as a heavy black segment). The ray from the origin to the point* (A,B) *is perpendicular to* ℓ*.*

7.9.3 Half-Planes and Triangles

From algebra we know that the function $F(x,y) = Ax+By+C$ (where A and B are not both zero) has a graph that's a plane intersecting the xy-plane in a line ℓ. We've seen that the ray from the origin to (A,B) is perpendicular to ℓ. The zero set of F is exactly the line ℓ, that is, if $P \in \ell$, then $F(P) = 0$. On one side of ℓ the function F is positive; on the other it's negative (see Figure 7.17). Thus, an inequality like $F(x,y) = Ax + By + C \geq 0$ defines a **half-plane bounded by** ℓ, provided that A and B are not both zero. But which side of the line ℓ does it describe? The answer is this: If you move from ℓ in the direction of the normal vector $\begin{bmatrix} A & B \end{bmatrix}^{\mathbf{T}}$, you are moving to the side where $F > 0$.

A triangle in the plane can also be characterized as the intersection of three half-planes. If the vertices are P, Q, and R, one can consider the half-plane bounded by the line PQ and containing R, the one bounded by QR and containing P, and the one defined by PR and containing Q. Such a description can be used for testing whether a point is contained in a triangle. If the three inequalities are $F_1 \geq 0$, $F_2 \geq 0$, and $F_3 \geq 0$, one can test a point X for inclusion in the triangle as follows:

- If $F_1(X) < 0$ then *outside*
- If $F_2(X) < 0$ then *outside*
- If $F_3(X) < 0$ then *outside*
- Else *inside*.

Notice that this sequence of tests does early rejection: If X is on the wrong side of one edge, we don't bother computing whether it's on the right or wrong side of the others.

Building these functions F_i is similar to what we just did in Section 7.9.2. The function for testing against edge AB in that case would be a dot product of $X - A$ with the vector $AB^{\perp}$, for example.

For a triangle in 3-space specified with three vertices, P_0, P_1, and P_2, the normal is $(P_1 - P_0) \times (P_2 - P_0)$, normalized. What happens if we re-order the vertices? We get one of two answers, depending on whether we do an even or an odd permutation. Every triangle therefore has two orientations; we'll stick with the convention above that the vertices of a triangle are always named *in some order,* and that this order determines the normal vector that you mean. When we build solid objects from collections of triangles, we'll make a policy that the normals always point "to the outside."

7.10 Polygons

A polygon is a shape like the ones shown in Figure 7.18; we typically describe a polygon by listing its vertices in order. Some polygons are non-self-intersecting ((a), (b), (d)); these are called **simple** polygons. Among these, some are **convex,** in the sense that a line segment drawn between any two points on the edge (like the dashed gray horizontal line shown in (a)) lies entirely inside the polygon. In the case of nonsimple polygons like (c) and (e), the angles at the vertices may all be nonzero, or they may, like the angle at the upper right in (e), be zero; such a point is sometimes called a reflex vertex.

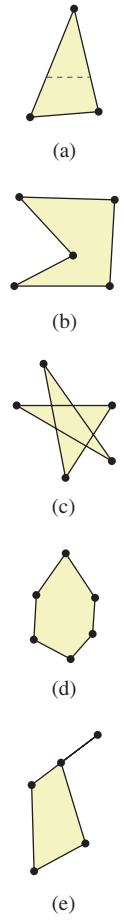


Figure 7.18: Polygons (a), (b), and (d) are simple, while (c) and (e) are not. Polygon (e) has a reflex vertex, (i.e., one with vertex angle zero) at the upper right.

7.10.1 Inside/Outside Testing

A polygon in the plane, in classic geometry, was simply a shape like the ones in Figure 7.18; because we attach an *order* to the vertices, we have slightly more structure. This allows us to define *inside* and *outside* for a larger class of polygons. Suppose that $(P_0, P_1, \ldots, P_n)$ is a polygon. The line segments P_0P_1, P_1P_2, etc., are the *edges* of the polygon; the vectors $\mathbf{v}_0 = P_1 - P_0$, $\mathbf{v}_1 = P_2 - P_1$, $\ldots \mathbf{v}_n = P_0 - P_n$ are the **edge vectors** of the polygon.[5] For each edge P_iP_{i+1}, the **inward edge normal** is the vector $\times\mathbf{v}_i$; the **outward edge normal** is $- \times \mathbf{v}_i$. For a convex polygon whose vertices are listed in counterclockwise order, the inward edge normals point toward the interior of the polygon, and the outward edge normals point toward the unbounded exterior of the polygon, corresponding to our ordinary intuition. But if the vertices of a polygon are given in clockwise order,

5. It's convenient to write $\mathbf{v}_i = P_{i+1} - P_i$, $i = 0, \ldots, n$; unfortunately, this formula fails at $i = n$ because P_{n+1} is undefined. Henceforth, it will be understood that in cases like this, $P_{n+1} = P_0$, $P_{n+2} = P_1$, etc. In other words, we continue the labeling cyclically; the same goes for indices less than zero: $P_{-1} = P_n$.

the interior and exterior swap roles. This is convenient in many situations. For example, imagine a polygon-shaped hole in a piece of metal. The interior of this polygon can (by suitable ordering of the vertices) be made to be the rest of the metal sheet. A light ray, trying to get from one side of the metal sheet to the other, is occluded exactly if it meets the interior of this polygon.

We can test whether a point in the plane is in the interior or exterior of a polygon, as shown in Figure 7.19, by casting a ray from the point in any direction. We find all intersections of the ray with polygon edges, and classify each as either *entering* (if the direction vector of the ray and the outward edge normal for the edge have a positive inner product) or *leaving* (if the inner product is negative). A point with more leaving intersections than entering ones is inside. Indeed, the difference between the number of leaving intersections and the number of entering intersections is called the **winding number** of the polygon about the point, and captures the notion of "how many times the polygon wraps around the point, counting counterclockwise." The simple description above depends on the intersections of the ray and the polygon being a finite set. It doesn't work when the ray actually contains an entire edge, and the answer, when the test point lies *on* an edge, depends on one's definition of the intersection of a ray and a segment (as does the answer when the ray passes through a vertex of the polygon). This ray-casting approach to computing the winding number implicitly uses a strong theorem that says that the ray-casting computation results in the same value as the computation of the winding number itself, which is defined in a very different way: For a polygon with vertices $P_1, P_2, \ldots, P_n$, the winding number about a point Q is defined by considering the angles between successive rays from Q to each P_i. If these angles sum to 2π, the winding number is 1; if they sum to 4π, it's 2, etc. Formally,

$$\text{WindingNumber}(Q, \{P_1, P_2, \ldots, P_n\}) = \frac{1}{2\pi} \sum_{i=1}^{n} \cos^{-1}\left(\frac{(P_{i+1} - Q) \cdot (P_i - Q)}{\|P_{i+1} - Q\| \, \|P_i - Q\|}\right). \tag{7.121}$$

Note that this is undefined if Q happens to be one of the vertices P_i, because the denominator is then zero. We also treat it as undefined when the argument to $\cos^{-1}$ is -1; this occurs when Q is on an edge of the polygon.

With these various definitions of the winding number, one can think of a curve as dividing the plane into a collection of regions. Within each region, the winding number is a constant; the labeling of regions by their winding numbers goes back to Listing, a student of Gauss [Lis48].

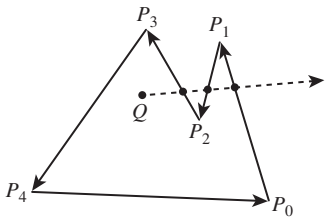


Figure 7.19: To test whether Q is in the interior of the polygon, we cast a ray in an arbitrary direction, and then count entering and leaving intersections with the edges. In this case, there are two leaving intersections (the first and third) and one entering intersection, so the point is inside.

Inline Exercise 7.16: (a) Develop a small program to test whether a point is in the interior of a *convex* polygon by testing whether it's strictly on the proper side of each edge. What's the running time in terms of the number n of polygon vertices? (b) Assuming that you want to test *many* points to determine whether they're inside or outside a *single* convex polygon, you can afford to do some preprocessing. Using a ray-casting test like the one described in this chapter, and a horizontal ray, show how you can create an $O(\log n)$ algorithm for inside/outside testing. Hint: Because the polygon is *convex,* you need only test ray intersection with a small number of edges. If you sort the vertices by their y-coordinates, how can you quickly find those edges?

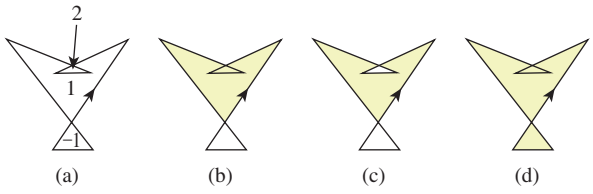


Figure 7.20: (a) A nonsimple polygon whose regions have been labeled by Listing's rule, (b) the interior defined by the positive winding number rule, (c) by the even-odd rule, and (d) by the nonzero winding number rule.

7.10.2 Interiors of Nonsimple Polygons

Most of the polygons we'll encounter will be triangles, and therefore simple. But occasionally we'll encounter nonsimple polygons in the plane, and there are several alternatives (see Figure 7.20) for defining the interior of these. All first compute the winding number of the polygon about the point P as before; in this case, though, the winding number may be different from one or zero. The **positive winding number rule** says that points with *positive* winding numbers are inside; the **odd winding number rule** says that those with *odd* winding numbers are inside—the result is a checkerboardlike appearance for the inside and outside regions; the **nonzero winding number rule** says that points with *nonzero* winding numbers are inside. Each has its uses; drawing programs should probably allow all three as alternatives.

7.10.3 The Signed Area of a Plane Polygon: Divide and Conquer

If $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ are points in the xy-plane (see Figure 7.21), and $Q = (0, 0)$ is the origin, then the **signed area** of the triangle QP_1P_2 is given by

$$\frac{1}{2}(x_1y_2 - y_1x_2). \tag{7.122}$$

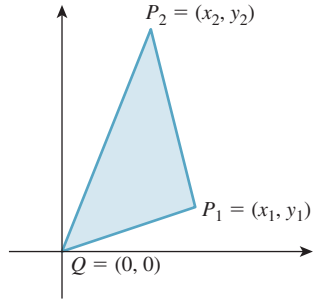


Figure 7.21: The signed area of the triangle QP_1P_2 is positive if the path from Q to P_1 to P_2 to Q is counterclockwise, and negative if it's clockwise. In the example shown, the signed area is positive.

The signed area is positive if the vertices Q, P_1, P_2 are in counterclockwise order around the triangle, and negative if they're in clockwise order. (It will be easy to prove this after you read Chapter 10, which discusses transformations in the plane: You first verify that the formula doesn't change when the triangle is rotated, so you can assume that P_1 lies on the positive x-axis, and therefore, $P_1 = (x_1, 0)$ with $x_1 > 0$. The triangle is then clockwise if and only if $P_2 = (x_2, y_2)$ is in the $y > 0$ half-space, that is, $y_2 > 0$. But the area formula then gives the area as $\frac{1}{2}x_1y_2 > 0$. The clockwise case is similarly easy to verify.)

From this formula for the area of a triangle, we can write a formula for the signed area of a triangle with vertices $P_0 = (x_0, y_0), P_1 = (x_1, y_1)$, and $P_2 = (x_2, y_2)$: If we change to a coordinate system based at P_0, the new coordinates of P_1 and P_2 are $(x_1 - x_0, y_1 - y_0)$ and $(x_2 - y_0, y_2 - y_0)$. Applying the formula to these coordinates gives

$$\frac{1}{2}((x_1y_2 - x_2y_1) + (x_2y_0 - x_0y_2) + (x_0y_1 - x_1y_0)) = \frac{1}{2}\sum_{i=0}^{2}(x_iy_{i+1} - x_{i+1}y_i), \tag{7.123}$$

where indices are considered modulo 3, so x_3 means x_0 and y_3 means y_0.

To find the (signed) area of a polygon $P_0P_1 \ldots P_n$, we can use the same technique (see Figure 7.22): We compute the signed area of QP_0P_1, of QP_1P_2, ..., and of QP_nP_0; the parts of those areas that are outside the polygon cancel, while the ones inside add up to the correct total area. The final form is

$$area = \frac{1}{2}\sum_{i=0}^{n}(x_i y_{i+1} - y_i x_{i+1}), \tag{7.124}$$

where x_{n+1} denotes x_0 (i.e., the indices are taken modulo $(n+1)$).

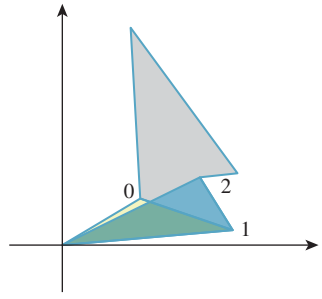


Figure 7.22: The pale yellow triangle is negatively oriented; the blue one positively. The next three will be negative, positive, and negative, and their signed areas will sum to give the gray polygon's area.

7.10.4 Normal to a Polygon in Space

A triangle $P_0P_1P_2$ in space has a normal $\mathbf{n}$ as we computed earlier, given by $\mathbf{n} = (P_1 - P_0) \times (P_2 - P_1)$. We could use this formula to compute the normal to a polygon in space as well, applying it to three successive vertices. But if these vertices happen to be collinear, we'll get $\mathbf{n} = \mathbf{0}$.

A more interesting approach, essentially due to Plücker [Plü68], is based on projected areas (see Figure 7.23 for an example where the polygon is a triangle): If one projects the polygon to the xy-, yz-, and zx-planes, one gets three planar polygons, each of whose areas can be computed; call these A_{xy}, A_{yz}, and A_{zx}. The normal vector to the polygon is then

$$\begin{bmatrix} A_{yz} \\ A_{zx} \\ A_{xy} \end{bmatrix}; \tag{7.125}$$

this is not generally a unit vector. In fact, its length is the area of the polygon.

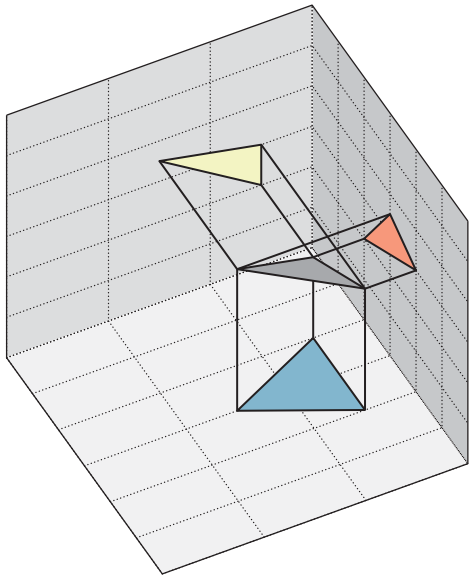


Figure 7.23: The gray triangle projects to three triangles, one in each coordinate plane. The signed areas of the orange, yellow, and blue triangles form the coordinates for the normal vector to the gray triangle.

We can see this, in the case that the polygon is a triangle, by examining coordinates: If $P_i = (x_i, y_i, z_i)$ for $i = 0, 1, 2$, then n_1, the first entry of the cross product, is just

$$n_1 = (y_1 - y_0)(z_2 - z_1) - (y_2 - y_1)(z_1 - z_0). \tag{7.126}$$

The area formula for a plane polygon, Equation 7.124, applied to the projection of $P_0P_1P_2$ to the yz-plane, gives

$$A_{xy} = \frac{1}{2}\sum_{i=0}^{n}(y_i z_{i+1} - z_i y_{i+1}), \text{ so} \tag{7.127}$$

$$2A_{xy} = (y_0 z_1 - z_0 y_1) + (y_1 z_2 - z_1 y_2) + (y_2 z_0 - z_2 y_0). \tag{7.128}$$

The eight terms in the expansion of the expression for n_1 match the six terms in the expression for $2A_{xy}$ because two y_1z_1 terms have opposite signs and cancel. The computations for n_2 and n_3 are similar. Thus, the vector

$$\mathbf{a} = \begin{bmatrix} A_{yz} \\ A_{zx} \\ A_{xy} \end{bmatrix} \tag{7.129}$$

is twice the cross product; since half the length of the cross product is that triangle area, we see that the length of $\mathbf{a}$ is the triangle area.

The more general case follows by decomposing the polygon into a union of triangles.

The advantage of Plücker's formula as applied to polygons in space is that if there are small numerical errors in the coordinates of a single vertex, they have relatively little impact on the computed normal vector.

7.10.5 Signed Areas for More General Polygons

If we have a polygon $P_0P_1 \ldots P_n$ in a plane S whose normal is the unit vector $\mathbf{n}$, we can find two orthogonal unit vectors $\mathbf{x}$ and $\mathbf{y}$ in S such that $\mathbf{x}, \mathbf{y}, \mathbf{n}$ is positively oriented, that is, $\mathbf{n} = \mathbf{x} \times \mathbf{y}$. Choosing some point of the plane as the origin, we have a coordinate system. We can then write the coordinates of each P_i in the xyn-coordinate system; the third coordinate will be zero, so the coordinates of P_i will be $(x_i, y_i, 0)$. We can apply the formula for a signed area to these coordinates. In the case of a triangle, if the signed area is positive (resp. negative), we say that the triangle is **positively** (resp. **negatively**) **oriented.** Notice, though, that if we had used $-\mathbf{n}$ instead of $\mathbf{n}$, the signs would have changed: The signed area, and the orientation of a triangle, are only defined relative to a plane-with-normal.

Just as in the case of the xz-plane, a triangle is positively oriented in a plane with normal $\mathbf{n}$ if, as we look from the tip of $\mathbf{n}$ toward the plane, the vertices of the triangle appear in counterclockwise order.

When we speak of the signed area of a polygon in the zx-plane, we mean that we're using a coordinate system in which the first basis vector is $\begin{bmatrix}0 & 0 & 1\end{bmatrix}^{\mathbf{T}}$, the second is $\begin{bmatrix}1 & 0 & 0\end{bmatrix}^{\mathbf{T}}$, and the normal vector is $\begin{bmatrix}0 & 1 & 0\end{bmatrix}^{\mathbf{T}}$; a parallel description applies to the xy- and yz-planes.

Inline Exercise 7.17: Confirm that this definition of signed area for a triangle in an arbitrary plane in 3-space agrees with our definition of signed area for a triangle in the xy-plane.

7.10.6 The Tilting Principle

One situation that arises repeatedly in computer graphics is shown in Figure 7.24: We have a triangle T' in a plane P' with unit normal $\mathbf{n}'$, and we project it to a triangle T in a plane P with unit normal $\mathbf{n}$, the projection being *along* $\mathbf{n}$. For instance, if P is the zx-plane, the projection is along the y-axis, taking (a, b, c) to $(a, 0, c)$.

The (signed) areas of T' and T are related by a cosine.

We call this *the tilting principle*.

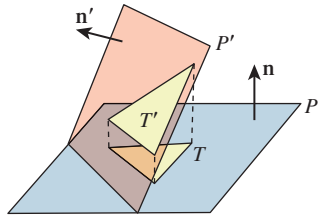


Figure 7.24: A tilted triangle and its projection.

✓ **THE TILTING PRINCIPLE:** If T' is an oriented triangle in plane P' with normal $\mathbf{n}'$, and T is its projection to plane P, the projection being along the unit normal $\mathbf{n}$ to P, then the signed area of T is $\mathbf{n}' \cdot \mathbf{n}$ times the signed area of T'.

Once we've shown that the tilting principle applies to triangles, it also will apply to a polygon and its projection: We simply triangulate the polygon, and form a corresponding triangulation on its projection, and consider the area ratios one triangle at a time.

We'll prove this claim about the ratio of signed areas for the case where the plane P is the xz-plane, and the triangle T' has vertices A', B', and C' with coordinates (a_x, a_y, a_z), etc.; the corresponding vertices of T are $A = (a_x, 0, a_z)$, etc.

Proving the principle in this special case is sufficient. We can always choose a coordinate system for space that makes the plane P be the xz-plane. Furthermore, we can rotate the coordinate system until the x-coordinate of the unit normal $\mathbf{n}'$ to T' is zero. Figure 7.25 shows this situation.

The vector $\mathbf{n}'$ has the form $\begin{bmatrix} 0 & y & z \end{bmatrix}^{\mathbf{T}}$ and length 1, so $y^2 + z^2 = 1$. Letting $\theta = \text{atan2}(y, z)$, we can write $\mathbf{n}' = \begin{bmatrix} 0 & \cos\theta & \sin\theta \end{bmatrix}^{\mathbf{T}}$. The dot product of $\mathbf{n}'$ with the xz-plane's unit normal $\mathbf{n} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^{\mathbf{T}}$ is exactly $\cos\theta$.

Let's compute the two signed areas. For T, the formula is

$$sa(T) = A_{zx} = \frac{1}{2}(a_z b_x - a_x b_z) + (b_z c_x - c_z b_x) + (c_z a_x - a_z c_x). \qquad (7.130)$$

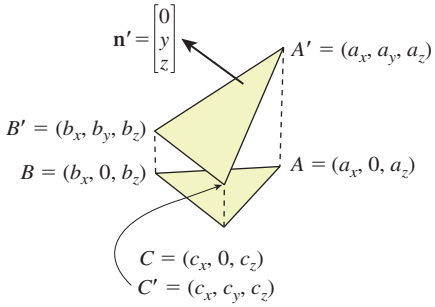


Figure 7.25: A triangle projected to the xz-plane.

For T', the (unsigned) area is the length of the vector $\mathbf{a}' = \left[A'_{yz}, A'_{zx}, A'_{xy}\right]$. Because T' is tilted only in the yz-plane, $A'_{yz} = 0$. And because the x- and z-coordinates for T and T' agree, we have

$$A'_{zx} = A_{zx}. \tag{7.131}$$

That leaves only A'_{xy} to consider:

$$A'_{xy} = (a_x b_y - a_y b_x) + (b_x c_y - c_x b_y) + (c_x a_y - a_x c_y). \tag{7.132}$$

Remembering that the normal to the plane of T is $\begin{bmatrix}0 & \cos\theta & \sin\theta\end{bmatrix}^{\mathbf{T}}$, we know the plane equation: Every point (x, y, z) on this plane must satisfy

$$0x + \cos(\theta)y + \sin(\theta)z = K \tag{7.133}$$

where K is some constant. (You should make sure that you understand why this is true.) This means that for any point (x, y, z) on the plane,

$$y = -\tan(\theta)z + \frac{K}{\cos(\theta)}. \tag{7.134}$$

Applying this to a_y, b_y, and c_y in Equation 7.132 and canceling many terms, we get

$$\begin{aligned} A'_{xy} &= (a_x \tan(\theta) b_z - \tan(\theta) a_z b_x) + (b_x \tan(\theta) c_z - c_x \tan(\theta) b_z) \\ &\quad + (c_x \tan(\theta) a_z - a_x \tan(\theta) c_z) && (7.135) \\ &= -\tan(\theta) A_{zx}. && (7.136) \end{aligned}$$

The length of $\mathbf{a}'$ (hence the area of T') is thus

$$\begin{aligned} area &= \sqrt{(A'_{yz})^2 + (A'_{zx})^2 + (A'_{xy})^2} && (7.137) \\ &= \sqrt{0^2 + A_{zx}^2 + (-\tan(\theta) A_{zx})^2} && (7.138) \\ &= \sqrt{(1 + \tan^2(\theta) A_{zx}^2} && (7.139) \\ &= \pm \sec(\theta) |A_{zx}|, && (7.140) \end{aligned}$$

while the area of T is $|A_{zx}|$. Thus, the area of T' is $|\cos\theta|$ times that of T.

There remains the question of the sign. If $\cos\theta > 0$, and the signed area of $\triangle A'B'C'$ is positive, then the vertices A, B, and C are organized counterclockwise as viewed from the tip of $\mathbf{n}$, and hence the signed area of $\triangle ABC$ is also positive. On the other hand, if $\cos\theta < 0$ and the signed area of $\triangle A'B'C'$ is positive, then the vertices A, B, and C as viewed from the tip of $\mathbf{n}$ are organized clockwise, and hence the signed area of $\triangle ABC$ is negative. If we reverse the order of A, B, and C, in both cases, both signed areas change sign. So, in all four possible cases, the ratio of unsigned areas is $|\cos\theta|$, and the sign of the ratio of signed areas is the sign of $\cos\theta$, hence the ratio of signed areas is exactly $\cos\theta = \mathbf{n} \cdot \mathbf{n}'$.

Inline Exercise 7.18: Suppose that instead of projecting T' onto the xy-plane by projecting in the y-direction, we projected in the $\mathbf{n}'$ direction. What would be the relationship between the signed area of the projected triangle T'' and that of T'?

7.10.7 Analogs of Barycentric Coordinates

Barycentric coordinates provide a very useful way to discuss point locations within a triangle, because they are **invariant under affine transformations.** That is, if the point Q has barycentric coordinates (s_0, s_1, s_2) in the triangle $P_0P_1P_2$, and we transform the triangle by applying the same transformation T to each P_i and to Q, and if T is an **affine transformation**—that is, a rotation, a translation, a dilation, or a combination of these—then the barycentric coordinates of $T(Q)$ in the triangle $T(P_0)T(P_1)T(P_2)$ will still be (s_0, s_1, s_2). Furthermore, on the edge between P_0 and P_1, we have $s_2 = 0$, and similarly for the other two edges. The barycentric coordinates are all positive for points inside the triangle, but for any point outside the triangle, at least one barycentric coordinate is negative. Is there an analogous set of "coordinates" for a point in a polygon? Warren and others have studied this question extensively, developing generalized barycentric coordinates for points in a convex polygon or set [War96] [WSHD04] and a generalization to coordinates that use all points of a mesh to define coordinates on points in and around that mesh [JSW05].

7.11 Discussion

The lessons to take away from this chapter are as follows.

- Be careful to say what you mean precisely when you use mathematics. Give your functions both domains and codomains; examine what happens when a formula could lead to a division by zero; and generalize to n dimensions whenever possible to better understand the intrinsic nature of the problem you're solving.
- Whenever possible, express computations that need to be done in $\mathbf{R^2}$ or $\mathbf{R^3}$ as vector computations, involving vector operations, point-vector combinations, point-point differences, and inner products, and thereby avoid per-coordinate expressions.
- Try to understand geometric problems geometrically rather than in terms of coordinates; use coordinates for computations only.

These approaches lead to clearer understanding and more easily maintainable programs, and can often lead to insights about algorithms because the compactness of well-expressed mathematics allows us to see patterns that might otherwise be obscure.

7.12 Exercises

Exercise 7.1: (a) Show that if s is a number and $\mathbf{v}$ is a vector, then $\|s\mathbf{v}\| = |s|\|\mathbf{v}\|$. (b) Give an example of a number s and vector $\mathbf{v}$ for which $\|s\mathbf{v}\| \neq s\|\mathbf{v}\|$.

Exercise 7.2: We showed how to get from a point P on a line and a vector $\mathbf{n}$ normal to the line to the more familiar $Ax + By + C = 0$ form. Figure out how to go the other way: Given a line defined by $Ax + By + C = 0$, with at least one of A and B nonzero, find a point on the line. Hint: A normal vector to the line is given by $\mathbf{n} = \begin{bmatrix} A \\ B \end{bmatrix}$. Use this to determine a point of the form $O + \alpha\mathbf{n}$ (where O is the origin) that lies on the line, by solving for α.

Exercise 7.3: The expression $\mathbf{u} \cdot (\times \mathbf{v})$ that arose in studying line intersections occurs quite often (as do its generalizations to higher dimensions). Show that it equals the determinant of a matrix whose first column is $\mathbf{v}$ and whose second column is $\mathbf{u}$. As a point of information, this works more generally: In 3-space, $\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w})$ is the same as the determinant of a matrix whose columns are $\mathbf{v}$, $\mathbf{w}$, and $\mathbf{u}$ and similar formulas hold in higher dimensions.

Exercise 7.4: Find the parametric form of the line between the points $(1, 1)$ and $(2, 2)$. Then find the parametric form of the line between the points $(3, 3)$ and $(5, 5)$. Are the two functions identical? Are the lines they describe identical? Explain why the mapping defined by the parametric line formula is actually a map from "two distinct points in the plane" to "a parametric representation of the line between these points," rather than from the line itself to a parametric representation.

Exercise 7.5: The same reasoning as in Exercise 7.4 applies to the implicit form of a line: Depending on the points chosen, we get different "standard implicit forms." Fortunately, all define the same line. Furthermore, any two standard implicit forms are proportional. Write down the implicit form of the line between the points $(1, 1)$ and $(2, 2)$. Then find the implicit form of the line between the points $(3, 3)$ and $(5, 5)$. Are the two implicit functions identical? Are they proportional? Are the points where they are zero identical?

Exercise 7.6: Since any two standard implicit forms of a line are proportional, is there a way to choose a "standard representative" once and for all? Suppose that $Ax + By + C$ and $A'x + B'y + C' = 0$ describe the same line. Then we know that the triples (A, B, C) and (A', B', C') are proportional, and that any other triple proportional to these (except (0,0,0)) will determine the same line. Can we choose just *one* and call it the "normal form of the line"? For instance, we might say, "Take your triple (A, B, C) and convert to a normal form where $B = 1$ by dividing through by B to get $(A/B, 1, C/B)$." Unfortunately, this doesn't work if $B = 0$; the same goes for A and C. We *can* take the triple (A, B, C) and divide by $\sqrt{A^2 + B^2 + C^2}$ to get a "normal form"; if we do this, the only ambiguity is a sign: The standard form for (A, B, C) and the one for $(-A, -B, -C)$, which determine the same line, end up being opposites. Explain why (a) if $\lambda \neq 0$, the lines determined by (A, B, C) and $(\lambda A, \lambda B, \lambda C)$ are identical; (b) if $\lambda > 0$, they have the same normal form; and (c) the factor $\sqrt{A^2 + B^2 + C^2}$ is never zero when $Ax + By + C = 0$ determines a line.

Exercise 7.7: Barycentric coordinates can be used to describe lines within a triangle PQR. For instance, all points on the line PQ satisfy the equation $\gamma = 0$ (where α, β, and γ are the barycentric coordinates). Let S be a point that's one-third of the way from P to Q, and consider the line passing through S and R. What equation, in barycentric coordinates, determines this line? (Hint: Draw a picture, and find the barycentric coordinates of at least two points on the line.)

Exercise 7.8: The inequality $4x+2y-6 \geq 0$ defines a half-plane; its boundary is the line ℓ consisting of solutions to $4x + 2y - 6 = 0$. Find the point of ℓ that's on the x-axis, and the point of ℓ that's on the y-axis. Is the origin in the half-space defined by the inequality? Draw the normal ray for the equation of ℓ, and verify that it points toward the positive half-space, as described in Section 7.9.3.

Exercise 7.9: Generalize to 3-space the result about normal vectors and the defining equation for half-planes: that for the half-plane defined by $Ax+By+C \geq 0$, the vector $\begin{bmatrix} A \\ B \end{bmatrix}$ points from the edge of the half-plane into the positive half-plane.

◈ **Exercise 7.10:** The winding number of a parametric curve γ about a point z_0 of the complex plane is defined as

$$n(z_0, \gamma) = \frac{1}{2\pi i} \int_\gamma \frac{dz}{z - z_0}. \tag{7.141}$$

Show that by parameterizing each edge of the polygon with vertices $P_1, \ldots, P_n$ with an interval of length 1, and treating the point with coordinates (x, y) as the complex number $x + iy$, we can reduce this integral definition to the simplified one we gave for polygons.

Exercise 7.11: (a) Show that the normal vector to a triangle with vertices P_0, P_1, P_2 computed by Plücker's method is indeed perpendicular to $P_1 - P_0$; a similar computation works to show it's perpendicular to $P_2 - P_0$.
(b) Verify that for $P_0 = (0, 0, 0)$, $P_1 = (1, 0, 0)$, and $P_2 = (0, 1, 0)$, the normal computed by Plücker's method points along the positive z-axis so that the vectors $P_2 - P_0$, $P_1 - P_0$, and $\mathbf{n}$ form a right-handed coordinate system.
◈ (c) Explain why the conclusion of part (b) holds regardless of the location of P_0, P_1, and P_2, as long as they do not lie on a single line.

Exercise 7.12: Let $P_0P_1P_2$ be a triangle in 3-space, and $\mathbf{n}$ its Plücker normal. Thinking of $\mathbf{n}$ as a covector, consider the function $\mathbf{v} \mapsto \mathbf{n} \cdot \mathbf{v}$. What is its value on vectors lying in the plane of the triangle?

Exercise 7.13: The inside-outside test for polygons based on ray intersections depends on the ray intersecting each edge in a single point. It also depends on the ray not passing through any polygon vertex, because if it did so, the intersections with both of the edges at the vertex would be counted. How can we avoid these problems?
(a) A randomized algorithm, in which the ray direction is chosen randomly, will fail with probability zero (assuming we are using infinite-precision numbers). If the chosen ray is a failure case, we can randomly choose a new one, and with probability one the algorithm will finish.
(b) We can find the set of all direction vectors of all edges in the polygon, and of all rays from the test point to all polygon vertices, and choose a direction which is different from all of these. Explain why, in the case of a very small quadrilateral, with vertices at $(\pm\epsilon, 0)$ and $(0, \pm\epsilon)$, where ϵ is the smallest floating-point number representable on the computer, and with a test point Q at the origin, both of these approaches fail in practice. Will the more complex sum-of-inverse-cosines formula for computing the winding number work in this case?

Exercise 7.14: (To do this exercise, you'll need to know something about transformations of the plane; these will be covered in Chapter 10.)
(a) Show that the area formula in Equation 7.122 is correct when P_1, P_2, and Q do not lie on a line, and when P_1 is not on the y-axis, by doing two transformations: Shear in y to move P_1 to the x-axis; then shear in x to move P_2 to the y-axis. Show that shearing doesn't change the computed area; Cavalieri's principle shows that it doesn't change the actual area.
(b) Show that the formula is right for the two remaining cases by presenting an argument for each. The formula for area is a continuous function of the coordinates of P_1, P_2, and Q. The actual area is *also* a continuous function of these coordinates. At this point we have shown that these two functions agree almost everywhere (e.g., whenever the three points do not lie on a line). By a continuity argument, they must therefore agree everywhere.

◈ (c) Show that the formula is correct by arguing that area is a continuous function of coordinates, and that there is, at most, one extension of a continuous function on X to a continuous function on $X \cup \text{boundary}(X)$ when X is a subset of R^n.

Exercise 7.15: Another approach to generalizing the area formula from the "one vertex at the origin" case to the general case is this: Compute the signed area of the triangle $P_0P_1P_2$ by computing the signed areas of QP_0P_1, QP_1P_2, and QP_2P_0 and then adding or subtracting appropriately. Draw a picture to figure out what the relationship among the four signed areas should be, and use it to derive the general formula for the signed area of $P_0P_1P_2$.

Exercise 7.16: Numerical considerations: In the formula for the area of a polygon, suppose that we are using finite precision arithmetic and we add L, where L is a very large number, to the x-coordinates of all the polygon's vertices. What happens to the computation? What if we instead perform the computation by writing the coordinates of the vertices in a coordinate system based at the "center" of the polygon—the average of all the vertices?

Exercise 7.17: Consider a ray that starts at $P = (-3, -3)$ and has direction $\mathbf{d} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$. It intersects the ellipse defined by $(\frac{x}{3})^2 + y^2 = 1$ at two points. To find these points, we can write

$$R(t) = P + t\mathbf{d} \tag{7.142}$$

$$R(t)^{\mathbf{T}} \begin{bmatrix} 1/3 & 0 \\ 0 & 1 \end{bmatrix} R(t) = 1 \tag{7.143}$$

where the $^{\mathbf{T}}$ indicates transpose. If we solve the second equation for t, we will have found the parameters t_1 and t_2 of the intersection points, from which we can compute the points.

(a) Draw a picture representing this situation.

(b) Confirm that the equations above really *do* determine the points of intersection by writing out the product explicitly.

(c) Now consider the intersection of the ray defined by the point $Q = (-1, -3)$ and the direction $\mathbf{e} = \begin{bmatrix} 1/3 \\ 2 \end{bmatrix}$, with the unit circle. Once again, draw a picture and express this problem as a similar pair of equations; the matrix in this case will be the identity matrix.

(d) Expand these latter equations; compare them to the ones you arrived at in part (b).

(e) Explain the similarity: How are $\mathbf{d}$ and $\mathbf{e}$ related? How are the ellipse and the unit circle related? We'll return to this kind of transformation from a general problem (compute an intersection with an ellipse) to an equivalent standard problem (compute an intersection of a different ray with the unit circle) when we study ray tracing.

Exercise 7.18: The function $\gamma : \mathbf{R} \to \mathbf{R^2} : t \mapsto (3 + 2t, 4 - 3t)$ describes a line in parametric form.

(a) Find two distinct points P and Q on the line. (There are an infinite number of correct answers to this part.)

(b) Use these two to find an implicit form for the line; convert via algebra to the form $Ax + By + C = 0$.

Exercise 7.19: (a) You're given two nondegenerate line segments in the plane, the first with endpoints A and B and the second with endpoints C and D, and all coordinates are integers. Your job is to determine whether the segments intersect, and the intersection point, if any.
(a) Write a short program to do this. If the segments are parallel, there are three cases: They're disjoint (return `false`), they share an endpoint (return `false`), or they overlap in an interval (return `true`, but don't return an intersection point, since it's not unique). If the segments are nonparallel and intersect, they may share an endpoint (return `false`), the endpoint of one may be interior to the other (return `true`, and the endpoint), or the interiors of the segments may intersect at some point (x, y) whose coordinates may not be integers, but will be rational numbers. In this case, you should return an *integer triple* (x, y, w), where the intersection is at $(x/w, y/w)$.
(b) Explain why returning an integer triple is more useful than returning a floating-point representation of the rational coordinates.
(c) Suppose you have two integer triples of the form above, (x_1, y_1, w_1) and (x_2, y_2, w_2). How would you test them for "equality," that is, how would you test whether they represent the same rational point in the plane?
(d) For an extra challenge, try to write all your code in part (a) so that there are no divisions, and the code is as clean as possible. The sort of multiple-case mess that's implicit in programs like this arises from the multiple possible ways that segments can intersect; you can't really make the code *too* clean.

Chapter 8

A Simple Way to Describe Shape in 2D and 3D

8.1 Introduction

We now turn to a discussion of the **triangle mesh,** the most widely used representation of shape in graphics. Triangle meshes consist of many triangles joined along their edges to form a surface (see Figure 8.1). Other meshes, in which the basic elements are quads (quadrilaterals), or other polygons are sometimes used, but there can be problems associated with them. For instance, it's easy to create a quadrilateral whose four vertices do not all lie in a plane; how should the interior be filled in? For triangles, this is not a problem: There's always a plane containing any three vertices. Because triangle meshes are so widespread, we concentrate on them in this chapter.

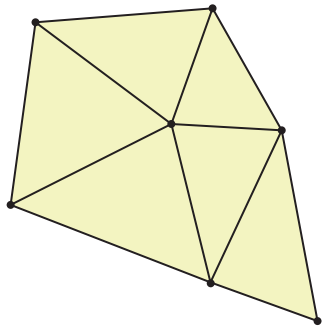


Figure 8.1: A triangle mesh, consisting of vertices, edges, and triangular faces.

It's easy to see how to create certain shapes with triangle meshes. Starting with any polyhedron, for instance, we can subdivide the faces into triangles. Figure 8.2 shows this for the cube. For more complex shapes, it's possible to *approximate* the shape with a mesh. One way to do this is to find the locations of many points on the shape, and then connect adjacent locations with a mesh structure. Such an approximation, if the points are close enough to one another, can look very much like a smooth surface. Consider the case of the icosahedron, which looks a lot like a sphere: Each point of the icosahedral mesh is very close to a point of the sphere, and vice versa. Similarly, each normal vector to a triangle mesh is very close to a vector normal to the sphere at a corresponding point, and vice versa. There is a distinction, however: The function that assigns normal vectors to points of the sphere is continuous, while for the icosahedron, it's **piecewise constant** (the normal vector doesn't change as you move about on a triangular facet). This distinction can be important when we try to consider the reflection of light from surfaces described with planar facets.

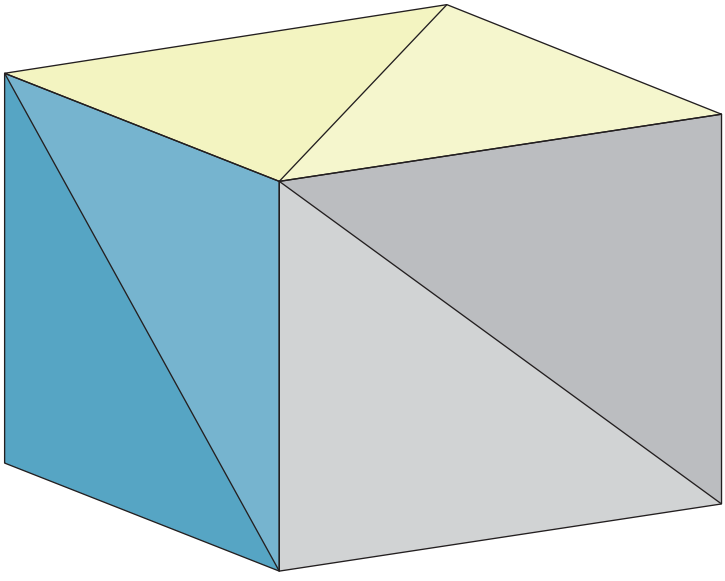


Figure 8.2: A triangle mesh that has the geometry of a cube.

One of the nicest properties of triangle meshes is their uniformity. This uniformity allows us to apply various operations to them with guarantees that have relatively simple proofs; it also makes it easy to try simple ideas. Among the most interesting operations we can perform on a mesh is **subdivision,** in which a single triangle is replaced by several smaller triangles in a fairly simple way. (There are many subdivision algorithms, some of which we'll discuss in Chapter 22.) Typically subdivision is used to smooth out a mesh that has sharp points or edges, so as to approach a limit surface that's fairly smooth. Of course, repeated subdivision operations increase the triangle count substantially; this can have a major impact on rendering performance.

Another important operation on meshes is **simplification,** in which a mesh is replaced by another mesh that's similar to it, topologically or geometrically, but has a more compact structure. If this is done repeatedly, one can arrive at a collection of simpler and simpler representations of the same surface, suitable for viewing at greater and greater distances. (A $10,000$-polygon object that covers only a single display pixel can almost certainly be rendered with *fewer* polygons, for instance—a simplified mesh is ideal here.) Hoppe [Hop96, Hop98] has studied this problem extensively.

Meshes are in common use in part because we are familiar with the geometry of triangles. Not every object in the world is well suited to mesh representation. Certain shapes, for instance, are characterized by having geometric detail at every scale (e.g., mica or fractured marble). Others have structure that is uniform in a way particularly unsuited for mesh representation, like hair, whose bent tubular structure can be far more compactly represented than with a mesh approximation.

Nonetheless, many research laboratories and commercial companies have managed to produce a great many successful images using an approach in which all shapes were approximated by triangle meshes.

8.2 “Meshes” in 2D: Polylines

The analog of a triangle mesh in space, taken one dimension lower, is a collection of line segments in the plane. (The space in which we work is one dimension lower, and the objects we’re working with are one dimension lower: line segments instead of triangles.) We’ll discuss these briefly as an introduction to mesh structures. We’ll call one of these a 1D mesh.

A 1D mesh (see Figure 8.3) consists of **vertices** and **edges,** which are line segments joining the vertices. Because the line segment between two vertices is completely determined by the vertices themselves, we can describe such a structure in two parts.

- A listing of the vertices and their locations. Typically the vertices are denoted by small integers; their locations are points in the plane.
- A listing of the edges, consisting of a collection of ordered pairs of vertices.

The following tables describe a simple 1D mesh:

Vertices	
1	(0, 0)
2	(0.5, 0)
3	(1.5, 1)
4	(0, 2.0)
5	(3, 0)
6	(4, 0)

Edges	
1	(1, 2)
2	(2, 3)
3	(3, 4)
4	(4, 1)
5	(5, 6)

This data structure has an interesting property: The **topology** of the mesh (which edges meet which other edges) is encoded in the Edges table, while the geometry is encoded in the Vertices table. If you adjusted one of the entries in the Vertices table a little, the number of connected components, for instance, would not vary.

One might argue that if you moved the vertices enough, then two edges might intersect when they didn’t intersect before. That’s true, but such intersections can be removed by adjusting the vertices; the fact that the edge $(1, 2)$ intersects the edge $(2, 3)$ cannot be altered by moving the vertices.

Indeed, one can treat the edge table (together with a listing of the vertex indices, in case some vertex is not in any edge) as describing an abstract graph (in the sense of graph theory). From this one can compute things like the Euler characteristic, the number of components, etc.

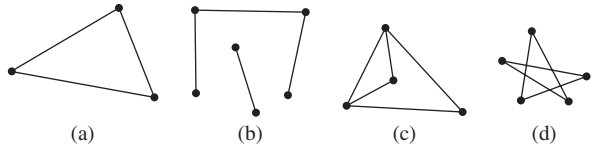


Figure 8.3: A 1D mesh consists of a collection of vertices and straight-line edges between them. The ones that most often interest us (shown in (a) and (b)) have one or two edges meeting each vertex, and no two edges intersect except at a vertex. But there are also meshes with more than two edges at some vertices, like (c), and ones where edges do intersect at nonvertex points (d).

8.2.1 Boundaries

The **boundary** of a 1D mesh defined as a *formal sum* of the vertices of the mesh in which the coefficient of each vertex is determined as follows: Each edge from vertex i to vertex j adds $+1$ to the coefficient for j, and -1 to the coefficient for i; we sometimes write that the boundary of the edge ij is $j - i$. Applying this to the mesh above, we find that the boundary formal sum is (reading edge by edge, and writing v_i for the ith vertex)

$$(v_2 - v_1) + (v_3 - v_2) + (v_4 - v_3) + (v_1 - v_4) + (v_6 - v_5), \tag{8.1}$$

which simplifies to $v_6 - v_5$. Informally, we say that the boundary consists of vertices 5 and 6.

The reason for the formalism arises when we consider more interesting meshes, like the one shown in Figure 8.4. The boundary in this case consists of $v_1 + v_2 + v_3 + v_4 + v_5 - 5v_0$.

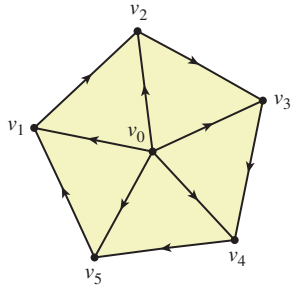


Figure 8.4: A wagon-wheel-shaped mesh. An arrow from vertex i to vertex j indicates that (i, j) is an edge of the mesh, and not (j, i).

A 1D mesh whose boundary is zero (i.e., the formal sum in which all coefficients are zero) has the property that it's easy to define "inside" and "outside" by a rule like the winding number rule for polygons in the plane. Such a mesh is called **closed.**

A 1D mesh where each vertex has degree 2 (i.e., where each vertex has an arriving edge and a leaving edge) is said to be a **manifold mesh:** In the abstract graph, every point has a neighborhood (a set of all points sufficiently near it) that resembles a small part of the real number line. A point in the interior of an edge, for example, has the edge interior as such a neighborhood. A vertex has the union of the interiors of the two adjacent edges, together with the vertex itself, as such a neighborhood.

We use the term "manifold mesh" to suggest that such meshes are like *manifolds,* which we will not formally define; there are many books that introduce the idea of manifolds with the appropriate supporting mathematics [dC76, GP10]. Informally, however, an n-dimensional manifold is an object M with the property that for any point $p \in M$, there's a neighborhood of p (i.e., a set of all points in M close to p, defined appropriately) that looks like the set $\{\mathbf{x} \in \mathbf{R}^n : \|\mathbf{x}\| < 1\}$ (the "open ball") in $\mathbf{R}^n$. "Looks like" means that there's a continuous map from the ball to the neighborhood and back. (These continuous maps are also required to be "consistent" with one another wherever their domains overlap; the precise details are beyond the scope of this book.) For example, the unit circle in the plane is a 1-manifold because one neighborhood of the point with angle coordinate θ consists of all points with coordinates $\theta - 0.1$ to $\theta + 0.1$; the correspondence to the unit ball in $\mathbf{R}$ (i.e., the open interval $-1 < x < 1$) is $u \mapsto 10(u - \theta)$. Similarly, familiar smooth surfaces in 3-space like the sphere, or the surface of a donut, are 2-manifolds. An atlas (i.e., a book showing maps of the whole world) is a kind of demonstration that the sphere is a manifold: Each page of the atlas gives a correspondence between some region of the globe (e.g., Western Europe) and a portion of the plane (i.e., the page of the atlas that shows Western Europe).

Shapes with corners (like a cube) can also be manifolds, but they are not *smooth* manifolds, which is what's usually meant when the term is used informally—a continuous map that takes a small region around the corner of the cube and sends it to the plane ends up distorting things too much for all the required conditions for "smoothness" to hold.

Self-intersecting shapes like a figure eight in the plane fail to be manifolds, because any small neighborhood of the crossing point in the middle of the figure eight looks like a small letter “x,” which cannot be made to correspond to a unit interval in a bicontinuous way. The technicalities in defining manifolds are quite subtle (indeed, it took several decades for them to eventually settle into their modern form). Fortunately for us, the shapes we use in graphics are generally “polyhedral manifolds.” The definitions are still subtle, but there are key theorems that tell us that in the one- and two-dimensional cases, we can instead verify that something’s a manifold by using far simpler methods. Those simpler methods are precisely the content of our notions of a manifold vertex-and-edge mesh, and a manifold triangle mesh (which we’ll define shortly).

Your goal, in reading the definitions here, should not be to gain a deep understanding that you could use to prove theorems—that requires a much more thorough treatment. But you should, when done, be able to say, “Sure, I can look at a simple mesh and identify it as a manifold mesh, or perhaps manifold-with-boundary mesh.”

Manifold meshes are commonplace and particularly easy to work with (and to prove theorems about). Note that the definition does not say that a manifold mesh must have only one connected component; indeed, a mesh consisting of two nonintersecting triangles is a valid manifold mesh. So is the empty mesh.

The meshes we’ve described, in which each edge is an *ordered* pair, are called **oriented meshes;** if we had described edges by unordered pairs, we would have had an **unoriented mesh,** in which case the definition of “boundary” would have made no sense. We’ll have no further use for such meshes, however.

8.2.2 A Data Structure for 1D Meshes

To prepare for the discussion of 2D meshes in 3-space, we’ll describe a data structure for 1D meshes in 2-space first, one which has strong analogies with more complex structures. The components are

- A vertex table, consisting of vertex indices and the associated points in the plane
- An edge table, consisting of ordered pairs of edges
- A **neighbor-list table,** consisting of an ordered cyclic list of edges meeting a vertex so that one can read off the list of edges at a vertex (in counterclockwise order)

We already encountered such a structure when we discussed subdivision of curves (see Figure 8.5) in Chapter 4.

The operations supported by this data structure (and their implementations) are as follows.

- Insert a vertex: Add it to the vertex table; leave other tables untouched.
- Insert an edge (i,j): Add it to the vertex table and the edge table (both $O(1)$); add it to the neighbor-list table both at vertex i and at vertex j. Insertion in the neighbor list for vertex i can take $O(e)$ time, where e is the number of edges in the table (one must insert it in the right place in the counterclockwise ordering of edges around vertex i, which might contain all e edges). In a *manifold* mesh, however, where there can be, at most, two edges at a vertex, this operation is $O(1)$.

- Get the edges meeting vertex i. (This is $O(1)$, since it consists of the neighbor list for vertex i.)
- Given a vertex i and an edge e containing i, find the other end of e.
- Given an edge e, find its two endpoints.
- Delete an edge. This requires deletion from the edge table, and if the edge is (i,j), deleting the edge from the neighbor lists for i and j, which may be an $O(e)$ operation, but in the manifold case is $O(1)$.

The choice of how to implement the vertex and edge tables depends on the expected use. An array implementation is convenient if there will be no deletions. But if there *will* be deletions, one must do one of the following.

- Mark array entries as invalid somehow.
- Shift the array contents to "fill in" when an item is deleted (which requires updating indices stored in other tables).

The marked-entries approach can create large but mostly empty tables if there are many insertions and deletions so that the "list all vertices" or "list all edges" operations become slow. The content-shifting approach actually works quite well, however. To start with a simple case, if there are n vertices and we want to delete vertex n, we just declare the end of the array to be the $n-1$st element, and delete all references to vertex n in other tables. To delete a different vertex—say, the second one—we reduce the problem to the earlier case: We exchange the second and the nth vertices, and then delete the nth. This requires replacing all references to vertex index 2 with vertex index n, and vice versa, in the other tables, but that's fairly straightforward.

Note that the choice to store the edges that meet a vertex is also application-dependent. It makes finding all vertices of topological distance one from vertex i very fast, but at the cost of making edge addition somewhat slow in the worst case. If you do not anticipate needing to find the neighbors of vertex i, maintaining a neighbor list is pointless. Similarly, the choice to store the neighbor lists in a counterclockwise-sorted order is useful primarily if one is interested not only in the structure of the 1D mesh, but also in the structure of the 2D regions into which it divides the plane; if these are of no interest, then the neighbor lists can be stored in hash tables or other similarly efficient structures (or in a two-element array, in the case of manifold meshes).

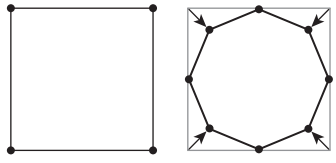


Figure 8.5: The square at left is subdivided to become the octagon at right. Note that for each vertex of the square, there's a vertex in the octagon, and for each edge *of the square, its midpoint is a vertex of the octagon as well.*

8.3 Meshes in 3D

The situation in 3D is analogous to that in 2D: To describe a mesh, we list the vertices and the triangles of the mesh. What about the edges? The tradition in graphics has been to infer the edges from the triangles: If vertices i, j, and k form a triangle, then the edges (i,j), (j,k), and (k,i) are assumed to be part of the mesh structure. This means that one cannot have dangling edges (see Figure 8.6), although isolated vertices are still allowed. The general descriptions of mesh structures (consisting of vertices, edges, triangles, tetrahedra, etc., with coordinates associated only to the vertices, and then extended to higher-dimensional pieces by interpolation) have been at the foundation of topology for more than 100 years [Spa66]; the student interested in such structures should consult the topology literature to avoid reinventing things.

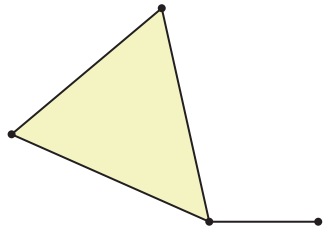


Figure 8.6: A triangle with a dangling edge, like the one shown here, cannot be represented by our mesh structure.

In graphics, however, the structure of a vertex table and "face table" or "triangle table" is well established. As in the 1D case, insertion of vertices and triangles is fast, and deletion of vertices is slow (because all associated triangles must be found and deleted). If we store a neighbor list for each vertex, deletion becomes faster. Because the neighbor list of triangles meeting a vertex is unordered, the insertion cost is low, but the deletion cost is high. When additional constraints are imposed on the mesh, the triangle list for a vertex can be ordered, slightly increasing insertion and deletion costs, but simplifying other operations like finding the two faces adjacent to an edge.

What about edges? While we cannot insert an edge, we can ask questions like "Is this pair of vertices an edge of the mesh?" In other words, is it the edge of some triangle of the mesh? And given a triangle with vertices i, j, and k, we can ask, "What are the other triangles that contain the edge (i,j)? These questions are $O(T)$, in the sense that answering them requires an exhaustive search of the triangle list. In special cases, which we discuss in the next section, they can be made faster.

If you are eager, at this point, to get on with making objects and pictures of objects, you can safely skip the remainder of this chapter and use the vertex- and triangle-table structure for meshes until you encounter problems with space or efficiency, at which point the remaining sections will be of use to you. If, on the other hand, you'd like to know more about how to work with meshes effectively, read on.

8.3.1 Manifold Meshes

A finite 2D mesh is a **manifold mesh** if the edges and triangles meeting a vertex v can be arranged in a cyclic order $t_1, e_1, t_2, e_2, \ldots, t_n, e_n$ without repetitions such that edge e_i is an edge of triangles t_i and t_{i+1} (indices taken mod n). This implies that for each edge, there are exactly two faces that contain it.

We can store a manifold mesh in a data structure analogous to the one we described for 1D meshes, consisting of a vertex table, a triangle table, and a neighbor-list table.

The neighbor list for vertex i consists of the triangles surrounding vertex i, in some cyclic order (so the kth and $k+1$st triangles in the list share an edge). (One can no longer disambiguate between the two possible cyclic orderings around a vertex with a notion like "counterclockwise," unfortunately, unless the manifold is *oriented,* which we describe in the next section.)

Manifold meshes unfortunately don't admit insertions or deletions of triangles: Any insertion or deletion ruins the manifold property. But it *is* easy to find the vertices adjacent to a given vertex (i.e., given a vertex index i, we can find all vertex indices j such that (i,j) is an edge): We simply take the set of all vertices of all triangles in the neighbor list for vertex i, and then remove vertex i.

It's also fairly easy, given a triangle containing edge (i,j), to find the other triangle containing that edge.

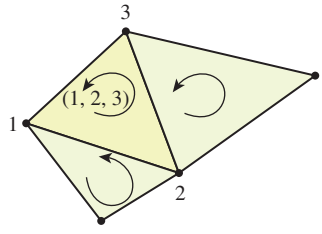


Figure 8.7: Two adjacent triangles in a mesh with consistent normal vectors (i.e., the normal vector tips are on the "same side" of the mesh). Note that the edge (i,j) is an edge of one triangle, but (j,i) is an edge of the other. In general, in a consistently oriented mesh, each edge appears twice, in opposite directions.

8.3.1.1 Orientation

We'll often have reason to care about the *orientation* of triangles in a mesh (see Figure 8.8) so that the triangles $(1,2,3)$ and $(2,1,3)$ are considered different (the triples are listings of vertex indices). One use of an orientation is in the determination of a *normal vector:* If the vertices of a nondegenerate triangle

(i.e., one with a nonzero area) are at locations P_i, P_j, and P_k, then we can compute $(P_j - P_i) \times (P_k - P_i)$, which is a vector perpendicular to the plane of the triangle.[1] Note that if we exchange vertices P_j and P_k, the resultant vector is negated. Since we often use the normal to a triangle in a mesh to determine what's "inside" or "outside" the mesh, the ordering of the vertices is critical.

If two adjacent triangles (see Figure 8.7) have consistently oriented normal vectors, then the edge they share will appear as (i,j) in one triangle and as (j,i) in the other. And if a manifold mesh can have its triangles oriented so that this is true, it can also be given consistently oriented normal vectors. This is a nontrivial theorem from combinatorial topology.

When a manifold is oriented, the triangles around a vertex have a natural order. Suppose that surrounding vertex 1 there are the triangles $(5,1,2)$, $(4,3,1)$, $(1,5,4)$, and $(1,3,2)$. We can cyclically permute each to put vertex 1 first: $(1,2,5)$, $(1,4,3)$, $(1,5,4)$, and $(1,3,2)$. Now starting with the first triangle, we can consider its "first" and "last" edges, $(1,2)$ and $(5,1)$. We choose, as our next triangle, the one whose first edge is the opposite, $(1,5)$, of this last edge. That's $(1,5,4)$. And the last edge of $(1,5,4)$ is $(4,1)$; $(1,4)$ is the first edge of $(1,4,3)$, whose last edge is $(3,1)$; $(3,1)$ is the first edge of $(1,3,2)$, and we've ordered the triangles: $(1,2,5)$, $(1,5,4)$, $(1,4,3)$, $(1,3,2)$.

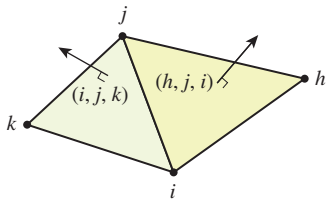


Figure 8.8: The triangles of this mesh are oriented*; the circular arrows indicate the cyclic ordering of the vertices. Note that the vertex triples* $(1,2,3)$ *and* $(2,3,1)$ *indicate the same oriented triangle (i.e., there are three equivalent descriptions of every oriented triangle).*

8.3.1.2 Boundaries

More interesting than manifold meshes (and more common as well) are meshes whose vertices are manifold or **boundarylike** (see Figure 8.9), in the sense that instead of the adjacent triangles forming a cycle, they form a chain, whose first and last elements share only one of their edges with other triangles in the chain; the *other* edge of the first triangle that meets the vertex is contained *only* in the first triangle, and not in any other triangle of the mesh. (The same condition applies to the last triangle.) This unshared edge is called a **boundary edge,** and the vertex is called a **boundary vertex.** Vertices that are not boundary vertices are called **interior vertices.**

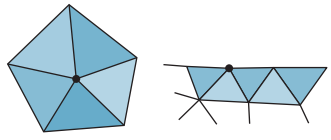


Figure 8.9: (left) A manifold vertex has a cycle of triangles around it; (right) by contrast, a boundarylike vertex v has a chain of triangles around it—the starting and ending triangles of the chain each share only one of their edges with other triangles of the chain. The other edges of those triangles that contain v are boundary edges.

8.3.1.3 Boundaries and Oriented 2D Meshes

Just as we defined the boundary of an edge from vertex i to vertex j to be the formal sum $v_j - v_i$, we can define the boundary of a triangle in a mesh with vertices i, j, and k to consist of the formal sum of edges

$$(i,j) + (j,k) + (k,i). \tag{8.2}$$

Furthermore, we can define an algebra on these formal sums in which the vertex (i,j) is identified with $-1(j,i)$ so that the boundary above could be written

$$(i,j) + (j,k) - (i,k) \tag{8.3}$$

instead. We can define the boundary of a collection of oriented triangles as the formal sum of their boundaries.

For an oriented manifold mesh, this boundary will be zero (i.e., the coefficient of each edge will be zero), because if (i,j) is part of the boundary of one face, then $(j,i) = -(i,j)$ is part of the boundary of another.

1. The description of the cross product and further discussion of normal vectors was given in Chapter 7.

For an oriented manifold-with-boundary mesh, the boundary will consist of exactly the edges we identified above as "boundary edges." In general, an oriented mesh with no boundary edges is called **closed.**

8.3.1.4 Operations on Manifold-with-Boundary Meshes

Manifold-with-boundary meshes support operations like vertex and face insertion and deletion. The efficiency of each operation depends on the implementation. If the representation is simply vertex tables and face tables, then insertion is $O(1)$, and deletion (after the "exchange with the last item in the list" trick) is too. If we maintain a neighbor list for each vertex, then insertion becomes $O(T)$, where T is the number of triangles, as does deletion.

Note that computing the boundary of such a mesh can be done in $O(T)$ time. (Use a hash table to count the number of times each edge appears, with sign. If the edge appears zero times, delete it from the hash table.) But one can also maintain a record of the boundary during insertions and deletions so that reporting it at any time is an $O(1)$ operation.

8.3.2 Nonmanifold Meshes

Just as in the 1D case, we sometimes encounter shapes that are not well represented by manifolds or manifolds with boundary. The two cubes shown in Figure 8.10 share a vertex which is a nonmanifold vertex; two cubes sharing an edge are similarly nonmanifold. There is an important difference between the two cases, however: It's easy to encounter a nonmanifold vertex in the course of constructing a manifold with boundary (see Figure 8.11). But once we construct a nonmanifold edge (one with three or more faces meeting it), it can never become a manifold edge through further additions.

Each vertex in the **directed-edge** structure also contains a reference to one of the edges containing it (see Figure 8.12). This allows one to compute the neighborhood of the vertex (all edges and triangles that meet it) in time proportional to the size of the neighborhood.

Campagna et al. [CKS98] show how this structure can be extended to handle non-manifold vertices and edges, and how to trade time for space by simplifying the structure for very large meshes.

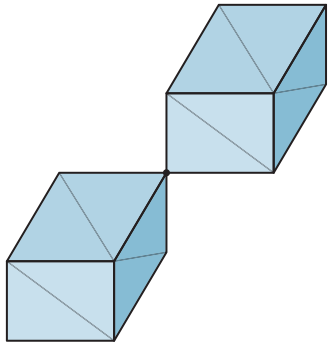


Figure 8.10: The shared vertex is nonmanifold: No neighborhood looks planar.

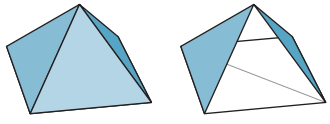


Figure 8.11: The pyramid at left has six faces; the bottom square is divided into two triangles that you cannot see. If we construct the pyramid so that after four triangles have been added, it appears as shown at the right, then the apex vertex is neither an interior vertex nor a boundary vertex according to the definitions, so this shape is neither a manifold nor a manifold with boundary. Once we add another face, it becomes a manifold with boundary, and when we add the last face, it becomes a manifold.

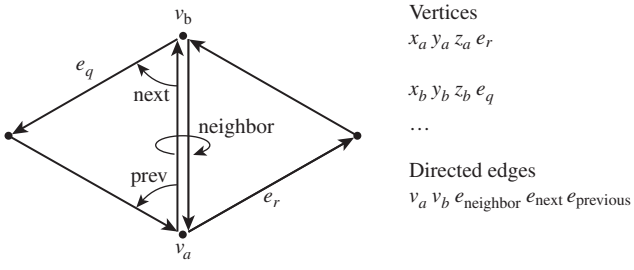


Figure 8.12: The directed-edge data structure (following Figures 4 and 5 of [CKS98]). A directed edge stores a reference to its starting and ending vertex, to the previous and next edges, and to its neighbor. For each real edge of the mesh, there are two directed edges, in opposite directions. Each vertex stores its coordinates and a reference to one of the directed edges that leaves it.

For more general planar meshes, in which the faces may not be triangles, one can use the **winged-edge** data structure [Bau72], which associates to each edge of the mesh the next edge of the face to its right, the next edge of the face to its left, and the previous edges of each of these as well (see Figure 8.13). This suffices to recover all the faces and edges, provided that all faces are simply connected (i.e., no face has a ringlike shape, like the surface of a moat). The winged-edge structure also stores, for each vertex, its *xyz*-coordinates, and a reference to one of the edges that it lies on (from which one can find all other edges). For each face, the structure stores a reference to one edge of the face (from which one can find all the others).

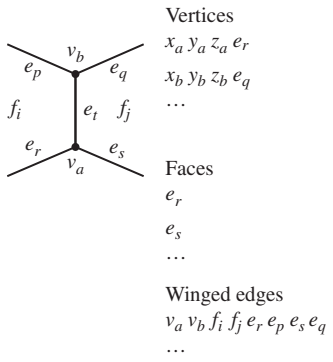


Figure 8.13: The winged-edge data structure records a "previous" and "next" pointer for the faces on each side of each edge.

8.3.3 Memory Requirements for Mesh Structures

Each of the structures we've described for representing meshes has certain memory requirements. Assuming 4-byte floating-point representations and 4-byte integers, we can compare their memory requirements as done by Campagna et al. [CKS98].

The vertex-table-and-triangle-table approach takes $12V$ bytes for the V vertices and $12T$ bytes for the T triangles. How are the number of vertices and triangles related? For a mesh representing a closed surface, Euler's formula tells us that $V - E + F = 2 - 2g$, where g is the **genus** of the surface.[2] Assuming further that every vertex is actually part of some triangle (so that the vertex table does not contain lots of unused vertices), and that the mesh is closed, we can simplify this: Each triangle has three edges, and each edge is shared by two triangles. So the number of edges, E, is $\frac{3}{2}T$. Thus,

$$V - \frac{3}{2}T + T = 2 - 2g, \tag{8.4}$$

which simplifies to

$$V - \frac{1}{2}T = 2 - 2g. \tag{8.5}$$

For low-genus surfaces with fine tessellations, the right-hand side is negligible compared to the left, so we find that the number of triangles is approximately twice the number of vertices; this gets us a total of $12(T + V) \approx 12(3V) = 36V \approx 18T$ bytes of storage. For the remaining mesh representations, we'll assume that we're representing closed manifolds of low genus so that we can replace V with $\frac{T}{2}$ and vice versa.

For the winged-edge structure, each vertex uses 16 bytes (three floats and an edge reference); each face uses four bytes (one edge reference), and each edge has four edge references, two face references, and two vertex references, for a total of eight references or 32 bytes. The total, again assuming we use *triangular faces only,* is $16V + 32E + 4T \approx 8T + 32\frac{3}{2}T + 4T = 60T$.

For the directed-edge data structure (in the store-all-references form), each vertex uses 12 bytes for coordinates and four for an edge reference. Each directed

2. The genus of a closed surface is, informally, the number of holes in it. A sphere has genus zero, a torus has genus one, a two-holed torus has genus two, etc. A slice of Swiss cheese tends to have quite high genus.

edge contains references to two vertices and three edges, using 20 bytes. Triangular faces are not explicitly represented. So the memory use is

$$16V + 40E \approx 8T + 60T = 68T \tag{8.6}$$

bytes. Note that in the analysis above, we assumed that a vertex or edge reference required only a single byte; for more complex meshes, this byte count may have to to be increased to roughly $\lceil \log_2(\frac{3T}{2}) \rceil$.

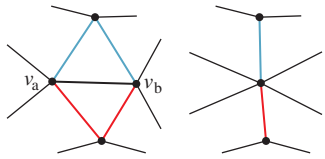


Figure 8.14: The edge from vertex a to vertex b is collapsed; the edge itself and the two adjacent faces are removed from the data structure; and the other two edges of the upper face, drawn in blue, become one, as do the two edges of the lower face, drawn in red. Nothing else changes. The two vertices $\mathbf{v}_a$ and $\mathbf{v}_b$ become a single vertex.

8.3.4 A Few Mesh Operations

One of the advantages of triangle meshes is that their homogeneity makes certain operations easy to perform. For manifold meshes, the homogeneity is even greater. In mesh simplification, for instance, one of the standard operations is an **edge collapse,** in which one edge is shrunk until it has length zero, resulting in the two adjacent triangles disappearing. In mesh **beautification** (where we try to make a mesh have nearly equilateral triangles and other nice properties), the **edge-swap** operation helps turn two long and skinny triangles into two more nearly equilateral ones. Both involve minimal operations on the data structure itself.

8.3.5 Edge Collapse

In the **edge-collapse** operation (see Figure 8.14), a single edge of a mesh is removed [HDD+93]. The two triangles that contain this edge are both eliminated, and the other two edges of each of them become a single edge in the new mesh. The vertices at the end of the eliminated segment become a single vertex.

The description above is purely topological; there's a geometric question as well: When we merge the two vertices, we must choose a *location* for the merged vertex. The location we choose depends on the goal of our simplification (see Figure 8.15). If computation is at a premium, simply using one of the old vertices as the new one is very fast. If we want to preserve some sort of shape, averaging the two vertices is easy. If such an averaging process moves a lot of points, and this will be visually distracting, we can choose a new location that minimizes the average or extreme distance between the old and new meshes. There is no one "right answer." As in most of graphics, the choice you make depends on your intended use of the data structure.

Figure 8.15: Different geometric choices for an edge collapse in 2D. The edge AB is collapsed; one can (a) place the collapsed vertex at A for computational simplicity, (b) at the midpoint of the segment AB, or (c) at a point Q that minimizes the maximum (or average) distance from every old mesh point to the nearest new mesh point. Other goals are possible as well.

8.3.6 Edge Swap

Meshes that get distorted or deformed in the course of an application's use of them may eventually get so deformed that individual triangles are long and skinny. Such triangles are characterized by their bad *aspect ratios*. In general, one can define an **aspect ratio** for a planar shape (see Figure 8.16) by finding, among all rectangles that enclose the object and touch it on all four sides (these are called **bounding boxes**), the one whose length-to-width ratio is greatest. This ratio is then called the aspect ratio. High-aspect-ratio triangles produce bad artifacts in many situations, so it's nice to be able to eliminate them when possible. An **edge-swap** operation (see Figure 8.17) can convert two adjacent high-aspect-ratio triangles to two with lower aspect ratios. (It can also, done in reverse, do the opposite: Selecting the right edge to swap in order to beautify a mesh requires examining the impact of each possible swap.)

Notice that the swap removes two triangles from the mesh structure and replaces them with two others. In the simple vertex- and triangle-table structure, this operation is trivial. In the directed-edge structure, the implementation is more complex, because (a) some vertex may point to the edge that's being swapped out, and this edge pointer needs to be found and replaced; and (b) it makes sense to replace the two old triangles with the two new ones, but there's substantial shuffling of directed edges in this process, and making sure that the new directed edges point to the correct new triangles is messy.

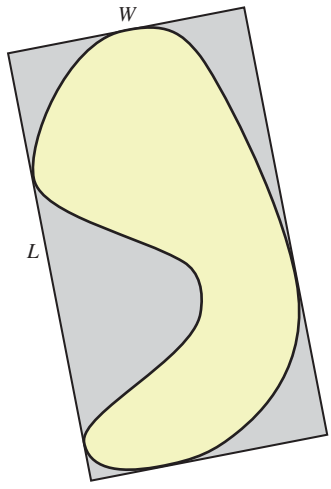


Figure 8.16: The aspect ratio of a planar shape is determined by finding the bounding box for the object for which the ratio of length to width is greatest.

8.4 Discussion and Further Reading

Triangle mesh representations and other representations for nontriangle meshes, for planar graphs, and for **simplicial complices**—assemblies of vertices, edges, triangles, tetrahedra, etc.—are widely studied in areas other than graphics; each representation is tuned to the application area. We've described a few representations here that are particularly suited to work in graphics, but those who develop CAD programs, for instance, may have to deal with computing the union and intersections of shapes represented by meshes. Unfortunately, the union of two manifold meshes (e.g., two cubes) may not be a manifold mesh (if the cubes share just one vertex, or just one edge), so structures suitable for nonmanifold representations may be essential. Those working in finite-element modeling of mechanical structures or fluid flow have their own constraints, such as the need for triangles and tetrahedra to be nicely shaped (no small angles in any triangles), or to have their size vary depending on the region in which they lie (e.g., turbulent flow may require a fine triangulation, while smooth flow may be adequately represented by a coarse one).

For most elementary graphics, the vertex- and triangle-table representations are adequate; their incredible simplicity makes them very versatile, and if you're implementing your own mesh structures, they're very easy to program properly. As your needs evolve, more complex structures may be suitable; be certain that you evaluate the more complex structures to ensure that their complexity solves your particular problem.

One structure we've completely ignored is the "list each triangle separately as a triple of *xyz*-coordinates structure." Although this is simple, it has so many disadvantages that we can never recommend its use. In particular, the only way to tell whether two triangles or edges share a vertex is with floating-point comparisons: If you move one copy of a vertex, you must move all others if you want to preserve the adjacency structure of the mesh; and determining any sort of adjacency information is, in general, $O(T)$.

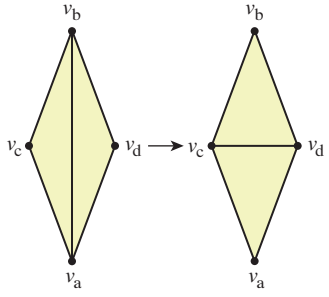


Figure 8.17: The triangles adjacent to the edge from $\mathbf{v}_a$ to $\mathbf{v}_b$ both have bad aspect ratios. By replacing the edge $\mathbf{v}_a\mathbf{v}_b$ with the edge $\mathbf{v}_c\mathbf{v}_d$, we get a new pair of triangles whose aspect ratios are better.

8.5 Exercises

Exercise 8.1: Suppose you know that triangle (i, j, k) is one of the triangles of a manifold mesh that's represented by a vertex table and a face table. Then edge (i, j) is in the mesh. Describe how to find the other triangle containing edge (i, j). Express the running time of this operation in terms of T, the number of triangles in the mesh. Draw an exemplar class of meshes that shows that the upper bound you found is actually realized in practice.

Exercise 8.2: Implement a form of the 1D mesh structure that's suitable for implementing a *subdivision* operation on *manifold* meshes. Given such a mesh, M, the associated subdivision mesh M' has one vertex for each vertex v of M: If the neighbors of v are u and w, the new vertex is at location $\frac{\alpha}{2}u + \frac{\alpha}{2}w + (1-\alpha)v$. M' also has one vertex for each *edge* of M: If the edge is between vertices t and u, the associated vertex of M' is at $\frac{1}{2}(t+u)$. Vertices in M' are connected by an edge if their associated vertices and/or edges in M meet (i.e., the vertex associated to the edge from u to v is connected to the vertex associated to u and to the vertex associated to v). Figure 8.5 shows an example. The parameter α determines the nature of the subdivision. The example shown in the figure uses $\alpha = 0.5$; on repeated subdivision, the square becomes a smoother and smoother curve. What happens for other values of α? Use the standard 2D test bed to make a program with which you can experiment. Note: Not every manifold mesh has just a single connected component.

Exercise 8.3: Draw examples to show how adding a triangle to a mesh can cause each of the four changes described in the nonmanifold vertex representation.

Exercise 8.4: Write pseudocode explaining how, given a vertex reference in a directed-edge data structure, to determine the list of all directed edges leaving that vertex in time proportional to the output size.

Exercise 8.5: Explain, in pseudocode, how, given a reference to a face in the winged-edge data structure, one can find all the edges of the face.

Exercise 8.6: Suppose that M is a connected manifold mesh with no boundary. M may be *orientable* without being *oriented:* It's possible that there's a consistent orientation of the faces of M, but that some faces are oriented inconsistently.
(a) Assuming that M is connected, describe an algorithm, based on depth-first search, for determining whether M is orientable.
(b) If M *is* orientable, explain why there are, at most, two possible orientations. Hint: Your algorithm may show why, once a single triangle orientation is chosen, all other triangle orientations are determined.
(c) If M is orientable, explain why there are exactly two orientations of M.
(d) Now suppose that M is not connected, but has $k \geq 2$ components. How many orientations of M are there?

Exercise 8.7: The 2D test bed was designed to aid in the study of things like meshes. Use it to build a program for drawing polylines in a 2D plane, getting mouse clicks, and reporting the closest vertex to a click (perhaps by changing the color of the vertex).

Exercise 8.8: Add a feature so that a shift-click on a vertex initializes edge drawing: The starting vertex is highlighted, and the next vertex clicked is connected to the starting vertex with an edge; if there's already an edge between the two, it should be deleted. And if the next click is *not* on a vertex, a vertex is created there and an edge from the preselected vertex to the new one is added. Modify the program to handle 2D meshes (i.e., vertices and *triangles*) by allowing the user to control-click on three vertices to create a triangle (or delete it if it already exists).

This page intentionally left blank

Chapter 9

Functions on Meshes

9.1 Introduction

In mathematics, functions are often described by an algebraic expression, like $f(x) = x^2 + 1$. Sometimes, on the other hand, they're **tabulated,** that is, the values for each possible argument are listed, as in

$$f : \{1, 2, 3\} \to \{0, 9\}; \tag{9.1}$$

$$f(1) = f(2) = 0; f(3) = 9. \tag{9.2}$$

A third, and very common, way to describe a function is to give its values at particular points and tell how to *interpolate* between these known values. For instance, we might plot the temperature at noon and midnight of each day of a week; such a plot consists of 15 distinct dots (see Figure 9.1). But we could also make a guess about the temperatures at times between each of these, saying, for instance, that if it was 60° at noon and 24° at midnight, that drop of 36° took place at a steady rate of 3° per hour. In other words, we would be **linearly interpolating** to define the function for *all* times rather than just at noon and midnight each day. The resultant function, defined on the whole week rather than just the 15 special times, is a connect-the-dots version of the original.

Let's now write that out in equations. Suppose that $t_0 < t_1 < t_2 < \ldots < t_n$ are the times at which the temperature is known, and that $f_0, f_1, \ldots, f_n$ are the temperatures in degrees Fahrenheit at those times.

Then

$$f : [t_0, t_n] \to \mathbf{R} : t \mapsto (1 - s)f_i + sf_{i+1} \tag{9.3}$$

where

$$t_i \leq t \leq t_{i+1} \quad \text{and} \tag{9.4}$$

$$s = \frac{t - t_i}{t_{i+1} - t_i}. \tag{9.5}$$

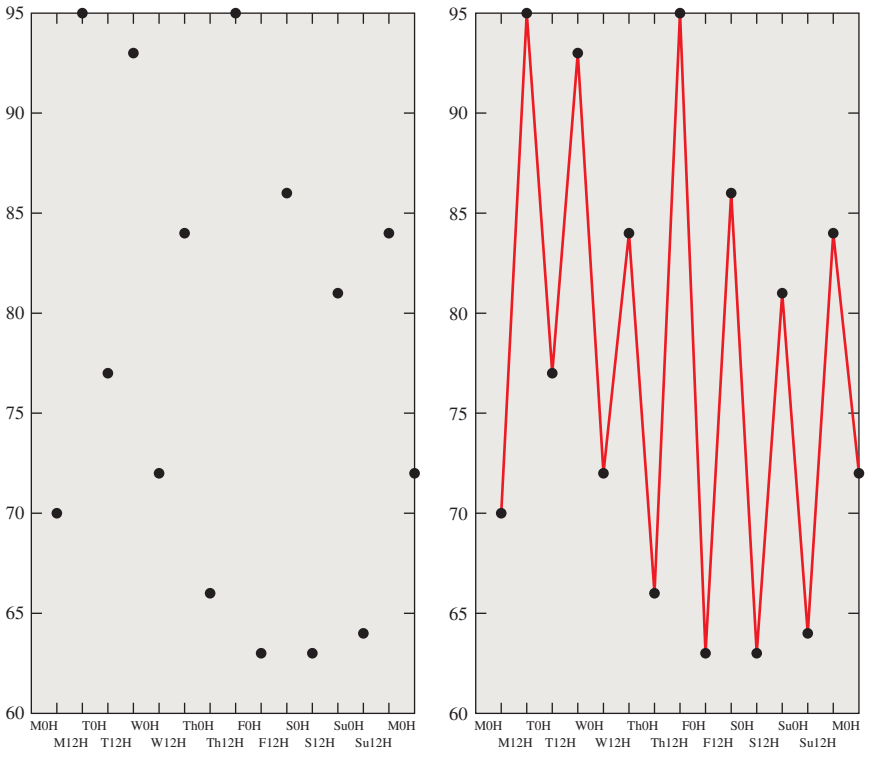


Figure 9.1: The temperature at noon and midnight for each day in a week. The domain of this function consists of 15 points. When we linearly interpolate the values between pairs of adjacent points, we get a function defined on the whole week—a connect-the-dots version of the original.

Further suppose that the times are measured in 12-hour units, starting at midnight Sunday, so $t_0 = 0, t_1 = 1$, etc.

In this formulation i is the index of the interval containing t, and s describes the fraction of the way from t_i to t_{i+1} where t lies (when $t = t_i$, s is zero; when $t = t_{i+1}$, s is one).

Inline Exercise 9.1: Suppose that $t_0 = 0, t_1 = 1$, etc., and that $f_0 = 7, f_1 = 3$, and $f_2 = 4$; evaluate $f(1.2)$ by hand. If we changed f_0 to 9, would it change the value you computed? Why or why not?

Just as we can have barycentric coordinates on a triangle (see Section 7.9.1), we can place them on an interval $[p, q]$ as well. The first coordinate varies from one at p to zero at q; the second varies from zero at p to one at q. Their sum is **everywhere one** that is, for every point of the interval $[p, q]$, the sum is one (see Exercise 9.8). With these barycentric coordinates, we can write a slightly more symmetric version of the formula above:

$$f(t) = c_0(t) f_i + c_1(t) f_{i+1}, \tag{9.6}$$

where $c_0(t)$ is the first barycentric coordinate of t and $c_1(t)$ is the other.

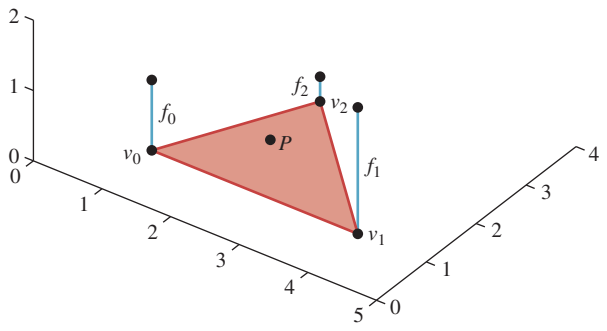


Figure 9.2: The point P is in the triangle with vertices v_0, v_1, and v_2, where the function has values f_0, f_1, and f_2, respectively. What value should we assign to the point P?

This "continuous extension" of f has analogs for other situations. Before we look at those, let's examine some of the properties. First, the value of the interpolated function at noon or midnight each day (t_i) is just f_i; it doesn't depend at all on the values at noon or midnight on the other days. Second, the value at noon on one day influences the shape of the graph only for the 12 hours before and the 12 hours after. Third, the interpolated function is in fact continuous.

Now let's consider interpolating values given at discrete points on a surface. Since we often use triangle meshes in graphics to represent surfaces, let's suppose that we have a function whose values are known at the vertices of the mesh; the value at vertex i is f_i. How can we "fill in" values at the other points of all the triangles in the mesh?

By analogy, we use barycentric coordinates. Consider a point P in a triangle with vertices v_0, v_1, and v_2 (see Figure 9.2); we can define

$$f(P) = c_0 f_0 + c_1 f_1 + c_2 f_2 \tag{9.7}$$

where (c_0, c_1, c_2) are the barycentric coordinates of P with respect to v_0, v_1, and v_2.

Once again, the interpolated function has several nice properties. First, the value at the vertex v_i is just f_i; the value along the edge from v_i to v_j (assuming they are adjacent) depends only on f_i and f_j; thus, for a point q on such an edge, it doesn't matter which of the two triangles sharing the edge from v_i to v_j is used to compute $f(q)$—the answer will be the same! Second, the value f_i at vertex v_i again influences other values only **locally,** that is, only on triangles that contain the vertex v_i. Third, the interpolated function is in fact continuous.

The remainder of this chapter investigates this idea of interpolating across faces, its relationship to barycentric coordinates, and some applications.

9.2 Code for Barycentric Interpolation

The discussion so far has been somewhat abstract; we'll now write some code to implement these ideas. Let's start with a simple task.

Input:

- A triangle mesh in the form of an $n \times 3$ table, `vtable`, of vertices
- A $k \times 3$ table, `ftable`, of triangular faces, where each row of `ftable` contains three indices into `vtable`

- An $n \times 1$ table, `fntable`, of function values at the vertices of the mesh
- A point P of the mesh, expressed by giving the index, t, of the triangle in which the point lies, and its barycentric coordinates α, β, γ in that triangle

Output:

- The value of the interpolated function at P

The first thing to realize is that only the tth row of `ftable` is relevant to our problem: The point P lies in the tth triangle; the other triangles might as well not exist. If the tth triangle has vertex indices $i0$, $i1$, and $i2$, then only those entries in `vtable` matter. With this in mind, our code is quite simple:

```
1  double meshinterp(double[,] vtable, int[,] ftable,
2    double[] fntable, int t, double alpha, double beta, double gamma)
3  {
4    int i0 = ftable[t, 0];
5    int i1 = ftable[t, 1];
6    int i2 = ftable[t, 2];
7    double fn0 = fntable[i0];
8    double fn1 = fntable[i1];
9    double fn2 = fntable[i2];
10   return alpha*fn0 + beta*fn1 + gamma*fn2;
11 }
```

Now suppose that P is given differently: We are given the coordinates of P in 3-space rather than the barycentric coordinates, and we are given the index t of the triangle to which P belongs, and we need to find the barycentric coordinates α, β, and γ. If we say that the vertices of the triangle t are A, B, and C, we want to have

$$\alpha A_x + \beta B_x + \gamma C_x = P_x \tag{9.8}$$

where the subscript x indicates the first coordinate of a point; we must also satisfy the same equations for y and z. But there's one more equation that has to hold: $\alpha + \beta + \gamma = 1$. We can rewrite that in a form that's analogous to the others:

$$\alpha 1 + \beta 1 + \gamma 1 = 1. \tag{9.9}$$

Now our system of equations becomes

$$\begin{bmatrix} A_x & B_x & C_x \\ A_y & B_y & C_y \\ A_z & B_z & C_z \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}. \tag{9.10}$$

There's really no solution here except to directly solve the system of equations. The problem is that we have four equations in three unknowns, and most solvers want to work with *square* matrices rather than rectangular ones. (Section 7.9.2 presented an alternative approach to this problem using some precomputation, but that precomputation amounts to doing much of the work of solving the system of equations.)

The good news is that the four equations are in fact redundant: The fact that P is *given* to us as a point of the triangle ensures that if we simply solve the first three equations, the fourth will hold. That assurance, however, is purely mathematical—computationally, it may happen that small errors creep in. There are several viable approaches.

- Express P as a convex combination of *four* points of $\mathbf{R}^4$: the three already written and a fourth, with coordinates n_x, n_y, n_z, and 0, where $\mathbf{n}$ is the normal vector to the triangle. The expression will have a fourth coefficient, δ, in the solution, representing the degree to which P is *not* in the plane of A, B, and C. We ignore this and scale up the computed α, β, and γ accordingly, using $\alpha/(1-\delta), \beta/(1-\delta)$, and $\gamma/(1-\delta)$ as the barycentric coordinates. This is a good solution (in the sense that if the numerical error in P is entirely in the $\mathbf{n}$ direction, the method produces the correct result), but it requires solving a 4×4 system of equations.
- Delete the fourth row of the system in Equation 9.10; adjust α, β, and γ to sum to one by dividing each by $\alpha + \beta + \gamma$. This reduces the problem to a 3×3 system, but it lacks the promise of correctness for normal-only errors.
- Use the pseudoinverse to solve the overconstrained system (see Section 10.3.9). This has the advantage that it's already part of many numerical linear algebra systems, and that it works even when the triangle is degenerate (i.e., the three points are collinear), in the sense that *if* P lies in the triangle, then the method produces numbers α, β, and γ with $\alpha A + \beta B + \gamma C = P$, even though the solution in this case is not unique. Better still, if P is not in the plane of A, B, and C, the solution returned will have the property that $\alpha A + \beta B + \gamma C$ is the point in the plane of A, B, and C that's closest to P. This is therefore the ideal.

So the restated problem looks like this.
Input:

- A triangle mesh in the form of an $n \times 3$ table, `vtable`, of vertices
- A $k \times 3$ table, `ftable`, of triangles, where each row of `ftable` contains three indices into `vtable`
- A point P of the mesh, expressed by giving the index, t, of the triangle in which the point lies, and its coordinates in 3-space

Output:

- The value of the barycentric coordinates of P with respect to the vertices of the kth triangle

And our revised solution is this:

```
1  double[3] barycentricCoordinates(double[,] vtable,
2     int[,] ftable, int t, double p[3])
3  {
4     int i0 = ftable[t, 0];
5     int i1 = ftable[t, 1];
6     int i2 = ftable[t, 2];
7     double[,] m = new double[4, 3];
8     for (int j = 0; j < 3; j++) {
9        for (int i = 0; i < 3; i++) {
10         m[i,j] = vtable[ftable[t, j], i];
11       }
12       m[3,j] = 1;
13    }
14
15    k = pseudoInverse(m);
16    return matrixVectorProduct(k, p);
17 }
```

Here we've assumed the existence of a matrix-vector product procedure and a pseudoinverse procedure, as provided by most numerical packages.

The two procedures above can be combined, of course, to produce the function value at a point P that's specified in xyz-coordinates.

Input:

- A triangle mesh in the form of an $n \times 3$ table, `vtable`, of vertices
- A $k \times 3$ table, `ftable`, of triangles, where each row of `ftable` contains three indices into `vtable`
- An $n \times 1$ table, `fntable`, of function values at the vertices of the mesh
- A point P of the mesh, expressed by giving the index, t, of the triangle in which the point lies, and its coordinates in 3-space

Output:

- The value of the function defined by the function table at the point P

```
1 double meshinterp2(double[,] vtable, int[,] ftable, double[] fntable,
2 int t, double p[3])
3 {
4   double[] barycentricCoords =
5      barycentricCoordinates(vtable, ftable, t, p);
6   return meshinterp2(vtable, ftable, fntable, t,
7      barycentricCoords[0], barycentricCoords[1], barycentricCoords[2]);
8 }
```

Of course, the same idea can be applied to the ray-intersect-triangle code of Section 7.9.2, where we first computed the barycentric coordinates (α, β, γ) of the ray-triangle intersection point Q with respect to the triangle ABC, and then computed the point Q itself as the barycentric weighted average of A, B, and C. If instead we had function values f_A,f_B, and f_C at those points, we could have computed the value at Q as $f_Q = \alpha f_A + \beta f_B + \gamma f_C$. Notice that this means we can compute the interpolated function value f_Q at the intersection point without ever computing the intersection point itself!

Computing barycentric coordinates (α, β, γ) for a point P of a triangle ABC, where $A, B, C \in \mathbf{R^2}$, is somewhat simpler than the corresponding problem in 3-space (see Figure 9.3). We know that the lines of constant α are parallel to BC. If we let $\mathbf{n} = (C - B)^{\perp}$, then the function defined by $f(P) = (P - B) \cdot \mathbf{n}$ is *also* constant on lines parallel to $B - C$. Scaling this down by $f(A)$ gives us the function we need: It's zero on line BC, and it's one at A. So we let

$$g : \mathbf{R^2} \to \mathbf{R} : P \mapsto \frac{(P - B) \cdot \mathbf{n}}{(A - B) \cdot \mathbf{n}}, \tag{9.11}$$

and the value of $g(P)$ is just α. A similar computation works for β and γ.

The resultant code looks like this:

```
1 double[3] barycenter2D(Point P, Point A, Point B, Point C)
2 C[2])
3 {
4   double[] result = new double[3];
5   result[0] = helper(P, A, B, C);
6   result[1] = helper(P, B, C, A);
7   result[2] = helper(P, C, A, B);
8   return result;
```

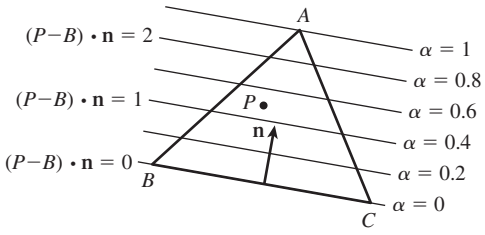


Figure 9.3: To write the point P as $\alpha A + \beta B + \gamma C$, we can use a trick. For points on line BC, we know $\alpha = 0$; on any line parallel to BC, α is also constant. We can compute the projection of $P - B$ onto the vector **n** *that's perpendicular to BC; this also gives a linear function that's constant on lines parallel to BC. If we scale this function so that its value at A is* 1*, we must have the function α.*

```
9  double helper(Point P, Point A, Point B, Point C)
10 {
11    Vector n = C - B;
12    double t = n.X;
13    n.X = -n.Y; // rotate C-B counterclockwise 90 degrees
14    n.Y = t;
15    return dot(P - B, n) / dot(A - B, n);
16 }
```

Of course, if the triangle is degenerate (e.g., if A lies on line BC), then the dot product in the denominator of the helper procedure will be zero; then again, in this situation the barycentric coordinates are not well defined. In production code, one needs to check for such cases; it would be typical, in such a case, to express P as a convex combination of two of the three vertices.

9.2.1 A Different View of Linear Interpolation

One way to understand the interpolated function is to realize that the interpolation process is *linear*. Suppose we have two sets of values, $\{f_i\}$ and $\{g_i\}$, associated to the vertices, and we interpolate them with functions F and G on the whole mesh. If we now try to interpolate the values $\{f_i + g_i\}$, the resultant function will equal $F + G$. That is to say, we can regard barycentric interpolation on the mesh as a function from "sets of vertex values" to "continuous functions on the mesh." Supposing there are n vertices, this gives a function

$$I : \mathbf{R}^n \to C(M) \tag{9.12}$$

where $C(M)$ is the set of all continuous functions on the mesh M. What we've just said is that

$$I(f + g) = I(f) + I(g) \tag{9.13}$$

where f denotes the set of values $\{f_1, f_2, \ldots, f_n\}$, and similarly for g; the other linearity rule—that $I(\alpha f) = \alpha I(f)$ for any real number α—also holds.

A good way to understand a linear function is to examine what it does to a *basis*. The standard basis for $\mathbf{R}^n$ consists of elements that are all zero except for a single entry that's one. Each such basis vector corresponds to interpolating a function that's zero at all vertices except one—say, v—and is one at v (see Figure 9.4).

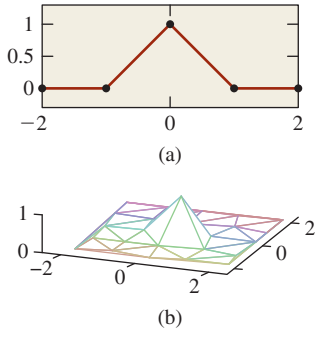


Figure 9.4: (a) The 2D interpolating basis function is tent-shaped near its center; (b) in 3D, for a mesh in the xy-plane, we can graph the function in z and again see a tentlike graph that drops off to zero by the time we reach any triangle that does not contain v.

The resultant interpolant is a **basis function** that has a graph that looks tentlike, with the peak of the tent sitting above the vertex where the value is one.

If we add up all these basis functions, the result is the function that interpolates the values $1, 1, \ldots, 1$ at all the vertices; this turns out to be the constant function 1. Why? Because *the barycentric coordinates of every point in every triangle sum to one.*

One might look at these tent-shaped functions and complain that they're not very continuous in the sense that they are continuous but not differentiable. Wouldn't it be nicer to use basis functions that looked like the ones in Figure 9.5? It would, but it turns out to be more difficult to have *both* the smoothness property *and* the property that the interpolant for the "all ones" set of values is the constant function 1. We'll discuss this further in Chapter 22.

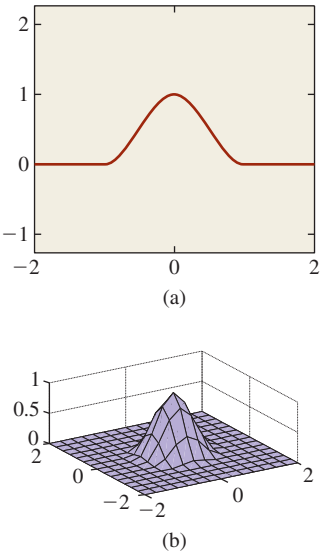


Figure 9.5: Basis functions for (a) 2D and (b) 3D interpolations that are smoother than the barycentric-interpolation functions.

9.2.1.1 Terminology for Meshes

This section introduces a few terms that are useful in discussing meshes. First, the vertices, edges, and faces of a mesh are all called **simplices**. Simplices come in categories: A vertex is a 0-simplex, an edge is a 1-simplex, and a face is a 2-simplex. Simplices contain their boundaries, so a 2-simplex in a mesh contains its three edges and a 1-simplex contains its two endpoints.

The **star** of a vertex (see Figure 9.6) is the set of triangles that contain that vertex. More generally, the star of a simplex is the set of all simplices that contain it.

The boundary of the star of a vertex is called the **link** of the vertex. This is useful in describing functions like the tent functions above: We can say that the tent has value 1 at the vertex v, has nonzero values only on the star of v, and is zero on the link of v.

There's a notion of "distance" in a mesh based on edge paths between vertices: The distance from v to w is the smallest number of edges in any chain of edges from v to w. Thus, all the vertices in the link of v have a mesh distance of one from v.

The sets of vertices at various distances from v have names as well. The 1-ring is the set of vertices whose distance from v is one or less; the 2-ring is the set of vertices whose distance from v is two or less, etc.

9.2.2 Scanline Interpolation

Frequently in graphics we need to compute some value at each point of a triangle; for example, we often compute an RGB color triple at each vertex of a triangle, and then interpolate the results over the interior (perhaps because doing the

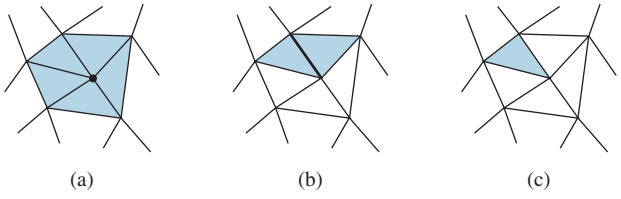


Figure 9.6: The star of a simplex. (a) The star of a vertex is the set of triangles containing it, (b) the star of an edge is the two triangles containing it, and (c) the star of a triangle is the triangle itself.

computation that produced the RGB triple at the vertices would be prohibitive if carried out for every interior point).

In the 1980s, when raster graphics (pixel-based graphics) technology was new, **scanline rendering** was popular. In scanline rendering, each horizontal line of the screen was treated individually, and all the triangles that met a line were processed to produce a row of pixel values, after which the renderer moved on to the next line. Usually the new scanline intersected many of the same triangles as the previous one, and so lots of data reuse was possible. Figure 9.7 shows a typical situation: At row 3, only the orange pixel meets the triangle; at row 4, the two blue pixels do. At row 6, the four gray pixels meet the triangle, and after that the intersecting span begins to shrink.

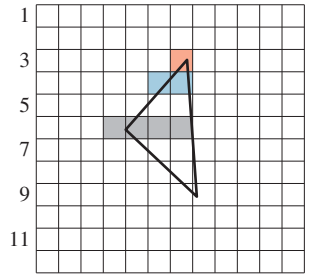


Figure 9.7: The processing of a single triangle by a scanline renderer.

One scheme that was used for interpolating RGB triples was to interpolate the vertex values along each edge of the triangle, and then to interpolate these across each scanline.

It's not difficult to show that this results in the same interpolant as the barycentric method we described (see Exercise 9.4).

But now suppose that we apply this method to a more interesting shape, like a quadrilateral. It's easy to see that the two congruent squares in Figure 9.8, with gray values of $0, 40, 0$, and 40 (in a range of 0 to 40) assigned to their vertices in clockwise order, lead to different interpolated values for the points P and P', the first being 20 and the second being 40.

The interpolated values in each configuration look decent, but when one makes an animation by rotating a shape, the interior coloring or shading seems to "swim" in a way that's very distracting.

What went wrong?

The *problem* was to infer values at the interior points of a polygon, given values at the vertices. But the *solution* depends on something unrelated to the problem, namely the scanlines in the output; this dependence shows up as an artifact in the results. The difficulty is that the solution is not based on the mathematics and physics of the problem, but rather on the computational constraints on the *solution*. When you limit the class of solutions that you're willing to examine *a priori,* there's always a chance that the best solution will be excluded. Of course, sometimes there's a good reason to constrain the allowable solutions, but in

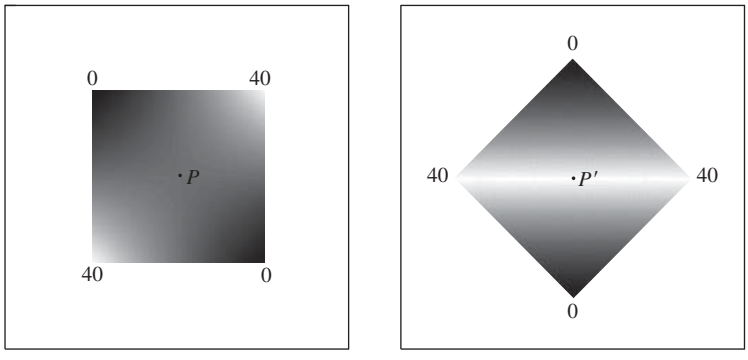


Figure 9.8: The congruent squares in these two renderings have the same gray values at corresponding vertices, but the scanline-interpolated gray values at P and P' differ.

doing so, you should be aware of the consequences. We embody these ideas in a principle:

> ✓ **THE DIVISION OF MODELING PRINCIPLE:** Separate the mathematical and/or physical model of a phenomenon from the numerical model used to represent it.

Attempting to computationally solve a problem often involves three separate choices. The first is an *understanding* of the problem; the second is a choice of *mathematical tools*; the third is a choice of *computational method*. For example, in trying to model ocean waves, we have to first look at what's known about them. There are large rolling waves and there are waves that crest and break. The first step is to decide which of these we want to simulate. Suppose we choose to simulate rolling (nonbreaking) waves. Then we can represent the surface of the water with a function, $y = f(t, x, z)$, expressing the height of the water at the point (x, z) at time t. Books on oceanography give differential equations describing how f changes over time. When it comes to *solving* the differential equation, there are many possible approaches. Finite-element methods, finite-difference methods, spectral methods, and many others can be used. If we choose, for example, to represent f as a sum of products of sines and cosines of x and z, then solving the differential equations becomes a process of solving systems of equations in the coefficients in the sums. If we limit our attention to a finite number of terms, then this problem becomes tractable.

But having made this choice, we can see certain influences: Because there's a highest-frequency sine or cosine in our sum, there's a smallest possible wavelength that can be represented. If we want our model to contain ripples smaller than this, the numerical choice prevents it. And if, having solved the problem, we decide we'd like to make waves that actually crest and break, our mathematical choice precludes it. We can, of course, add a "breaking wave" adjustment to the output of our model, altering the shapes of certain wave crests, but it should come as no surprise that such an ad hoc addition is more likely to produce problems than good results. Furthermore, it will make debugging of our system almost impossible, because without a clear notion of the "right solution," it's very difficult to detect an error conclusively.

The idea of carefully structuring your models and separating the mathematical from the numerical is discussed at length by Barzel [Bar92].

9.3 Limitations of Piecewise Linear Extension

The method of extending a function on a triangle mesh from vertices to the interiors of triangles is called **piecewise linear extension.** By looking at the basis functions—the tent-shaped graphs—we can see that the graphs of such extensions will have sharp corners and edges. Depending on the application, these artifacts may be significant. For instance, if we interpolate gray values across a triangle mesh, the human eye tends to notice the second-order discontinuities at triangle edges: When the gray values change linearly across the triangle interiors, things look fine; when the rate of change changes at a triangle edge, the eye picks it out.

Some people tend to see "bands" near such discontinuities; the effect is called *Mach banding* (see Section 1.7).

If we use piecewise linear interpolation in animation, having computed the positions of objects at certain "key" times, then between these times objects move with constant velocities, and hence zero acceleration; all the acceleration is concentrated in the key moments. This can be very distracting.

9.3.1 Dependence on Mesh Structure

If we have a polyhedral shape with nontriangular faces, we can triangulate each face to get a triangle mesh. Then we can, as before, interpolate function values at the vertices over the triangular faces. But the results of this interpolation can vary wildly depending on the particular triangulation. It's easiest to see this with a very simple example (see Figure 9.9) in which a function defined on the corners of a square is extended to the interior in two different ways. The results are evidently triangulation-dependent.

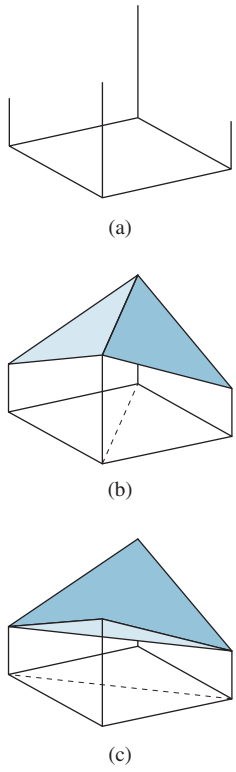


Figure 9.9: (a) A square with heights assigned at the four corners; (b) one piecewise linear interpolation of these values; and (c) a different interpolation of the same values.

9.4 Smoother Extensions

As we hinted above, taking function values at the vertices of a mesh and trying to find *smoothly* interpolated values over the interior of the mesh is a difficult task. Part of the difficulty arises in defining what it means to be a smooth function on a mesh. If the mesh happens to lie in the *xy*-plane, it's easy enough: We can use the ordinary definition of smoothness (existence of various derivatives) on the plane. But when the mesh is simply a polyhedral surface in 3-space (e.g., like a dodecahedron), it's no longer clear how to measure smoothness.

Of course, if we replace the dodecahedron with the sphere that passes through its vertices, then defining smoothness is once again relatively easy. Each point of the dodecahedron corresponds to a point on the surrounding sphere (e.g., by radial projection), and we can declare a function that's smooth on the sphere to be smooth on the dodecahedron as well. Unfortunately, finding a smooth shape that passes through the vertices of a polyhedron is itself an instance of the extension problem: We have a function (the *xyz*-coordinates of a point) defined at each vertex of the mesh; we'd like a function (the *xyz*-coordinates of the smooth-surface points) that's defined on the interiors of triangles. Such a function is what a solution to the smooth interpolation problem would give us. Thus, in suggesting that we use a smooth approximating shape, we haven't really simplified the problem at all.

A partial solution to this is provided by creating a sequence of meshes through a process called **subdivision** of the original surface. These subdivided meshes converge, in the limit, to a fairly smooth surface. We'll discuss this further in Chapter 22.

9.4.1 Nonconvex Spaces

The piecewise linear extension technique works when the values at the vertices are real numbers; it's easy to extend this to tuples of real numbers (just do the extension on one coordinate at a time). It's also easy to apply it to other spaces in which convex combinations, that is,

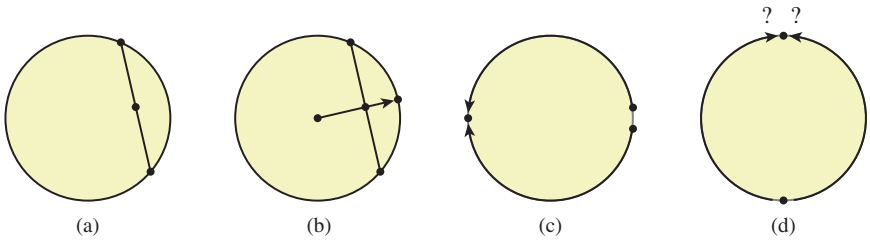


Figure 9.10: (a) If we take convex combinations of points on a circle in $\mathbf{R^2}$*, the result is a point in* $\mathbf{R^2}$*, but it is not generally on the circle; (b) if we radially project back to the circle, it works better … but the value is undefined when the original convex combination falls at the origin. (c) If we do angular interpolation, the point halfway between* 355° *and* 5° *turns out to be* 180°*. (d) If we try angular interpolation along the shortest route, our blend becomes undefined when the points are opposites.*

$$c_i f_i + c_j f_j + c_k f_k, \text{ where } c_1 + c_2 + c_3 = 1 \text{ and } c_1, c_2, c_3 \geq 0, \tag{9.14}$$

make sense. For example, if you have a 2×2 symmetric matrix associated to each vertex, you can perform barycentric blending of the matrices, because a convex combination of symmetric matrices is still a symmetric matrix.

Unfortunately, there are many interesting spaces in which convex combinations either don't make sense or fail to be defined for certain cases. The exemplar is the circle $\mathbf{S^1}$. If you treat the circle as a subset of $\mathbf{R^2}$, then (see Figure 9.10) forming convex combinations of two points makes sense … but the result is a point in the unit disk $\mathbf{D^2} \subset \mathbf{R^2}$, and generally *not* a point on $\mathbf{S^1}$.

The usual attempt at solving this is to "re-project" back to the circle, replacing the convex combination point C with $C/\|C\|$. Unfortunately, this fails when C turns out to be the origin. The problem is not one that can be solved by any clever programming trick.


It's a fairly deep theorem of topology that if

$$h : \mathbf{D^2} \to \mathbf{S^1} \tag{9.15}$$

has the property that $h(p) = p$ for all points of $\mathbf{S^1}$, then h *must* be discontinuous somewhere.

Another possible approach is to treat the values as angles, and just interpolate them. Doing this directly leads to some oddities, though: Blending some points in $\mathbf{S^1}$ that are very close (like 350° and 10°) yields a point (180° in this case) that's far from both. Doing so by "interpolating along the shorter arc" addresses *that* problem, but it introduces a new one: There are two shortest arcs between opposite points on the circle, so the answer is not well defined.


Once again, a theorem from topology explains the problem. If we simply consider the problem of finding a *halfway* point between two others, then we're seeking a function

$$H : \mathbf{S^1} \times \mathbf{S^1} \to \mathbf{S^1} \tag{9.16}$$

with certain properties. For instance, we want H to be continuous, and we want $H(p,p) = p$ for every point $p \in S^1$, and we want $H(p,q) = H(q,p)$, because "the halfway point between p and q" should be the same as "the halfway point between q and p." It turns out that even these two simple conditions are too much: No such function exists.

The situation is even worse, however: Interpolation between pairs of points, if it were possible, would let us extend our function's domain from vertices to edges of the mesh. Suppose that somehow we were given such an extension; could we *then* extend continuously over triangles? Alas, no. The study of when such extensions exist is a part of homotopy theory, and particularly of obstruction theory [MS74]. We mention this not because we expect you to learn obstruction theory, but because we hope that the existence of theorems like the ones above will dissuade you from trying to find ad hoc methods for extending functions whose codomains don't have simple enough topology.

9.4.2 Which Interpolation Method Should I Really Use?

The problem of interpolating over the interior of a triangle applies in many situations. If we are interpolating a color value, then linear interpolation in some space where equal color differences correspond to equal representation distances (see Chapter 28) makes sense. If we are interpolating a unit normal vector value, then linear interpolation is surely wrong, because the interpolated normal will generally not end up a unit vector, and if you use it in a computation that relies on unit normals, you'll get the wrong answer. And if the value is something discrete, like an object identifier, then interpolation makes no sense at all.

All of this seems to lead to the answer, "It depends!" That's true, but there's something deeper here:

> ✓ **THE MEANING PRINCIPLE:** For every number that appears in a graphics program, you need to know the semantics, the **meaning,** of that number.

Sometimes this meaning is given by units ("That number represents speed in meters per second"); sometimes it helps you place a bound on possible values for a variable ("This is a solid angle,[1] so it should be between 0 and $4\pi \approx 12.5$" or "This is a unit normal vector, so its length should be 1.0"); and sometimes the meaning is discrete ("This represents the number of paths that have ended at this pixel"). It's important to distinguish the *meaning* of the number from its representation. For instance, we often are interested in the **coverage** of a pixel (how much of a small square is covered by some shape) as a number α between zero and one, but α is sometimes represented as an 8-bit unsigned integer, that is, an integer between 0 and 255. Despite the discrete nature of the representation, it makes perfectly good sense to average two coverage values (although representing the average in the 8-bit form may introduce a roundoff error, of course).

9.5 Functions Multiply Defined at Vertices

Until now, we've discussed the very common case of functions that have a single value at each vertex and need to be interpolated across faces. But another situation arises frequently in graphics: a function where there's a value at each vertex *for each triangle that meets that vertex*. Consider, for instance, the colored octahedron in Figure 9.11. Each triangle has a color gradient across it, but no two

1. Solid angles are discussed in Chapter 26.

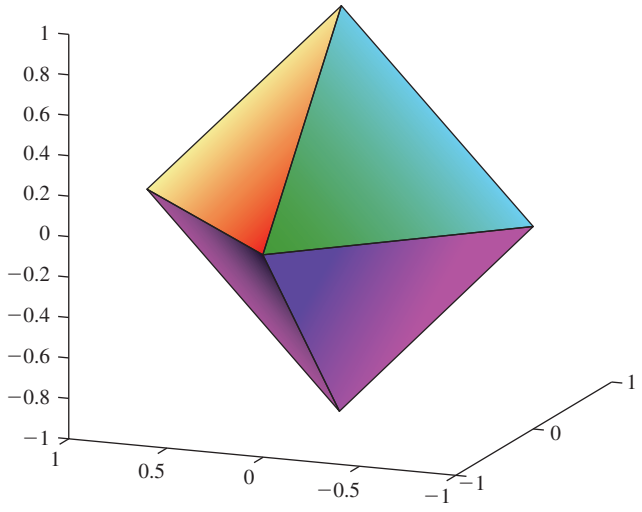


Figure 9.11: An octahedron on which each face has a different color gradient. At each vertex of the octahedron, we need to store four different colors, one for each of the faces.

triangles have the same color at each vertex. The way to generate a color function for this shape is to consider a function defined on a larger domain than the vertices. Consider all vertex-triangle pairs in which the triangle contains the vertex, that is,

$$Q = \{(v,t) : v \in t\} \subset V \times T, \tag{9.17}$$

where V and T are the sets of vertices and triangles of the mesh, respectively. Until now, we've taken a function defined h in V and extended it to all points of the mesh. We now instead define a function on Q and extend this to all points of the mesh. A point in a triangle t with vertices i, j, and k gets its value by a barycentric interpolation of the values $h(i,t)$, $h(j,t)$, and $h(k,t)$.

This leads to one important problem: A point on the edge (i,j) is a point of two different triangles. What color should it have? The answer is "It depends." From a strictly mathematical standpoint, there's no single correct answer; there will be a color associated to the interpolation of colors on one face, and a different one associated to the interpolation of colors on the other face. Neither is implicitly "right." The best we can do is to say that interpolation defines a function on

$$U = \{(P,t) : P \in t\} \subset M \times T, \tag{9.18}$$

where P is a point of the mesh M; that is, for each point P *and* each triangle t that contains it, we get a value $h(P,t)$. Since most points are in only one triangle, the second argument is generally redundant. But for those in more than one triangle, the function value is defined only with reference to the triangle under consideration.

9.6 Application: Texture Mapping

We mentioned in Chapter 1 that models are often described not only by geometry, but by **textures** as well: To each point of an object, we can associate some property (surface color is a common one), and this property is then used in rendering the

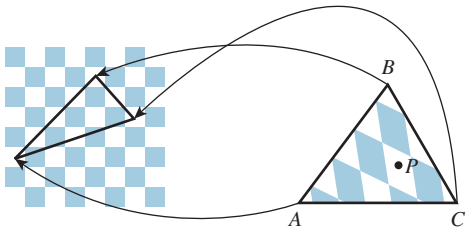


Figure 9.12: The point P of the triangle $T = \triangle ABC$ has its color determined by a texture map. The points A, B, and C have been assigned to points in the checkerboard image, as shown by the arrows; the point P corresponds to a point in a white square, so its texture color is white.

object. At a high level, when we are determining the color of a pixel in a rendering we typically base that computation on information about the underlying object that appears at that pixel; if it's a triangle of some mesh, we use, for instance, the orientation of that triangle in determining how brightly lit the point is by whatever illumination is in the scene. In some cases, the triangle may have been assigned a single color (i.e., its appearance under illumination by white light), which we also use, but in others there may be a color associated to each vertex, as in the previous section, and we can interpolate to get a color at the point of interest. Very often, however, the vertices of the triangle are associated to locations in a **texture map,** which is typically some $n \times k$ image; it's easy to think of the triangle as having been stretched and deformed to sit in the image. The color of the point of interest is then determined by looking at its location in the texture map, as in Figure 9.12, and finding the color there.

9.6.1 Assignment of Texture Coordinates

When we say that sometimes the vertices of the triangle are associated to locations in a texture map, it's natural to ask, "How did they get associated?" The answer is "by the person who created the model." There are some simple models for which the association is particularly easy. If we start with an $n \times k$ grid of triangles as shown in Figure 9.13, top, with $n = 6$ and $k = 8$, we can associate to each vertex (i, j) of this mesh a point in 3-space by setting

$$\theta = 2\pi j/(k-1) \tag{9.19}$$

$$\phi = \phi = -\frac{\pi}{2} + \pi i/(n-1) \tag{9.20}$$

$$X = \cos(\theta)\cos(\phi) \tag{9.21}$$

$$Y = \sin(\phi) \tag{9.22}$$

$$Z = \sin(\theta)\cos(\phi), \tag{9.23}$$

that is, letting θ and ϕ denote longitude and latitude, respectively. The approximately spherical shape that results is shown in the middle. We also have a 100×200 texture image of an "unprojected" map of the Earth (the vertical coordinate is proportional to latitude; the horizontal proportional to longitude). We assign texture coordinates $100i/(n-1), 200j/(k-1)$ to the vertex at position (i, j), and the resultant globe is shown rendered with the texture map at the bottom.

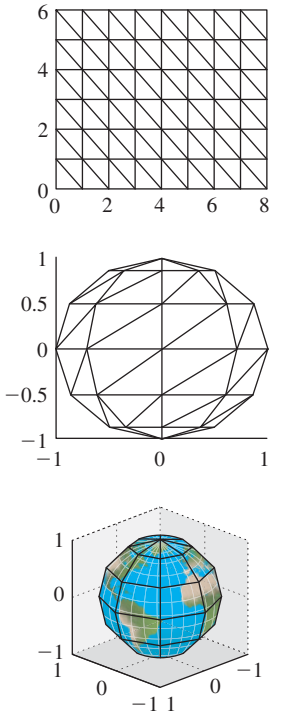


Figure 9.13: Texture-mapping a globe.

In this case, the way we created the mesh made it natural to create the texture coordinates at the same time. There is one troubling aspect of this approach, though: The texture coordinates depend on the number of pixels in the image that we use as our world map. If we created this shape and decided that the result looked bad, we might want to use a higher-resolution image, but that would entail changing the texture coordinates as well. Because of this, texture coordinates are usually specified as numbers between zero and one, representing a fraction of the way up[2] or across the image, so texture coordinates $(0.75, 0.5)$ correspond to a point three-quarters of the way up the texture image (regardless of size) and midway across, left to right.

It's commonplace to name these texture coordinates u and v so that a typical vertex now has five attributes: x, y, z, u, and v. Sometimes texture coordinates are referred to as ***uv*-coordinates.**

9.6.2 Details of Texture Mapping

If we have a mesh with texture coordinates assigned at each vertex, how do we determine the texture coordinates at some location within a triangle? *We use the techniques of this chapter, one coordinate at a time.* For instance, we have a u-coordinate at every vertex; such an assignment of a real value at every vertex uniquely defines a piecewise linear function at every point of the mesh: If P is a point of the triangle ABC, and the u-coordinates of A, B, and C are u_A, u_B, and u_C, we can determine a u-coordinate for P using barycentric coordinates. We write P in the form

$$P = \alpha A + \beta B + \gamma B \tag{9.24}$$

and then define

$$u(P) = \alpha u_A + \beta u_B + \gamma u_C. \tag{9.25}$$

We can do the same thing for v, and this uniquely determines the uv-coordinates for P.

If the triangle ABC happens to cover many pixels, we'll do this computation—find the barycentric coordinates and use them to combine the texture coordinates from the vertices—many times. Fortunately, the regularity of pixel spacing makes this repeated computation particularly easy to do in hardware, as described in Chapter 38.

9.6.3 Texture-Mapping Problems

If a triangle has texture coordinates that make it cover a large area of the texture image, but the triangle itself, when rendered, occupies a relatively small portion of the final image, then each pixel in the final image (thought of as a little square) corresponds to *many* pixels in the texture image. Our approach finds a *single* point in the texture image for each final image pixel, but perhaps it seems that the right thing to do would be to blend together many texture image pixels to get a combined result. If we do *not* do this, we get an effect called **texture aliasing,** which we'll discuss further in Chapters 17, 20 and 38. If we *do* blend together texture

2. Or down, if the image pixels are indexed from top to bottom, as often happens.

image pixels for each pixel to be rendered, the texturing process becomes very slow. One way to address this is through precomputation; we'll discuss a particular form of this precomputation, called **MIP mapping,** in Chapter 20.

9.7 Discussion

The main idea of this chapter is so simple that many graphics researchers tend to not even notice it: A real-valued function on the vertices of a mesh can be extended piecewise linearly to a real-valued function on the whole mesh, via the barycentric coordinates on each triangle. In a triangle with vertices v_i, v_j, and v_k, where the values are f_i,f_j, and f_k, the value at the point whose barycentric coordinates are c_i, c_j, and c_k is $c_i f_i + c_j f_j + c_k f_k$. This extension from vertices to interior is so ingrained that it's done without mention in countless graphics papers. There *are* other possible extensions from values at vertices to values on whole triangles, some of which are discussed in Chapter 22, but this piecewise linear interpolation dominates.

This same piecewise linear interpolation method works for functions with other codomains (e.g., $\mathbf{R^2}$ or $\mathbf{R^3}$), as long as those codomains support the idea of a "convex combination." For domains that do not (like the circle, or the sphere, or the set of 3×3 rotation matrices) there may be no reasonable way to extend the function over triangles.

Solving problems like this interpolation from vertices in the abstract helps avoid implementation-dependent artifacts. If we had studied the problem in the context of interpolating values represented by 8-bit integers across the interior of a triangle in a GPU, we might have gotten distracted by the low bit-count representation, rather than trying to understand the general problem and then adapt the solution to our particular constraints. This is another instance of the Approximate the Solution principle.

One important application of piecewise linear interpolation is texture mapping, in which a property of a surface is associated to each vertex, and these property values are linearly interpolated over the interiors of triangles. If the property is "a location in a texture image," then the interpolated values can be used to add finely detailed variations in color to the object. Chapter 20 discusses this.

9.8 Exercises

Exercise 9.1: The basis functions we described in this chapter, shown in Figure 9.4(a), not only *correspond* to a basis for $\mathbf{R}^n$, they also *form* a basis of another vector space—a subspace of the vector space of all continuous functions on the domain $[1, n]$. To justify this claim, show that these functions are in fact linearly independent *as functions*.

Exercise 9.2: (a) Draw a tetrahedron; pick a vertex and draw its link and its star. Suppose v and w are distinct vertices of the tetrahedron. What is the intersection of the star of v and the star of w?
(b) Draw an octahedron, and answer the same question when v is the top vertex and w is the bottom one.

Exercise 9.3: Think about a mesh that contains vertices, edges, triangles, and tetrahedra. You could call this a *solid* mesh rather than a surface mesh. Consider a

nonboundary vertex of such a mesh. What is the topology of the star of the vertex? What is the topology of the link of the vertex?

Exercise 9.4: Show that the interpolation of values across a triangle using the scanline method is the same as the one defined using the barycentric method. Hint: Show that if the triangle is in the xy-plane, then each of the methods defines a function of the form $f(x, y) = Ax + By + C$. Now suppose you have two such functions with the same values at the three vertices of a triangle. Explain why they must take the same values at all interior points as well.

Exercise 9.5: The function H of Equation 9.16 cannot exist; here's a reason why. Suppose that it did. Define a new function

$$K : [0, 2\pi] \times [0, 2\pi] \to [0, 2\pi] : (\theta, \phi) \mapsto H(\theta, \phi), \tag{9.26}$$

where we're implicitly using the correspondence of the number θ in the interval $[0, 2\pi]$ with the point $(\cos\theta, \sin\theta)$ of $\mathbf{S^1}$. Now consider the loop in the domain of K consisting of three straight lines, from $(0, 0)$ to $(2\pi, 0)$, from $(2\pi, 0)$ to $(2\pi, 2\pi)$, and from $(2\pi, 2\pi)$ back to $(0, 0)$.

(a) Draw this path.

(b) For each point p of this path, $K(p)$ is an element of $\mathbf{S^1}$. Restricting K to this path gives a map from the path to $\mathbf{S^1} \subset \mathbf{R^2}$. We can compute the winding number of this path about the origin in $\mathbf{R^2}$. Explain, using the assumed properties of H, why the winding numbers of the first two parts of the path must be equal.

(c) Explain why the winding number of the last part must be one.

(d) Conclude that the total winding number must be odd.

(e) Now consider shrinking the triangular loop toward the center of the triangle. The winding number will be a continuous integer-valued function of the triangle size. Explain why this means the winding number must be constant.

(f) When the triangle has shrunk to a single point, explain why the winding number must be zero.

(g) Explain why this is a contradiction.

Exercise 9.6: Use the 2D test bed to create a program to experiment with texture mapping. Display, on the left, a 100×100 checkerboard image, with 10×10 squares. Atop this, draw a triangle whose vertices are draggable. On the right, draw a fixed equilateral triangle above a 100×100 grid of tiny squares (representing display pixels). For each of these display pixels, compute and store the barycentric coordinates of its center (with respect to the equilateral triangle). Using the locations of the three draggable vertices in the checkerboard as texture coordinates, compute, for each display pixel within the equilateral triangle, the uv-coordinates of the pixel center, and then use these uv-coordinates to determine the pixel color from the checkerboard texture image (see Figure 9.14). Experiment with mapping the equilateral triangle to a tiny triangle in the texture image, and to a large one; experiment with mapping it to a tall and narrow triangle. What problems do you observe?

Exercise 9.7: Suppose that you have a polyline in the plane with vertices $P_0, P_1, \ldots, P_n$ and you want to "resample" it, placing multiple points on each edge, equally spaced, for a total of $k + 1$ points $Q_0 = P_0, \ldots, Q_k = P_n$.

(a) Write a program to do this: First compute the total length L of the polyline, then place the Qs along the polyline so that they're separated by L/k. This will require special handling at each vertex of the original polyline.

(b) When you're done, you'll notice that if n and k are nearly equal, many "corners" tend to get cut off the original polyline. It's natural to say, "I want equally

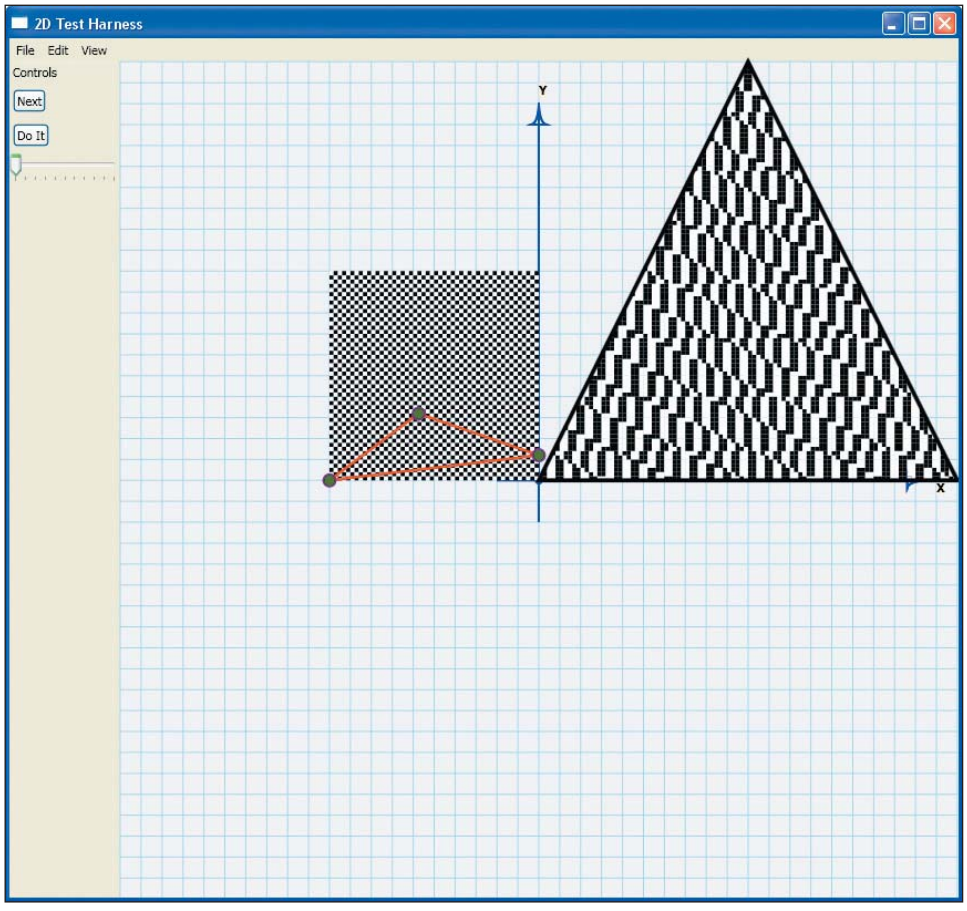


Figure 9.14: A screen capture for the texture-mapping program of Exercise 9.6, showing the texture on the left, a large triangle, and the locations of the triangle's vertices in texture coordinates atop the texture. The resultant texturing on the triangle's interior is shown on the right.

spaced points, but I want them to include all the original points!" That's generally not possible, but we can come close. Suppose that the shortest edge of the original polygon has length s. Show that you can place approximately L/s points $Q_0, Q_1, \ldots$ on the original polyline, including all the points $P_0, \ldots, P_n$, with the property that the ratio of the greatest gap between adjacent points and the smallest gap is no more than 2.
(c) Suppose you let yourself place CL/s points with the same constraints as in the previous part, for some C greater than one. Estimate the max-min gap ratio in terms of C.

Exercise 9.8: Consider the interval $[p, q]$ where $p \neq q$. If we define $\alpha(x) = \frac{x-p}{q-p}$ and $\beta(x) = \frac{x-q}{p-q}$, then α and β are called the **barycentric coordinates of** x.
(a) Show that for $x \in [p, q]$, both $\alpha(x)$ and $\beta(x)$ are between 0 and 1.
(b) Show that $\alpha(x) + \beta(x) = 1$.
◈ (c) Clearly α and β can be defined on the rest of the real line, and these definitions depend on p and q; if we call them α_{pq} and β_{pq}, then we can, for another

interval $[p', q']$, define corresponding barycentric coordinates. How are $\alpha_{pq}(x)$ and $\alpha_{p',q'}(x)$ related?

Exercise 9.9: Suppose you have a nondegenerate triangle in 3-space with vertices P_0, P_1, and P_2 so that $\mathbf{v}_1 = P_1 - P_0$ and $\mathbf{v}_2 = P_2 - P_0$ are nonzero and nonparallel. Further, suppose that we have values $f_0, f_1, f_2 \in \mathbf{R}$ associated with the three vertices. Barycentric interpolation of these values over the triangle defines a function that can be written in the form

$$f(P) = f_0 + (P - P_0) \cdot \mathbf{w} \tag{9.27}$$

for some vector $\mathbf{w}$. We'll see this in two steps: First, we'll compute a possible value for $\mathbf{w}$, and then we'll show that if it has this value, f actually matches the given values at the vertices.

(a) Show that the vector $\mathbf{w}$ must satisfy $\mathbf{v}_i \cdot \mathbf{w} = f_i - f_0$ for $i = 1, 2$ for the function defined by Equation 9.27 to satisfy $f(P_1) = f_1$ and $f(P_2) = f_2$.

(b) Let $\mathbf{S}$ be the matrix whose columns are the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$. Show that the conditions of part (a) can be rewritten in the form

$$\mathbf{S}^\mathbf{T}\mathbf{w} = \begin{bmatrix} f_1 - f_0 \\ f_2 - f_0 \end{bmatrix}, \tag{9.28}$$

and that therefore $\mathbf{w}$ must also satisfy

$$\mathbf{S}\mathbf{S}^\mathbf{T}\mathbf{w} = \mathbf{S} \begin{bmatrix} f_1 - f_0 \\ f_2 - f_0 \end{bmatrix}. \tag{9.29}$$

(c) Explain why $\mathbf{S}\mathbf{S}^\mathbf{T}$ must be invertible.

(d) Conclude that $\mathbf{w} = (\mathbf{S}\mathbf{S}^\mathbf{T})^{-1}\mathbf{S} \begin{bmatrix} f_1 - f_0 \\ f_2 - f_0 \end{bmatrix}$.

(e) Verify that if we use this formula for $\mathbf{w}$, then $f(P_i) = f_i$ for $i = 0, 1, 2$.

(f) Suppose that $\mathbf{w}' = \mathbf{w} + \alpha\mathbf{n}$, where $\mathbf{n} = \mathbf{v}_1 \times \mathbf{v}_2$ is the normal vector to the triangle. Show that we can replace $\mathbf{w}$ with $\mathbf{w}'$ in the formula for f and still have $f(P_i) = f_i$ for $i = 0, 1, 2$.

Chapter 10

Transformations in Two Dimensions

10.1 Introduction

As you saw in Chapters 2 and 6, when we think about taking an object for which we have a geometric model and putting it in a scene, we typically need to do three things: *Move* the object to some location, *scale* it up or down so that it fits well with the other objects in the scene, and *rotate* it until it has the right orientation. These operations—translation, scaling, and rotation—are part of every graphics system. Both scaling and rotation are **linear transformations** on the coordinates of the object's points. Recall that a linear transformation,

$$T : \mathbf{R^2} \to \mathbf{R^2}, \tag{10.1}$$

is one for which $T(\mathbf{v} + \alpha\mathbf{w}) = T(\mathbf{v}) + \alpha T(\mathbf{w})$ for any two vectors $\mathbf{v}$ and $\mathbf{w}$ in $\mathbf{R^2}$, and any real number α. Intuitively, it's a transformation that preserves lines and leaves the origin unmoved.

Inline Exercise 10.1: Suppose T is linear. Insert $\alpha = 1$ in the definition of linearity. What does it say? Insert $\mathbf{v} = \mathbf{0}$ in the definition. What does it say?

Inline Exercise 10.2: When we say that a linear transformation "preserves lines," we mean that if ℓ is a line, then the set of points $T(\ell)$ must also *lie in* some line. You might expect that we'd require that $T(\ell)$ actually *be* a line, but that would mean that transformations like "project everything perpendicularly onto the x-axis" would not be counted as "linear." For this particular projection transformation, describe a line ℓ such that $T(\ell)$ is contained in a line, but is not itself a line.

The definition of linearity guarantees that for any linear transformation T, we have $T(\mathbf{0}) = \mathbf{0}$: If we choose $\mathbf{v} = \mathbf{w} = \mathbf{0}$ and $\alpha = 1$, the definition tells us that

$$T(\mathbf{0}) = T(\mathbf{0} + 1\mathbf{0}) = T(\mathbf{0}) + 1T(\mathbf{0}) = T(\mathbf{0}) + T(\mathbf{0}). \tag{10.2}$$

Subtracting $T(\mathbf{0})$ from the first and last parts of this chain gives us $\mathbf{0} = T(\mathbf{0})$. This means that **translation**—moving every point of the plane by the same amount—is, in general, *not* a linear transformation except in the special case of translation by zero, in which all points are left where they are. Shortly we'll describe a trick for putting the Euclidean plane into $\mathbf{R^3}$ (but *not* as the $z = 0$ plane as is usually done); once we do this, we'll see that certain linear transformations on $\mathbf{R^3}$ end up performing translations on this embedded plane.

For now, let's look at only the plane. We assume that you have *some* familiarity with linear transformations already; indeed, the serious student of computer graphics should, at some point, study linear algebra carefully. But one can learn a great deal about graphics with only a modest amount of knowledge of the subject, which we summarize here briefly.

In the first few sections, we use the convention of most linear-algebra texts: The vectors are arrows at the origin, and we think of the vector $\begin{bmatrix} u \\ v \end{bmatrix}$ as being identified with the point (u, v). Later we'll return to the point-vector distinction.

For any 2×2 matrix $\mathbf{M}$, the function $\mathbf{v} \mapsto \mathbf{Mv}$ is a linear transformation from $\mathbf{R^2}$ to $\mathbf{R^2}$. We refer to this as a **matrix transformation.** In this chapter, we look at five such transformations in detail, study matrix transformations in general, and introduce a method for incorporating translation into the matrix-transformation formulation. We then apply these ideas to transforming objects *and* changing coordinate systems, returning to the clock example of Chapter 2 to see the ideas in practice.

10.2 Five Examples

We begin with five examples of linear transformations in the plane; we'll refer to these by the names $T_1, \ldots, T_5$ throughout the chapter.

Example 1: Rotation. Let $\mathbf{M}_1 = \begin{bmatrix} \cos 30^\circ & -\sin 30^\circ \\ \sin 30^\circ & \cos 30^\circ \end{bmatrix}$ and

$$T_1 : \mathbf{R^2} \to \mathbf{R^2} : \begin{bmatrix} x \\ y \end{bmatrix} \mapsto \mathbf{M}_1 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos 30^\circ & -\sin 30^\circ \\ \sin 30^\circ & \cos 30^\circ \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}. \tag{10.3}$$

Recall that $\mathbf{e}_1$ denotes the vector $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\mathbf{e}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$; this transformation sends $\mathbf{e}_1$ to the vector $\begin{bmatrix} \cos 30^\circ \\ \sin 30^\circ \end{bmatrix}$ and $\mathbf{e}_2$ to $\begin{bmatrix} -\sin 30^\circ \\ \cos 30^\circ \end{bmatrix}$, which are vectors that are 30° counterclockwise from the x- and y-axes, respectively (see Figure 10.1).

There's nothing special about the number 30 in this example; by replacing 30° with any angle, you can build a transformation that rotates things counterclockwise by that angle.

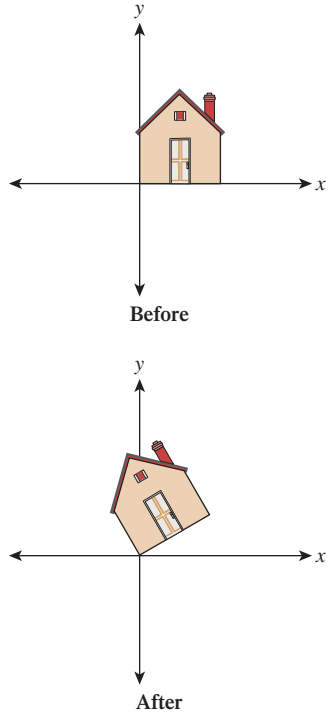


Figure 10.1: Rotation by 30°.

Inline Exercise 10.3: Write down the matrix transformation that rotates everything in the plane by 180° counterclockwise. Actually compute the sines and cosines so that you end up with a matrix filled with numbers in your answer. Apply this transformation to the corners of the unit square, $(0,0)$, $(1,0)$, $(0,1)$, and $(1,1)$.

Example 2: Nonuniform scaling. Let $\mathbf{M}_2 = \begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix}$ and

$$T_2 : \mathbf{R^2} \to \mathbf{R^2} : \begin{bmatrix} x \\ y \end{bmatrix} \mapsto \mathbf{M}_2 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3x \\ 2y \end{bmatrix}. \tag{10.4}$$

This transformation stretches everything by a factor of three in the x-direction and a factor of two in the y-direction, as shown in Figure 10.2. If both stretch factors were three, we'd say that the transformation "scaled things up by three" and is a **uniform scaling transformation.** T_2 represents a generalization of this idea: Rather than scaling uniformly in each direction, it's called a **nonuniform scaling transformation** or, less formally, a **nonuniform scale.**

Once again the example generalizes: By placing numbers other than 2 and 3 along the diagonal of the matrix, we can scale each axis by any amount we please. These scaling amounts can include zero and negative numbers.

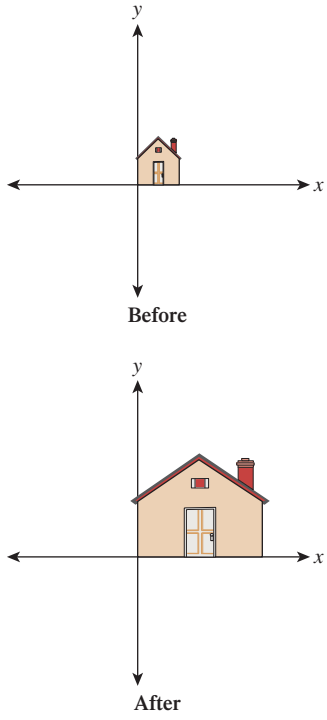


Figure 10.2: T_2 stretches the x-axis by three and the y-axis by two.

Inline Exercise 10.4: Write down the matrix for a uniform scale by -1. How does your answer relate to your answer to inline Exercise 10.3? Can you explain?

Inline Exercise 10.5: Write down a transformation matrix that scales in x by zero and in y by 1. Informally describe what the associated transformation does to the house.

Example 3: Shearing. Let $\mathbf{M}_3 = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$ and

$$T_3 : \mathbf{R^2} \to \mathbf{R^2} : \begin{bmatrix} x \\ y \end{bmatrix} \mapsto \mathbf{M}_3 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x+2y \\ y \end{bmatrix}. \tag{10.5}$$

As Figure 10.3 shows, T_3 preserves height along the y-axis but moves points parallel to the x-axis, with the amount of movement determined by the y-value. The x-axis itself remains fixed. Such a transformation is called a **shearing transformation.**

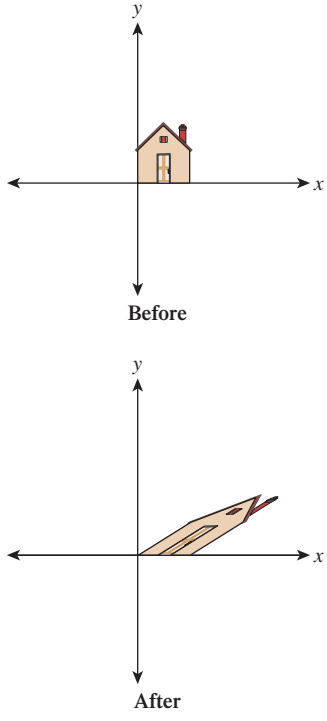


Figure 10.3: A shearing transformation, T_3.

Inline Exercise 10.6: Generalize to build a transformation that keeps the y-axis fixed but shears vertically instead of horizontally.

Example 4: A general transformation. Let $\mathbf{M}_4 = \begin{bmatrix} 1 & -1 \\ 2 & 2 \end{bmatrix}$ and

$$T_4 : \mathbf{R^2} \to \mathbf{R^2} : \begin{bmatrix} x \\ y \end{bmatrix} \mapsto \mathbf{M}_4 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 2 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}. \tag{10.6}$$

Figure 10.4 shows the effects of T_4. It distorts the house figure, but not by just a rotation or scaling or shearing along the coordinate axes.

Example 5: A degenerate (or singular) transformation Let

$$T_5 : \mathbf{R^2} \to \mathbf{R^2} : \begin{bmatrix} x \\ y \end{bmatrix} \mapsto \begin{bmatrix} 1 & -1 \\ 2 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x - y \\ 2x - 2y \end{bmatrix}. \tag{10.7}$$

Figure 10.5 shows why we call this transformation **degenerate:** Unlike the others, it collapses the whole two-dimensional plane down to a one-dimensional subspace, a line. There's no longer a nice correspondence between points in the domain and points in the codomain: Certain points in the codomain no longer correspond to *any* point in the domain; others correspond to *many* points in the domain. Such a transformation is also called **singular,** as is the matrix defining it. Those familiar with linear algebra will note that this is equivalent to saying that the determinant of $\mathbf{M}_5 = \begin{bmatrix} 1 & -1 \\ 2 & -2 \end{bmatrix}$ is zero, or saying that its columns are linearly dependent.

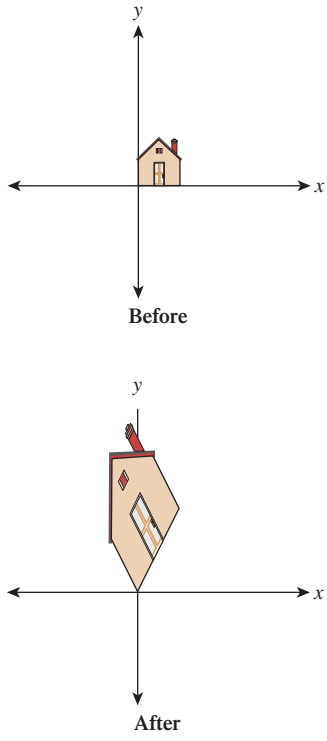


Figure 10.4: A general transformation. The house has been quite distorted, in a way that's hard to describe simply, as we've done for the earlier examples.

10.3 Important Facts about Transformations

Here we'll describe several properties of linear transformations from $\mathbf{R^2}$ to $\mathbf{R^2}$. These properties are important in part because they all generalize: They apply (in some form) to transformations from $\mathbf{R}^n$ to $\mathbf{R}^k$ for any n and k. We'll mostly be concerned with values of n and k between 1 and 4; in this section, we'll concentrate on $n = k = 2$.

10.3.1 Multiplication by a Matrix Is a Linear Transformation

If $\mathbf{M}$ is a 2×2 matrix, then the function $T_\mathbf{M}$ defined by

$$T_\mathbf{M} : \mathbf{R^2} \to \mathbf{R^2} : \mathbf{x} \mapsto \mathbf{Mx} \tag{10.8}$$

is linear. All five examples above demonstrate this.

For nondegenerate transformations, lines are sent to lines, as T_1 through T_4 show. For degenerate ones, a line may be sent to a single point. For instance, T_5 sends the line consisting of all vectors of the form $\begin{bmatrix} b \\ b \end{bmatrix}$ to the zero vector.

Because multiplication by a matrix $\mathbf{M}$ is always a linear transformation, we'll call $T_\mathbf{M}$ the **transformation associated to the matrix M.**

10.3.2 Multiplication by a Matrix Is the *Only* Linear Transformation

In $\mathbf{R}^n$, it turns out that for *every* linear transform T, there's a matrix $\mathbf{M}$ with $T(\mathbf{x}) = \mathbf{Mx}$, which means that every linear transformation is a matrix transformation. We'll see in Section 10.3.5 how to find $\mathbf{M}$, given T, even if T is expressed in some other way. This will show that the matrix $\mathbf{M}$ is completely determined by the transformation T, and we can thus call it the **matrix associated to the transformation.**

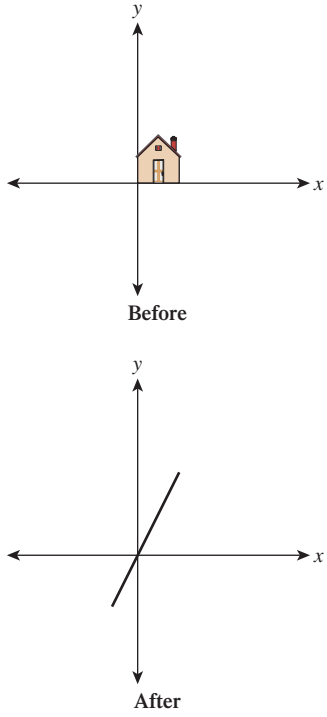


Figure 10.5: A degenerate transformation, T_5.

As a special example, the matrix $\mathbf{I}$, with ones on the diagonal and zeroes off the diagonal, is called the **identity matrix;** the associated transformation

$$T(\mathbf{x}) = \mathbf{I}\mathbf{x} \tag{10.9}$$

is special: It's the identity transformation that leaves every vector $\mathbf{x}$ unchanged.

Inline Exercise 10.7: There is an identity matrix of every size: a 1×1 identity, a 2×2 identity, etc. Write out the first three.

10.3.3 Function Composition and Matrix Multiplication Are Related

If $\mathbf{M}$ and $\mathbf{K}$ are 2×2 matrices, then they define transformations $T_{\mathbf{M}}$ and $T_{\mathbf{K}}$. When we compose these, we get the transformation

$$\begin{aligned} T_{\mathbf{M}} \circ T_{\mathbf{K}} : \mathbf{R^2} \to \mathbf{R^2} : \mathbf{x} \mapsto T_{\mathbf{M}}(T_{\mathbf{K}}(\mathbf{x})) &= T_{\mathbf{M}}(\mathbf{K}\mathbf{x}) && (10.10)\\ &= \mathbf{M}(\mathbf{K}\mathbf{x}) && (10.11)\\ &= (\mathbf{M}\mathbf{K})\mathbf{x} && (10.12)\\ &= T_{\mathbf{M}\mathbf{K}}(\mathbf{x}). && (10.13) \end{aligned}$$

In other words, the composed transformation is also a matrix transformation, with matrix $\mathbf{MK}$. Note that when we write $T_{\mathbf{M}}(T_{\mathbf{K}}(\mathbf{x}))$, the transformation $T_{\mathbf{K}}$ is applied *first.* So, for example, if we look at the transformation $T_2 \circ T_3$, it first shears the house and *then* scales the result nonuniformly.

Inline Exercise 10.8: Describe the appearance of the house after transforming it by $T_1 \circ T_2$ and after transforming it by $T_2 \circ T_1$.

10.3.4 Matrix Inverse and Inverse Functions Are Related

A matrix $\mathbf{M}$ is **invertible** if there's a matrix $\mathbf{B}$ with the property that $\mathbf{BM} = \mathbf{MB} = \mathbf{I}$. If such a matrix exists, it's denoted $\mathbf{M}^{-1}$.

If $\mathbf{M}$ is invertible and $S(\mathbf{x}) = \mathbf{M}^{-1}\mathbf{x}$, then S is the inverse function of $T_{\mathbf{M}}$, that is,

$$\begin{aligned} S(T_{\mathbf{M}}(\mathbf{x})) &= \mathbf{x} \quad \text{and} && (10.14)\\ T_{\mathbf{M}}(S(\mathbf{x})) &= \mathbf{x}. && (10.15) \end{aligned}$$

Inline Exercise 10.9: Using Equation 10.13, explain why Equation 10.15 holds.

If $\mathbf{M}$ is not invertible, then $T_{\mathbf{M}}$ has no inverse.

Let's look at our examples. The matrix for T_1 has an inverse: Simply replace 30 by -30 in all the entries. The resultant transformation rotates clockwise by 30°; performing one rotation and then the other effectively does nothing (i.e., it is the identity transformation). The inverse for the matrix for T_2 is diagonal, with entries

$\frac{1}{3}$ and $\frac{1}{2}$. The inverse of the matrix for T_3 is $\begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}$ (note the negative sign). The associated transformation also shears parallel to the x-axis, but vectors in the upper half-plane are moved to the *left,* which undoes the moving to the right done by T_3.

For these first three it was fairly easy to guess the inverse matrices, because we could understand how to invert the transformation. The inverse of the matrix for T_4 is

$$\frac{1}{4}\begin{bmatrix} 2 & 1 \\ -2 & 1 \end{bmatrix}, \tag{10.16}$$

which we computed using a general rule for inverses of 2×2 matrices (the only such rule worth memorizing):

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}. \tag{10.17}$$

Finally, for T_5, the matrix has no inverse; if it did, the function T_5 would be invertible: It would be possible to identify, for each point in the codomain, a single point in the domain that's sent there. But we've already seen this isn't possible.

Inline Exercise 10.10: Apply the formula from Equation 10.17 to the matrix for T_5 to attempt to compute its inverse. What goes wrong?

10.3.5 Finding the Matrix for a Transformation

We've said that every linear transformation really is just multiplication by some matrix, but how do we *find* that matrix? Suppose, for instance, that we'd like to find a linear transformation to flip our house across the y-axis so that the house ends up on the left side of the y-axis. (Perhaps you can guess the transformation that does this, and the associated matrix, but we'll work through the problem directly.)

The key idea is this: If we know where the transformation sends $\mathbf{e}_1$ and $\mathbf{e}_2$, we know the matrix. Why? We know that the transformation must have the form

$$T\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix}; \tag{10.18}$$

we just don't know the values of a, b, c, and d. Well, $T(\mathbf{e}_1)$ is then

$$T\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} a \\ c \end{bmatrix}. \tag{10.19}$$

Similarly, $T(\mathbf{e}_2)$ is the vector $\begin{bmatrix} b \\ d \end{bmatrix}$. So knowing $T(\mathbf{e}_1)$ and $T(\mathbf{e}_2)$ tells us all the matrix entries. Applying this to the problem of flipping the house, we know that $T(\mathbf{e}_1) = -\mathbf{e}_1$, because we want a point on the positive x-axis to be sent to the corresponding point on the negative x-axis, so $a = -1$ and $c = 0$. On the other hand, $T(\mathbf{e}_2) = \mathbf{e}_2$, because every vector on the y-axis should be left untouched, so $b = 0$ and $d = 1$. Thus, the matrix for the house-flip transformation is just

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{10.20}$$

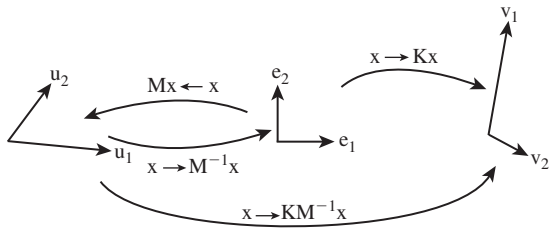


Figure 10.6: Multiplication by the matrix $\mathbf{M}$ *takes* $\mathbf{e}_1$ *and* $\mathbf{e}_2$ *to* $\mathbf{u}_1$ *and* $\mathbf{u}_2$*, respectively, so multiplying* $\mathbf{M}^{-1}$ *does the opposite. Multiplying by* $\mathbf{K}$ *takes* $\mathbf{e}_1$ *and* $\mathbf{e}_2$ *to* $\mathbf{v}_1$ *and* $\mathbf{v}_2$*, so multiplying first by* $\mathbf{M}^{-1}$ *and then by* $\mathbf{K}$*, that is, multiplying by* $\mathbf{KM}^{-1}$*, takes* $\mathbf{u}_1$ *to* $\mathbf{e}_1$ *to* $\mathbf{v}_1$*, and similarly for* $\mathbf{u}_2$*.*

Inline Exercise 10.11: (a) Find a matrix transformation sending $\mathbf{e}_1$ to $\begin{bmatrix}0\\4\end{bmatrix}$ and $\mathbf{e}_2$ to $\begin{bmatrix}1\\1\end{bmatrix}$.
(b) Use the relationship of matrix inverse to the inverse of a transform, and the formula for the inverse of a 2×2 matrix, to find a transformation sending $\begin{bmatrix}0\\4\end{bmatrix}$ to $\mathbf{e}_1$ and $\begin{bmatrix}1\\1\end{bmatrix}$ to $\mathbf{e}_2$ as well.

As Inline Exercise 10.11 shows, we now have the tools to send the **standard basis vectors** $\mathbf{e}_1$ and $\mathbf{e}_2$ to any two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, and vice versa (provided that $\mathbf{v}_1$ and $\mathbf{v}_2$ are independent, that is, neither is a multiple of the other). We can combine this with the idea that composition of linear transformations (performing one after the other) corresponds to multiplication of matrices and thus create a solution to a rather general problem.

Problem: Given independent vectors $\mathbf{u}_1$ and $\mathbf{u}_2$ and any two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, find a linear transformation, in matrix form, that sends $\mathbf{u}_1$ to $\mathbf{v}_1$ and $\mathbf{u}_2$ to $\mathbf{v}_2$.

Solution: Let $\mathbf{M}$ be the matrix whose columns are $\mathbf{u}_1$ and $\mathbf{u}_2$. Then

$$T : \mathbf{R^2} \to \mathbf{R^2} : \mathbf{x} \mapsto \mathbf{Mx} \tag{10.21}$$

sends $\mathbf{e}_1$ to $\mathbf{u}_1$ and $\mathbf{e}_2$ to $\mathbf{u}_2$ (see Figure 10.6). Therefore,

$$S : \mathbf{R^2} \to \mathbf{R^2} : \mathbf{x} \mapsto \mathbf{M}^{-1}\mathbf{x} \tag{10.22}$$

sends $\mathbf{u}_1$ to $\mathbf{e}_1$ and $\mathbf{u}_2$ to $\mathbf{e}_2$.

Now let $\mathbf{K}$ be the matrix with columns $\mathbf{v}_1$ and $\mathbf{v}_2$. The transformation

$$R : \mathbf{R^2} \to \mathbf{R^2} : \mathbf{x} \mapsto \mathbf{Kx} \tag{10.23}$$

sends $\mathbf{e}_1$ to $\mathbf{v}_1$ and $\mathbf{e}_2$ to $\mathbf{v}_2$.

If we apply first S and then R to $\mathbf{u}_1$, it will be sent to $\mathbf{e}_1$ (by S), and thence to $\mathbf{v}_1$ by R; a similar argument applies to $\mathbf{u}_2$. Writing this in equations,

$$R(S(\mathbf{x})) = R(\mathbf{M}^{-1}\mathbf{x}) \tag{10.24}$$

$$= \mathbf{K}(\mathbf{M}^{-1}\mathbf{x}) \tag{10.25}$$

$$= (\mathbf{KM}^{-1})\mathbf{x}. \tag{10.26}$$

Thus, the matrix for the transformation sending the $\mathbf{u}$'s to the $\mathbf{v}$'s is just $\mathbf{KM}^{-1}$.

Let's make this concrete with an example. We'll find a matrix sending

$$\mathbf{u}_1 = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \qquad \text{and} \qquad \mathbf{u}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \tag{10.27}$$

to

$$\mathbf{v}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad \text{and} \qquad \mathbf{v}_2 = \begin{bmatrix} 2 \\ -1 \end{bmatrix}, \tag{10.28}$$

respectively. Following the pattern above, the matrices $\mathbf{M}$ and $\mathbf{K}$ are

$$\mathbf{M} = \begin{bmatrix} 2 & 1 \\ 3 & -1 \end{bmatrix} \tag{10.29}$$

$$\mathbf{K} = \begin{bmatrix} 1 & 2 \\ 1 & -1 \end{bmatrix}. \tag{10.30}$$

Using the matrix inversion formula (Equation 10.17), we find

$$\mathbf{M}^{-1} = \frac{-1}{5} \begin{bmatrix} -1 & -1 \\ -3 & 2 \end{bmatrix} \tag{10.31}$$

so that the matrix for the overall transformation is

$$\mathbf{J} = \mathbf{KM}^{-1} = \begin{bmatrix} 1 & 2 \\ 1 & -1 \end{bmatrix} \cdot \frac{-1}{5} \begin{bmatrix} -1 & -1 \\ -3 & 2 \end{bmatrix} \tag{10.32}$$

$$= \begin{bmatrix} 7/5 & -3/5 \\ -2/5 & 3/5 \end{bmatrix}. \tag{10.33}$$

As you may have guessed, the kinds of transformations we used in WPF in Chapter 2 are internally represented as matrix transformations, and transformation groups are represented by sets of matrices that are multiplied together to generate the effect of the group.

Inline Exercise 10.12: Verify that the transformation associated to the matrix $\mathbf{J}$ in Equation 10.32 really does send $\mathbf{u}_1$ to $\mathbf{v}_1$ and $\mathbf{u}_2$ to $\mathbf{v}_2$.

Inline Exercise 10.13: Let $\mathbf{u}_1 = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$ and $\mathbf{u}_2 = \begin{bmatrix} 1 \\ 4 \end{bmatrix}$; pick any two nonzero vectors you like as $\mathbf{v}_1$ and $\mathbf{v}_2$, and find the matrix transformation that sends each $\mathbf{u}_i$ to the corresponding $\mathbf{v}_i$.

The recipe above for building matrix transformations shows the following: Every linear transformation from $\mathbf{R^2}$ to $\mathbf{R^2}$ is determined by its values on two independent vectors. In fact, this is a far more general property: Any linear transformation from $\mathbf{R^2}$ to $\mathbf{R}^k$ is determined by its values on two independent vectors, and indeed, any linear transformation from $\mathbf{R}^n$ to $\mathbf{R}^k$ is determined by its values on n independent vectors (where to make sense of these, we need to extend our definition of "independence" to more than two vectors, which we'll do presently).

10.3.6 Transformations and Coordinate Systems

We tend to think about linear transformations as moving points around, but leaving the origin fixed; we'll often use them that way. Equally important, however, is their use in changing coordinate systems. If we have two coordinate systems on $\mathbf{R}^2$ with the same origin, as in Figure 10.7, then every arrow has coordinates in both the red and the blue systems. The two red coordinates can be written as a vector, as can the two blue coordinates. The vector $\mathbf{u}$, for instance, has coordinates $\begin{bmatrix}3\\2\end{bmatrix}$ in the red system and approximately $\begin{bmatrix}-0.2\\3.6\end{bmatrix}$ in the blue system.

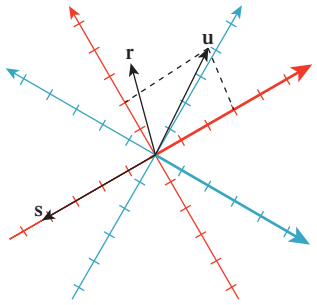


Figure 10.7: Two different coordinate systems for $\mathbf{R}^2$; the vector $\mathbf{u}$, expressed in the red coordinate system, has coordinates 3 and 2, indicated by the dotted lines, while the coordinates in the blue coordinate system are approximately -0.2 and 3.6, where we've drawn, in each case, the positive side of the first coordinate axis in bold.

Inline Exercise 10.14: Use a ruler to find the coordinates of $\mathbf{r}$ and $\mathbf{s}$ in each of the two coordinate systems.

We could tabulate every imaginable arrow's coordinates in the red and blue systems to convert from red to blue coordinates. But there is a far simpler way to achieve the same result. The conversion from red coordinates to blue coordinates is *linear* and can be expressed by a matrix transformation. In this example, the matrix is

$$\mathbf{M} = \frac{1}{2}\begin{bmatrix}1 & -\sqrt{3}\\ \sqrt{3} & 1\end{bmatrix}. \tag{10.34}$$

Multiplying $\mathbf{M}$ by the coordinates of $\mathbf{u}$ in the red system gets us

$$\begin{aligned}\mathbf{v} &= \mathbf{M}\mathbf{u} && (10.35)\\ &= \frac{1}{2}\begin{bmatrix}1 & -\sqrt{3}\\ \sqrt{3} & 1\end{bmatrix}\begin{bmatrix}3\\2\end{bmatrix} && (10.36)\\ &= \frac{1}{2}\begin{bmatrix}3-2\sqrt{3}\\ 3\sqrt{3}+2\end{bmatrix} && (10.37)\\ &\approx \begin{bmatrix}-0.2\\3.6\end{bmatrix}, && (10.38)\end{aligned}$$

which is the coordinate vector for $\mathbf{u}$ in the blue system.

Inline Exercise 10.15: Confirm, for each of the other arrows in Figure 10.7, that the same transformation converts red to blue coordinates.

By the way, when creating this example we computed $\mathbf{M}$ just as we did at the start of the preceding section: We found the blue coordinates of each of the two basis vectors for the red coordinate system, and used these as the columns of $\mathbf{M}$.

In the special case where we want to go from the usual coordinates on a vector to its coordinates in some coordinate system with basis vectors $\mathbf{u}_1, \mathbf{u}_2$, which are *unit vectors* and *mutually perpendicular,* the transformation matrix is one whose *rows* are the transposes of $\mathbf{u}_1$ and $\mathbf{u}_2$.

For example, if $\mathbf{u}_1 = \begin{bmatrix}3/5\\4/5\end{bmatrix}$ and $\mathbf{u}_2 = \begin{bmatrix}-4/5\\3/5\end{bmatrix}$ (check for yourself that these are unit length and perpendicular), then the vector $\mathbf{v} = \begin{bmatrix}4\\2\end{bmatrix}$, expressed in $\mathbf{u}$-coordinates, is

$$\begin{bmatrix} 3/5 & 4/5 \\ -4/5 & 3/5 \end{bmatrix} \begin{bmatrix} 4 \\ 2 \end{bmatrix} = \begin{bmatrix} 4 \\ -2 \end{bmatrix}. \tag{10.39}$$

Verify for yourself that these really *are* the **u**-coordinates of **v**, that is, that the vector **v** really is the same as $4\mathbf{u}_1 + (-2)\mathbf{u}_2$.

10.3.7 Matrix Properties and the Singular Value Decomposition

Because matrices are so closely tied to linear transformations, and because linear transformations are so important in graphics, we'll now briefly discuss some important properties of matrices.

First, **diagonal** matrices—ones with zeroes everywhere except on the diagonal, like the matrix $\mathbf{M}_2$ for the transformation T_2—correspond to remarkably simple transformations: They just scale up or down each axis by some amount (although if the amount is a negative number, the corresponding axis is also flipped). Because of this simplicity, we'll try to understand other transformations in terms of these diagonal matrices.

Second, if the columns of the matrix **M** are $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k \in R^n$, and they are pairwise orthogonal unit vectors, then $\mathbf{M^T M} = \mathbf{I}_k$, the $k \times k$ identity matrix.

In the special case where $k = n$, such a matrix is called **orthogonal.** If the determinant of the matrix is 1, then the matrix is said to be a **special orthogonal** matrix. In $\mathbf{R^2}$, such a matrix must be a rotation matrix like the one in T_1; in $\mathbf{R^3}$, the transformation associated to such a matrix corresponds to rotation around some vector by some amount.[1]

Less familiar to most students, but of enormous importance in much graphics research, is the **singular value decomposition (SVD)** of a matrix. Its existence says, informally, that if we have a transformation T represented by a matrix **M**, and if we're willing to use new coordinate systems on both the domain and codomain, then the transformation simply looks like a nonuniform (or possibly uniform) scaling transformation. We'll briefly discuss this idea here, along with the application of the SVD to solving equations; the web materials for this chapter show the SVD for our example transformations and some further applications of the SVD.

The singular value decomposition theorem says this:

Every $n \times k$ matrix **M** can be factored in the form

$$\mathbf{M} = \mathbf{UDV^T}, \tag{10.40}$$

where **U** is $n \times r$ (where $r = \min(n, k)$) with orthonormal columns, **D** is $r \times r$ diagonal (i.e., only entries of the form d_{ii} can be nonzero), and **V** is $r \times k$ with orthonormal columns (see Figure 10.8).

By convention, the entries of **D** are required to be in nonincreasing order (i.e., $|d_{1,1}| \geq |d_{2,2}| \geq |d_{3,3}| \ldots$) and are indicated by single subscripts (i.e., we write d_1 instead of $d_{1,1}$). They are called the **singular values** of **M**. It turns out that M is degenerate (i.e., singular) exactly if any singular value is 0. As a general

1. As we mentioned in Chapter 3, rotation about a vector in $\mathbf{R^3}$ is better expressed as rotation *in a plane,* so instead of speaking about rotation about z, we speak of rotation in the xy-plane. We can then say that any special orthogonal matrix in $\mathbf{R}^4$ corresponds to a sequence of two rotations in two planes in 4-space.

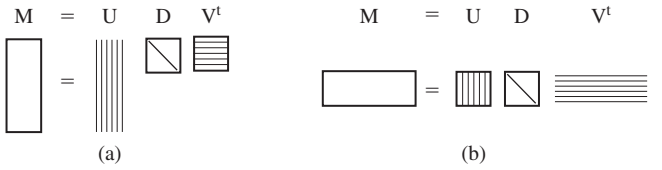


Figure 10.8: (a) An $n \times k$ matrix, with $n > k$, factors as a product of an $n \times n$ matrix with orthonormal columns (indicated by the vertical stripes on the first rectangle), a diagonal $k \times k$ matrix, and a $k \times k$ matrix with orthonormal rows (indicated by the horizontal stripes), which we write as $\mathbf{UDV^T}$, where $\mathbf{U}$ and $\mathbf{V}$ have orthonormal columns. *(b) An $n \times k$ matrix with $n < k$ is written as a similar product; note that the diagonal matrix in both cases is square, and its size is the smaller of n and k.*

guideline, if the ratio of the largest to the smallest singular values is very large (say, 10^6), then numerical computations with the matrix are likely to be unstable.

Inline Exercise 10.16: The singular value decomposition is not unique. If we negate the first row of $\mathbf{V^T}$ and the first column of $\mathbf{U}$ in the SVD of a matrix $\mathbf{M}$, show that the result is still an SVD for $\mathbf{M}$.

In the special case where $n = k$ (the one we most often encounter), the matrices $\mathbf{U}$ and $\mathbf{V}$ are both square and represent change-of-coordinate transformations in the domain and codomain. Thus, we can see the transformation

$$T(\mathbf{x}) = \mathbf{Mx} \tag{10.41}$$

as a sequence of three steps: (1) Multiplication by $\mathbf{V^T}$ converts $\mathbf{x}$ to $\mathbf{v}$-coordinates; (2) multiplication by $\mathbf{D}$ amounts to a possibly nonuniform scaling along each axis; and (3) multiplication by $\mathbf{U}$ treats the resultant entries as coordinates in the $\mathbf{u}$-coordinate system, which then are transformed back to standard coordinates.

10.3.8 Computing the SVD

How do we find $\mathbf{U}$, $\mathbf{D}$, and $\mathbf{V}$? In general it's relatively difficult, and we rely on numerical linear algebra packages to do it for us. Furthermore, the results are by no means unique: A single matrix may have multiple singular value decompositions. For instance, if $\mathbf{S}$ is *any* $n \times n$ matrix with orthonormal columns, then

$$\mathbf{I} = \mathbf{SIS^T} \tag{10.42}$$

is one possible singular value decomposition of the identity matrix. Even though there are many possible SVDs, the singular values are the same for all decompositions.

The **rank** of the matrix $\mathbf{M}$, which is defined as the number of linearly independent columns, turns out to be exactly the number of nonzero singular values.

10.3.9 The SVD and Pseudoinverses

Again, in the special case where $n = k$ so that $\mathbf{U}$ and $\mathbf{V}$ are square, it's easy to compute $\mathbf{M}^{-1}$ if you know the SVD:

$$\mathbf{M}^{-1} = \mathbf{V}D^{-1}\mathbf{U^T}, \tag{10.43}$$

where D^{-1} is easy to compute—you simply invert all the elements of the diagonal. If one of these elements is zero, the matrix is singular and no such inverse exists; in this case, the **pseudoinverse** is also often useful. It's defined as

$$\mathbf{M}^{\dagger} = \mathbf{V}D^{\dagger}\mathbf{U}^{\mathbf{T}}, \tag{10.44}$$

where $D^{\dagger}$ is just D with every nonzero entry inverted (i.e., you try to invert the diagonal matrix D by inverting diagonal elements, and every time you encounter a zero on the diagonal, you ignore it and simply write down 0 in the answer). The definition of the pseudoinverse makes sense even when $n \neq k$; the pseudoinverse can be used to solve "least squares" problems, which frequently arise in graphics.

The Pseudoinverse Theorem:

(a) If $\mathbf{M}$ is an $n \times k$ matrix with $n > k$, the equation $\mathbf{Mx} = \mathbf{b}$ generally represents an overdetermined system of equations[2] which may have no solution. The vector

$$\mathbf{x}_0 = \mathbf{M}^{\dagger}\mathbf{b} \tag{10.45}$$

represents an optimal "solution" to this system, in the sense that $\mathbf{Mx}_0$ is as close to $\mathbf{b}$ as possible.

(b) If $\mathbf{M}$ is an $n \times k$ matrix with $n < k$, and rank n, the equation $\mathbf{Mx} = \mathbf{b}$ represents an underdetermined system of equations.[3] The vector

$$\mathbf{x}_0 = \mathbf{M}^{\dagger}\mathbf{b} \tag{10.46}$$

represents an optimal solution to this system, in the sense that $\mathbf{x}_0$ is the *shortest* vector satisfying $\mathbf{Mx} = \mathbf{b}$.

Here are examples of each of these cases.

Example 1: An overdetermined system

The system

$$\begin{bmatrix}2\\1\end{bmatrix}\begin{bmatrix}t\end{bmatrix} = \begin{bmatrix}4\\3\end{bmatrix} \tag{10.47}$$

has *no* solution: There's simply no number t with $2t = 4$ and $1t = 3$ (see Figure 10.9). But among all the multiples of $\mathbf{M} = \begin{bmatrix}2\\1\end{bmatrix}$, there *is* one that's closest to the vector $\mathbf{b} = \begin{bmatrix}4\\3\end{bmatrix}$, namely $2.2\begin{bmatrix}2\\1\end{bmatrix} = \begin{bmatrix}4.4\\2.2\end{bmatrix}$, as you can discover with elementary geometry. The theorem tells us we can compute this directly, however, using the pseudoinverse. The SVD and pseudoinverse of $\mathbf{M}$ are

$$\mathbf{M} = \mathbf{UDV}^{\mathbf{T}} = \left(\frac{1}{\sqrt{5}}\begin{bmatrix}2\\1\end{bmatrix}\right)\begin{bmatrix}\sqrt{5}\end{bmatrix}\begin{bmatrix}1\end{bmatrix} \tag{10.48}$$

$$\mathbf{M}^{\dagger} = \mathbf{VD}^{\dagger}\mathbf{U} = \begin{bmatrix}1\end{bmatrix}\begin{bmatrix}1/\sqrt{5}\end{bmatrix}\left(\frac{1}{\sqrt{5}}\begin{bmatrix}2 & 1\end{bmatrix}\right) \tag{10.49}$$

$$= \begin{bmatrix}0.4 & 0.2\end{bmatrix}. \tag{10.50}$$

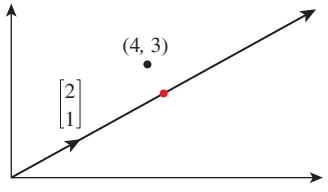


Figure 10.9: The equations $t\begin{bmatrix}2\\1\end{bmatrix} = \begin{bmatrix}4\\3\end{bmatrix}$ *have no common solution. But the multiples of the vector* $\begin{bmatrix}2 & 1\end{bmatrix}^{\mathbf{T}}$ *form a line in the plane that passes by the point* $(4, 3)$*, and there's a point of this line (shown in a red circle on the topmost arrow) that's as close to* $(4, 3)$ *as possible.*

2. In other words, a situation like "five equations in three unknowns."
3. That is, a situation like "three equations in five unknowns."

And the solution guaranteed by the theorem is

$$t = \mathbf{M}^{\dagger}\mathbf{b} = \begin{bmatrix}0.4 & 0.2\end{bmatrix}\begin{bmatrix}4\\3\end{bmatrix} = 2.2. \tag{10.51}$$

Example 2: An underdetermined system

The system

$$\begin{bmatrix}1 & 3\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix} = 4 \tag{10.52}$$

has a great many solutions; any point (x, y) on the line $x + 3y = 4$ is a solution (see Figure 10.10). The solution that's *closest to the origin* is the point on the line $x + 3y = 4$ that's as near to $(0, 0)$ as possible, which turns out to be $x = 0.4; y = 1.2$. In this case, the matrix $\mathbf{M}$ is $\begin{bmatrix}1 & 3\end{bmatrix}$; its SVD and pseudoinverse are simply

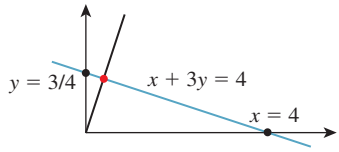


Figure 10.10: Any point of the blue line is a solution; the red point is closest to the origin.

$$\mathbf{M} = \mathbf{U}\mathbf{D}\mathbf{V}^{\mathbf{T}} = \begin{bmatrix}1\end{bmatrix}\begin{bmatrix}\sqrt{10}\end{bmatrix}\begin{bmatrix}1/\sqrt{10} & 3/\sqrt{10}\end{bmatrix} \text{ and} \tag{10.53}$$

$$\mathbf{M}^{\dagger} = \mathbf{V}\mathbf{D}^{\dagger}\mathbf{U} = \begin{bmatrix}1/\sqrt{10}\\3/\sqrt{10}\end{bmatrix}\begin{bmatrix}1/\sqrt{10}\end{bmatrix}\begin{bmatrix}1\end{bmatrix} = \begin{bmatrix}1/10\\3/10\end{bmatrix}. \tag{10.54}$$

And the solution guaranteed by the theorem is

$$\mathbf{M}^{\dagger}\mathbf{b} = \begin{bmatrix}1/10\\3/10\end{bmatrix}\begin{bmatrix}4\end{bmatrix} = \begin{bmatrix}0.4\\1.2\end{bmatrix}. \tag{10.55}$$

Of course, this kind of computation is much more interesting in the case where the matrices are much larger, but all the essential characteristics are present even in these simple examples.

A particularly interesting example arises when we have, for instance, two polyhedral models (consisting of perhaps hundreds of vertices joined by triangular faces) that might be "essentially identical": One might be just a translated, rotated, and scaled version of the other. In Section 10.4, we'll see how to represent translation along with rotation and scaling in terms of matrix multiplication. We can determine whether the two models are in fact essentially identical by listing the coordinates of the first in the columns of a matrix $\mathbf{V}$ and the coordinates of the second in a matrix $\mathbf{W}$, and then seeking a matrix $\mathbf{A}$ with

$$\mathbf{A}\mathbf{V} = \mathbf{W}. \tag{10.56}$$

This amounts to solving the "overconstrained system" problem; we find that $\mathbf{A} = \mathbf{V}^{\dagger}\mathbf{W}$ is the best possible solution. If, having computed $\mathbf{A}$, we find that

$$\mathbf{A}\mathbf{V} = \mathbf{W}, \tag{10.57}$$

then the models are essentially identical; if the left and right sides differ, then the models are not essentially identical. (This entire approach depends, of course, on corresponding vertices of the two models being listed in the corresponding order; the more general problem is a lot more difficult.)

10.4 Translation

We now describe a way to apply linear transformations to generate *translations,* and at the same time give a nice model for the points-versus-vectors ideas we've espoused so far.

The idea is this: As our Euclidean plane (our set of *points*), we'll take the plane $w = 1$ in xyw-space (see Figure 10.11). The use of w here is in preparation for what we'll do in 3-space, which is to consider the three-dimensional set defined by $w = 1$ in $xyzw$-space.

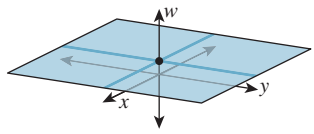


Figure 10.11: The $w = 1$ plane in xyw-space.

Having done this, we can consider transformations that multiply such vectors by a 3×3 matrix $\mathbf{M}$. The only problem is that the result of such a multiplication may not have a 1 as its last entry. We can restrict our attention to those that do:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ p & q & r \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}. \tag{10.58}$$

For this equation to hold for every x and y, we must have $px + qy + r = 1$ for all x, y. This forces $p = q = 0$ and $r = 1$.

Thus, we'll consider transformations of the form

$$\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}. \tag{10.59}$$

If we examine the special case where the upper-left corner is a 2×2 identity matrix, we get

$$\begin{bmatrix} 1 & 0 & c \\ 0 & 1 & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x+c \\ y+f \\ 1 \end{bmatrix}. \tag{10.60}$$

As long as we pay attention only to the x- and y-coordinates, this looks like a translation! We've added c to each x-coordinate and f to each y-coordinate (see Figure 10.12). Transformations like this, restricted to the plane $w = 1$, are called **affine transformations** of the plane. Affine transformations are the ones most often used in graphics.

On the other hand, if we make $c = f = 0$, then the third coordinate becomes irrelevant, and the upper-left 2×2 matrix can perform any of the operations we've seen up until now. Thus, with the simple trick of adding a third coordinate and requiring that it always be 1, we've managed to unify rotation, scaling, and all the other linear transformations with the new class of transformations, *translations,* to get the class of affine transformations.

10.5 Points and Vectors Again

Back in Chapter 7, we said that points and vectors could be combined in certain ways: The difference of points is a vector, a vector could be added to a point

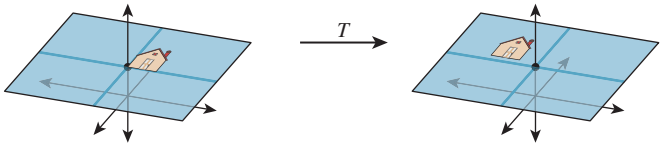


Figure 10.12: The house figure, before and after a translation generated by shearing *parallel to the $w = 1$ plane.*

to get a new point, and more generally, affine combinations of points, that is, combinations of the form

$$\alpha_1 P_1 + \alpha_2 P_2 + \ldots + \alpha_k P_k, \tag{10.61}$$

were allowed if and only if $\alpha_1 + \alpha_2 + \ldots + \alpha_k = 1$.

We now have a situation in which these distinctions make sense in terms of familiar mathematics: We can regard *points* of the plane as being elements of $\mathbf{R}^3$ whose third coordinate is 1, and *vectors* as being elements of $\mathbf{R}^3$ whose third coordinate is 0.

With this convention, it's clear that the difference of points is a vector, the sum of a vector and a point is a point, and combinations like the one in Equation 10.61 yield a point if and only if the sum of the coefficients is 1 (because the third coordinate of the result will be exactly the sum of the coefficients; for the sum to be a *point,* this third coordinate is required to be 1).

You may ask, "Why, when we're already familiar with vectors in 3-space, should we bother calling some of them 'points in the Euclidean plane' and others 'two-dimensional vectors'?" The answer is that the distinctions have geometric significance when we're using this subset of 3-space as a model for 2D transformations. Adding vectors in 3-space is defined in linear algebra, but adding together two of our "points" gives a location in 3-space that's not on the $w = 1$ plane or the $w = 0$ plane, so we don't have a name for it at all.

Henceforth we'll use E^2 (for "Euclidean two-dimensional space") to denote this $w = 1$ plane in xyw-space, and we'll write (x, y) to mean the point of E^2 corresponding to the 3-space vector $\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$. It's conventional to speak of an affine transformation as acting on E^2, even though it's defined by a 3×3 matrix.

10.6 Why Use 3 × 3 Matrices Instead of a Matrix and a Vector?

Students sometimes wonder why they can't just represent a linear transformation plus translation in the form

$$T(\mathbf{x}) = \mathbf{M}\mathbf{x} + \mathbf{b}, \tag{10.62}$$

where the matrix $\mathbf{M}$ represents the linear part (rotating, scaling, and shearing) and $\mathbf{b}$ represents the translation.

First, you *can* do that, and it works just fine. You might save a tiny bit of storage (four numbers for the matrix and two for the vector, so six numbers instead of nine), but since our matrices always have two 0s and a 1 in the third column, we don't really need to store that column anyhow, so it's the same. Otherwise, there's no important difference.

Second, the reason to unify the transformations into a single matrix is that it's then very easy to take multiple transformations (each represented by a matrix) and **compose** them (perform one after another): We just multiply their matrices together in the right order to get the matrix for the composed transformation. You can do this in the matrix-and-vector formulation as well, but the programming is slightly messier and more error-prone.

There's a third reason, however: It'll soon become apparent that we can also work with triples whose third entry is neither 1 nor 0, and use the operation of **homogenization** (dividing by w) to convert these to points (i.e., triples with $w = 1$), except when $w = 0$. This allows us to study even more transformations, one of which is central to the study of perspective, as we'll see later.

The singular value decomposition provides the tool necessary to decompose not just linear transformations, but affine ones as well (i.e., combinations of linear transformations and translations).

10.7 Windowing Transformations

As an application of our new, richer set of transformations, let's examine **windowing transformations,** which send one axis-aligned rectangle to another, as shown in Figure 10.13. (We already discussed this briefly in Chapter 3.)

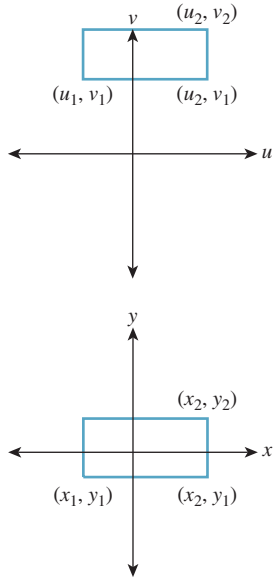


Figure 10.13: Window transformation setup. We need to move the uv-rectangle to the xy-rectangle.

We'll first take a direct approach involving a little algebra. We'll then examine a more automated approach.

We'll need to do essentially the same thing to the first and second coordinates, so let's look at how to transform the first coordinate only. We need to send u_1 to x_1 and u_2 to x_2. That means we need to scale up any coordinate *difference* by the factor $\frac{x_2-x_1}{u_2-u_1}$. So our transformation for the first coordinate has the form

$$t \mapsto \frac{x_2 - x_1}{u_2 - u_1} t + \text{something}. \tag{10.63}$$

If we apply this to $t = u_1$, we know that we want to get x_1; this leads to the equation

$$\frac{x_2 - x_1}{u_2 - u_1} u_1 + \text{something} = x_1. \tag{10.64}$$

Solving for the missing offset gives

$$\begin{aligned} x_1 - \frac{x_2 - x_1}{u_2 - u_1} u_1 &= x_1 \frac{u_2 - u_1}{u_2 - u_1} - \frac{x_2 - x_1}{u_2 - u_1} u_1 && (10.65)\\ &= \frac{x_1 u_2 - x_1 u_1 - x_2 u_1 + x_1 u_1}{u_2 - u_1} && (10.66)\\ &= \frac{x_1 u_2 - x_2 u_1}{u_2 - u_1}, && (10.67) \end{aligned}$$

so that the transformation is

$$t \mapsto \frac{x_2 - x_1}{u_2 - u_1} t + \frac{x_1 u_2 - x_2 u_1}{u_2 - u_1}. \tag{10.68}$$

Doing essentially the same thing for the v and y terms (i.e., the second coordinate) we get the transformation, which we can write in matrix form:

$$T(\mathbf{x}) = \mathbf{M}\mathbf{x}, \tag{10.69}$$

where

$$\mathbf{M} = \begin{bmatrix} \frac{x_2-x_1}{u_2-u_1} & 0 & \frac{x_1u_2-x_2u_1}{u_2-u_1} \\ 0 & \frac{y_2-y_1}{v_2-v_1} & \frac{y_1v_2-y_2v_1}{v_2-v_1} \\ 0 & 0 & 1 \end{bmatrix}. \tag{10.70}$$

Inline Exercise 10.17: Multiply the matrix $\mathbf{M}$ of Equation 10.70 by the vector $\begin{bmatrix} u_1 & v_1 & 1 \end{bmatrix}^{\mathbf{T}}$ to confirm that you do get $\begin{bmatrix} x_1 & y_1 & 1 \end{bmatrix}^{\mathbf{T}}$. Do the same for the opposite corner of the rectangle.

We'll now show you a second way to build this transformation (and many others as well).

10.8 Building 3D Transformations

Recall that in 2D we could send the vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ to the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ by building a matrix $\mathbf{M}$ whose columns were $\mathbf{v}_1$ and $\mathbf{v}_2$, and then use two such matrices (inverting one along the way) to send any two independent vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ to any two vectors $\mathbf{w}_1$ and $\mathbf{w}_2$. We can do the same thing in 3-space: We can send the standard basis vectors $\mathbf{e}_1, \mathbf{e}_2$, and $\mathbf{e}_3$ to any three other vectors, just by using those vectors as the columns of a matrix. Let's start by sending $\mathbf{e}_1$, $\mathbf{e}_2$, and $\mathbf{e}_3$ to three corners of our first rectangle—the two we've already specified and the lower-right one, at location (u_2, v_1). The three vectors corresponding to these points are

$$\begin{bmatrix} u_1 \\ v_1 \\ 1 \end{bmatrix}, \quad \begin{bmatrix} u_2 \\ v_2 \\ 1 \end{bmatrix}, \text{ and } \begin{bmatrix} u_2 \\ v_1 \\ 1 \end{bmatrix}. \tag{10.71}$$

Because the three corners of the rectangle are not collinear, the three vectors are independent. Indeed, this is our definition of independence for vectors in n-space: Vectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$ are independent if there's no $(k-1)$-dimensional subspace containing them. In 3-space, for instance, three vectors are independent if there's no plane through the origin containing all of them.

So the matrix

$$\mathbf{M}_1 = \begin{bmatrix} u_1 & u_2 & u_2 \\ v_1 & v_2 & v_1 \\ 1 & 1 & 1 \end{bmatrix}, \tag{10.72}$$

which performs the desired transformation, will be invertible.

We can similarly build the matrix $\mathbf{M}_2$, with the corresponding xs and ys in it. Finally, we can compute

$$\mathbf{M}_2\mathbf{M}_1^{-1}, \tag{10.73}$$

which will perform the desired transformation. For instance, the lower-left corner of the starting rectangle will be sent, by $\mathbf{M}_1^{-1}$, to $\mathbf{e}_1$ (because $\mathbf{M}_1$ sent $\mathbf{e}_1$ to the lower-left corner); multiplying $\mathbf{e}_1$ by $\mathbf{M}_2$ will send it to the lower-left corner of the target rectangle. A similar argument applies to all three corners. Indeed, if we compute the inverse algebraically and multiply out everything, we'll once again arrive at the matrix given in Equation 10.7. But we don't need to do so: We know that this must be the right matrix. Assuming we're willing to use a matrix-inversion routine, there's no need to think through anything more than "I want these three points to be sent to these three other points."

Summary: Given any three noncollinear points P_1, P_2, P_3 in E^2, we can find a matrix transformation and send them to any three points Q_1, Q_2, Q_3 with the procedure above.

10.9 Another Example of Building a 2D Transformation

Suppose we want to find a 3×3 matrix transformation that rotates the entire plane $30°$ counterclockwise around the point $P = (2, 4)$, as shown in Figure 10.14. As you'll recall, WPF expresses this transformation via code like this:

```
<RotateTransform Angle="-30" CenterX="2" CenterY="4"/>
```

An implementer of WPF then must create a matrix like the one we're about to build.

Here are two approaches.

First, we know how to rotate about the origin by $30°$; we can use the transformation T_1 from the start of the chapter. So we can do our desired transformation in three steps (see Figure 10.15).

1. Move the point $(2, 4)$ to the origin.
2. Rotate by $30°$.
3. Move the origin back to $(2, 4)$.

The matrix that moves the point $(2, 4)$ to the origin is

$$\begin{bmatrix} 1 & 0 & -2 \\ 0 & 1 & -4 \\ 0 & 0 & 1 \end{bmatrix}. \tag{10.74}$$

The one that moves it back is similar, except that the 2 and 4 are not negated. And the rotation matrix (expressed in our new 3×3 format) is

$$\begin{bmatrix} \cos 30° & -\sin 30° & 0 \\ \sin 30° & \cos 30° & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{10.75}$$

The matrix representing the entire sequence of transformations is therefore

$$\begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos 30° & -\sin 30° & 0 \\ \sin 30° & \cos 30° & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -2 \\ 0 & 1 & -4 \\ 0 & 0 & 1 \end{bmatrix}. \tag{10.76}$$

Inline Exercise 10.18: (a) Explain why this is the correct order in which to multiply the transformations to get the desired result.
(b) Verify that the point $(2, 4)$ is indeed left unmoved by multiplying $\begin{bmatrix} 2 & 4 & 1 \end{bmatrix}^{\mathbf{T}}$ by the sequence of matrices above.

The second approach is again more automatic: We find three points whose target locations we know, just as we did with the windowing transformation above. We'll use $P = (2, 4)$, $Q = (3, 4)$ (the point one unit to the right of P), and $R = (2, 5)$ (the point one unit above P). We know that we want P sent to P, Q sent to $(2 + \cos 30°, 4 + \sin 30°)$, and R sent to $(2 - \sin 30°, 4 + \cos 30°)$. (Draw a picture to convince yourself that these are correct). The matrix that achieves this is just

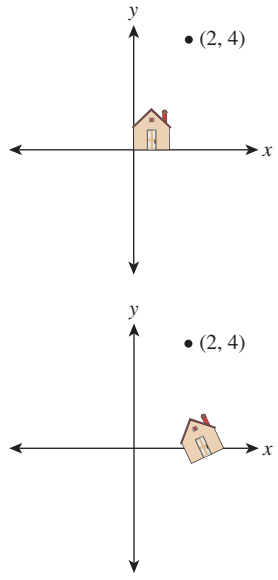


Figure 10.14: We'd like to rotate the entire plane by $30°$ counterclockwise about the point $P = (2, 4)$.

$$\begin{bmatrix} 2 & 2+\cos 30^\circ & 4-\sin 30^\circ \\ 4 & 4+\sin 30^\circ & 4+\cos 30^\circ \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 3 & 2 \\ 4 & 4 & 5 \\ 1 & 1 & 1 \end{bmatrix}^{-1}. \tag{10.77}$$

Both approaches are reasonably easy to work with.

There's a third approach—a variation of the second—in which we specify where we want to send a point and two vectors, rather than three points. In this case, we might say that we want the point P to remain fixed, and the vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ to go to

$$\begin{bmatrix} \cos 30^\circ \\ \sin 30^\circ \\ 0 \end{bmatrix} \text{ and } \begin{bmatrix} -\sin 30^\circ \\ \cos 30^\circ \\ 0 \end{bmatrix}, \tag{10.78}$$

respectively. In this case, instead of finding matrices that send the vectors $\mathbf{e}_1, \mathbf{e}_2$, and $\mathbf{e}_3$ to the desired three points, before and after, we find matrices that send those vectors to the desired point and two vectors, before and after. These matrices are

$$\begin{bmatrix} 2 & 1 & 0 \\ 4 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \text{ and } \begin{bmatrix} 2 & \cos 30^\circ & -\sin 30^\circ \\ 4 & \sin 30^\circ & \cos 30^\circ \\ 1 & 0 & 0 \end{bmatrix}, \tag{10.79}$$

so the overall matrix is

$$\begin{bmatrix} 2 & \cos 30^\circ & -\sin 30^\circ \\ 4 & \sin 30^\circ & \cos 30^\circ \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 4 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}^{-1}. \tag{10.80}$$

These general techniques can be applied to create any linear-plus-translation transformation of the $w = 1$ plane, but there are some specific ones that are good to know. Rotation in the xy-plane, by an amount θ (rotating the positive x-axis toward the positive y-axis) is given by

$$R_{xy}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{10.81}$$

In some books and software packages, this is called **rotation around z;** we prefer the term "rotation in the xy-plane" because it also indicates the direction of rotation (from x, toward y). The other two standard rotations are

$$R_{yz}(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \tag{10.82}$$

and

$$R_{zx}(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}; \tag{10.83}$$

note that the last expression rotates z toward x, and *not* the opposite. Using this naming convention helps keep the pattern of plusses and minuses symmetric.

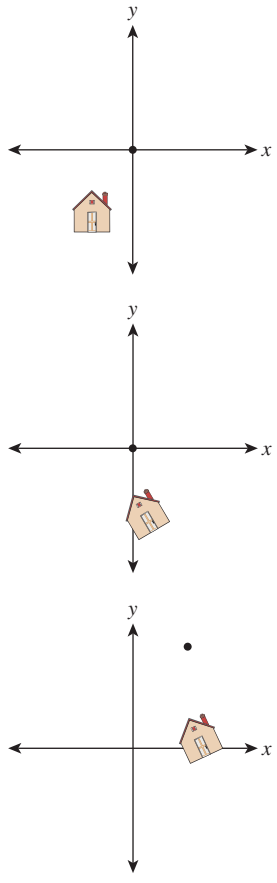


Figure 10.15: The house after translating $(2,4)$ to the origin, after rotating by 30°, and after translating the origin back to $(2,4)$.

10.10 Coordinate Frames

In 2D, a linear transformation is completely specified by its values on two independent vectors. An affine transformation (i.e., linear plus translation) is completely specified by its values on any three noncollinear points, or on any point and pair of independent vectors. A projective transformation on the plane (which we'll discuss briefly in Section 10.13) is specified by its values on four points, no three collinear, or on other possible sets of points and vectors. These facts, and the corresponding ones for transformations on 3-space, are so important that we enshrine them in a principle:

> ✓ **THE TRANSFORMATION UNIQUENESS PRINCIPLE:** For each class of transformations—linear, affine, and projective—and any corresponding coordinate frame, and any set of corresponding target elements, there's a unique transformation mapping the frame elements to the correponding elements in the target frame. If the target elements themselves constitute a frame, then the transformation is invertible.

To make sense of this, we need to define a **coordinate frame.** As a first example, a coordinate frame for linear transformations is just a "basis": In two dimensions, that means "two linearly independent vectors in the plane." The elements of the frame are the two vectors. So the principle says that if $\mathbf{u}$ and $\mathbf{v}$ are linearly independent vectors in the plane, and $\mathbf{u}'$ and $\mathbf{v}'$ are any two vectors, then there's a unique linear transformation sending $\mathbf{u}$ to $\mathbf{u}'$ and $\mathbf{v}$ to $\mathbf{v}'$. It further says that if $\mathbf{u}'$ and $\mathbf{v}'$ are independent, then the transformation is invertible.

More generally, a **coordinate frame** is a set of geometric elements rich enough to uniquely characterize a transformation in some class. For linear transformations of the plane, a coordinate frame consists of two independent vectors in the plane, as we said; for affine transforms of the plane, it consists of three noncollinear points in the plane, *or* of one point and two independent vectors, etc.

In cases where there are multiple kinds of coordinate frames, there's always a way to convert between them. For 2D affine transformations, the three noncollinear points P, Q, and R can be converted to $P, \mathbf{v}_1 = Q - P$, and $\mathbf{v}_2 = R - P$; the conversion in the other direction is obvious. (It may not be obvious that the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ are linearly independent. See Exercise 10.4.)

There's a restricted use of "coordinate frame" for affine maps that has some advantages. Based on the notion that the origin and the unit vectors along the positive directions for each axis form a frame, we'll say that a **rigid coordinate frame** for the plane is a triple $(P, \mathbf{v}_1, \mathbf{v}_2)$, where P is a point and $\mathbf{v}_1$ and $\mathbf{v}_2$ are *perpendicular* unit vectors with the rotation from $\mathbf{v}_1$ toward $\mathbf{v}_2$ being counterclockwise (i.e., with $\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{v}_1 = \mathbf{v}_2$). The corresponding definition for 3-space has one point and three mutually perpendicular unit vectors forming a right-hand coordinate system. Transforming one rigid coordinate frame $(P, \mathbf{v}_1, \mathbf{v}_2)$ to another $(Q, \mathbf{u}_1, \mathbf{u}_2)$ can always be effected by a sequence of transformation,

$$T_Q \circ R \circ T_P^{-1}, \tag{10.84}$$

where $T_P(A) = A + P$ is translation by P, and similarly for T_Q, and R is the rotation given by

$$R = [\mathbf{u}_1; \mathbf{u}_2] \cdot [\mathbf{v}_1; \mathbf{v}_2]^{\mathbf{T}}, \tag{10.85}$$

where the semicolon indicates that $\mathbf{u}_1$ is the first column of the first factor, etc.

The G3D library, which we use in examples in Chapters 12, 15, and 32, uses rigid coordinate frames extensively in modeling, encapsulating them in a class, `CFrame`.

10.11 Application: Rendering from a Scene Graph

Figure 10.16: Our clock model.

We've discussed affine transformations on a two-dimensional affine space, and how, once we have a coordinate system and can represent points as triples, as in $\mathbf{x} = \begin{bmatrix} x & y & 1 \end{bmatrix}^{\mathbf{T}}$, we can represent a transformation by a 3×3 matrix $\mathbf{M}$. We transform the point $\mathbf{x}$ by multiplying it on the left by $\mathbf{M}$ to get $\mathbf{Mx}$. With this in mind, let's return to the clock example of Chapter 2 and ask how we could start from a WPF description and convert it to an image, that is, how we'd do some of the work that WPF does. You'll recall that the clock shown in Figure 10.16 was created in WPF with code like this,

```
1  <Canvas ... >
2    <Ellipse
3       Canvas.Left="-10.0" Canvas.Top="-10.0"
4       Width="20.0" Height="20.0"
5       Fill="lightgray" />
6    <Control Name="Hour Hand" .../>
7    <Control Name="Minute Hand" .../>
8    <Canvas.RenderTransform>
9       <TransformGroup>
10       <ScaleTransform ScaleX="4.8" ScaleY="4.8" />
11       <TranslateTransform X="48" Y="48" />
12    </TransformGroup>
13   </Canvas.RenderTransform>
14 </Canvas>
```

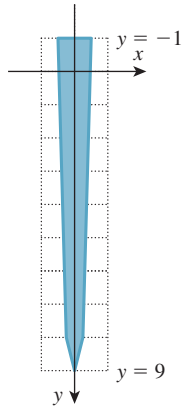


Figure 10.17: The clock-hand template.

where the code for the hour hand is

```
1  <Control Name="HourHand" Template="{StaticResource ClockHandTemplate}">
2    <Control.RenderTransform>
3      <TransformGroup>
4        <ScaleTransform ScaleX="1.7" ScaleY="0.7" />
5        <RotateTransform Angle="180"/>
6        <RotateTransform x:Name="ActualTimeHour" Angle="0"/>
7      </TransformGroup>
8    </Control.RenderTransform>
9  </Control>
```

and the code for the minute hand is similar, the only differences being that `ActualTimeHour` is replaced by `ActualTimeMinute` and the scale by 1.7 in X and 0.7 in Y is omitted.

The `ClockHandTemplate` was a polygon defined by five points in the plane: $(-0.3, -1), (-0.2, 8), (0, 9), (0.2, 8)$, and $(0.3, -1)$ (see Figure 10.17).

We're going to slightly modify this code so that the clock face and clock hands are both described in the same way, as polygons. We *could* create a polygonal version of the circular face by making a regular polygon with, say, 1000 vertices, but to keep the code simple and readable, we'll make an octagonal approximation of a circle instead.

Now the code begins like this:

```
<Canvas ...
    <Canvas.Resources>
      <ControlTemplate x:Key="ClockHandTemplate">
        <Polygon
          Points="-0.3,-1   -0.2,8   0,9   0.2,8   0.3,-1"
           Fill="Navy"/>
      </ControlTemplate>
      <ControlTemplate x:Key="CircleTemplate">
        <Polygon
           Points="1,0   0.707,0.707   0,1    -.707,.707
                  -1,0   -.707,-.707   0,-1   0.707,-.707"
           Fill="LightGray"/>
      </ControlTemplate>
    </Canvas.Resources>
```

This code defines the geometry that we'll use to create the face and hands of the clock. With this change, the circular clock face will be defined by transforming a template "circle," represented by eight evenly spaced points on the unit circle. This form of specification, although not idiomatic in WPF, is quite similar to scene specification in many other scene-graph packages.

The actual creation of the scene now includes building the clock face from the `CircleTemplate`, and building the hands as before.

```
  <!- 1. Background of the clock ->
  <Control Name="Face"
           Template="{StaticResource CircleTemplate}">
     <Control.RenderTransform>
           <ScaleTransform ScaleX="10" ScaleY="10" />
     </Control.RenderTransform>
  </Control>

  <!- 2. The minute hand ->
  <Control Name="MinuteHand"
           Template="{StaticResource ClockHandTemplate}">
     <Control.RenderTransform>
        <TransformGroup>
           <RotateTransform Angle="180" />
           <RotateTransform x:Name="ActualTimeMinute" Angle="0" />
        </TransformGroup>
     </Control.RenderTransform>
  </Control>

  <!- 3. The hour hand ->
  <Control Name="HourHand" Template="{StaticResource ClockHandTemplate}">
     <Control.RenderTransform>
        <TransformGroup>
          <ScaleTransform ScaleX="1.7" ScaleY="0.7" />
          <RotateTransform Angle="180" />
          <RotateTransform x:Name="ActualTimeHour"
                           Angle="0" />
        </TransformGroup>
     </Control.RenderTransform>
  </Control>
```

All that remains is the transformation from `Canvas` to WPF coordinates, and the timers for the animation, which set the `ActualTimeMinute` and `ActualTimeHour` values.

```
  <Canvas.RenderTransform>
    ...same as before...
   </Canvas.RenderTransform>

  <Canvas.Triggers>
    <EventTrigger RoutedEvent="FrameworkElement.Loaded">
       <BeginStoryboard>
          <Storyboard>
            <DoubleAnimation
              Storyboard.TargetName="ActualTimeHour"
              Storyboard.TargetProperty="Angle"
              From="0.0" To="360.0"
              Duration="00:00:01:0" RepeatBehavior="Forever"
              />
            <DoubleAnimation
              Storyboard.TargetName="ActualTimeMinute"
              Storyboard.TargetProperty="Angle"
              From="0.0" To="4320.0"
              Duration="00:00:01:0" RepeatBehavior="Forever"
              />
          </Storyboard>
       </BeginStoryboard>
    </EventTrigger>
  </Canvas.Triggers>

</Canvas>
```

As a starting point in transforming this scene description into an image, we'll assume that we have a basic graphics library that, given an array of points representing a polygon, can draw that polygon. The points will be represented by a $3 \times k$ array of homogeneous coordinate triples, so the first column of the array will be the homogeneous coordinates of the first polygon point, etc.

We'll now explain how we can go from something like the WPF description to a sequence of `drawPolygon` calls. First, let's transform the XAML code into a tree structure, as shown in Figure 10.18, representing the scene graph (see Chapter 6).

We've drawn transformations as diamonds, geometry as blue boxes, and named parts as beige boxes. For the moment, we've omitted the matter of instancing of the `ClockHandTemplate` and pretended that we have two separate identical copies of the geometry for a clock hand. We've also drawn next to each transformation the matrix representation of the transformation. We've assumed that the angle in `ActualTimeHour` is 15° (whose cosine and sine are approximately 0.96 and 0.26, respectively) and the angle in `ActualTimeMinutes` is 180° (i.e., the clock is showing 12:30).

Inline Exercise 10.19: (a) Remembering that rotations in WPF are specified in degrees and that they rotate objects in a *clockwise* direction, check that the matrix given for the rotation of the hour hand by 15° is correct.
(b) If you found that the matrix was wrong, recall that in WPF x increases to the right and y increases *down.* Does this change your answer? By the way, if you ran this program in WPF and debugged it and printed the matrix, you'd find the negative sign on the $(2, 1)$ entry instead of the $(1, 2)$ entry. That's because WPF internally uses row vectors to represents points, and multiplies them by transformation matrices on the right.

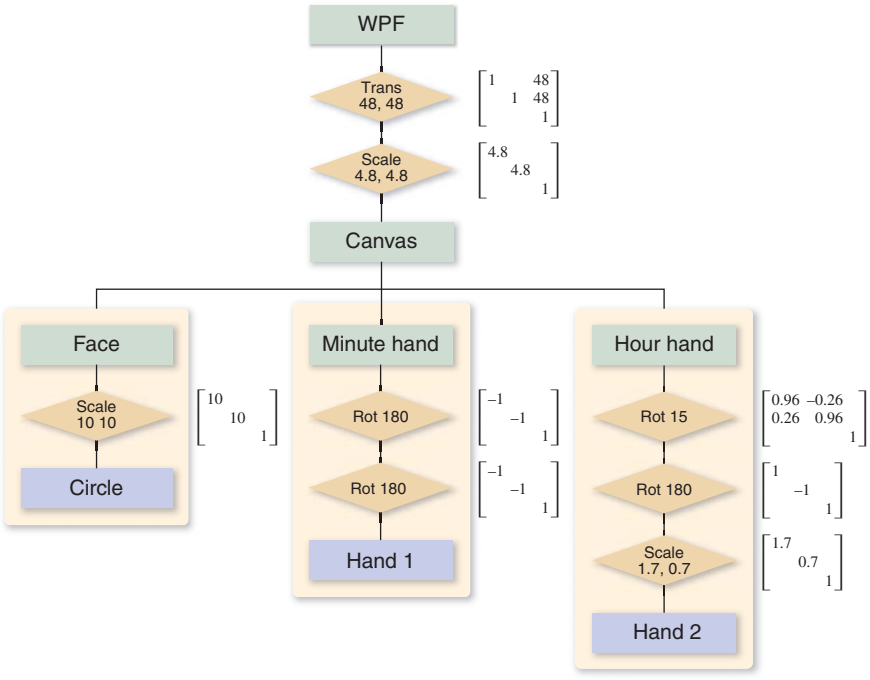


Figure 10.18: A scene-graph representation of the XAML code for the clock.

The order of items in the tree is a little different from the textual order, but there's a natural correspondence between the two. If you consider the hour hand and look at all transformations that occur in its associated render transform or in the render transform of anything containing it (i.e., the whole clock), those are exactly the transforms you encounter as you read from the leaf node corresponding to the hour hand up toward the root node.

Inline Exercise 10.20: Write down all transformations applied to the circle template that's used as the clock face by reading the XAML program. Confirm that they're the same ones you get by reading upward from the "Circle" box in Figure 10.18.

In the scene graph we've drawn, the transformation matrices are the most important elements. We're now going to discuss how these matrices and the coordinates of the points in the geometry nodes interact.

Recall that there are two ways to think about transformations. The first is to say that the minute hand, for instance, has a rotation operation applied to each of its points, creating a new minute hand, which in turn has a translation applied to each point, creating yet another new minute hand, etc. The tip of the minute hand is at location $(0, 9)$, once and for all. The tip of the *rotated* minute hand is somewhere else, and the tip of the translated and rotated minute hand is somewhere else again. It's common to talk about all of these as if they were the same thing ("*Now* the tip of the minute hand is at $(3, 17)\ldots$"), but that doesn't really make sense—the tip of the minute hand cannot be in two different places.

The second view says that there are several different coordinate systems, and that the transformations tell you how to get from the tip's coordinates in one system to its coordinates in another. We can then say things like, "The tip of the minute hand is at $(0, 9)$ in **object space** or **object coordinates,** but it's at $(0, -9)$ in canvas coordinates." Of course, the position in canvas coordinates depends on the amount by which the tip of the minute hand is rotated (we've assumed that the `ActualTimeMinute` rotation is 180°, so it has just undergone two 180° rotations). Similarly, the WPF coordinates for the tip of the minute hand are computed by further scaling each canvas coordinate by 4.8, and then adding 48 to each, resulting in WPF coordinates of $(48, 4.8)$.

The terms **object space, world space, image space,** and **screen space** are frequently used in graphics. They refer to the idea that a single point of some object (e.g., "Boston" on a texture-mapped globe) starts out as a point on a unit sphere (object space), gets transformed into the "world" that we're going to render, eventually is projected onto an image plane, and finally is displayed on a screen. In some sense, all those points refer to the same thing. But each point has different coordinates. When we talk about a certain point "in world space" or "in image space," we really mean that we're working with the coordinates of the point in a coordinate system associated with that space. In image space, those coordinates may range from -1 to 1 (or from 0 to 1 in some systems), while in screen space, they may range from 0 to 1024, and in object space, the coordinates are a triple of real numbers that are typically in the range $[-1, 1]$ for many standard objects like the sphere or cube.

For this example, we have seven coordinate systems, most indicated by pale green boxes. Starting at the top, there are WPF coordinates, the coordinates used by `drawPolygon()`. It's possible that internally, `drawPolygon()` must convert to, say, pixel coordinates, but this conversion is hidden from us, and we won't discuss it further. Beneath the WPF coordinates are canvas coordinates, and within the canvas are the clock-face coordinates, minute-hand coordinates, and hour-hand coordinates. Below this are the hand coordinates, the coordinate system in which the single prototype hand was created, and circle coordinates, in which the prototype octagonal circle approximation was created. Notice that in our model of the clock, the clock-face, minute-hand, and hour-hand coordinates all play similar roles: In the hierarchy of coordinate systems, they're all children of the canvas coordinate system. It might also have been reasonable to make the minute-hand and hour-hand coordinate systems children of the clock-face coordinate system. The advantage of doing so would have been that translating the clock face would have translated the whole clock, making it easier to adjust the clock's position on the canvas. Right now, adjusting the clock's position on the canvas requires that we adjust three different translations, which we'd have to add to the face, the minute hand, and the hour hand.

We're hoping to draw each shape with a `drawPolygon()` call, which takes an array of point coordinates as an argument. For this to make sense, we have to declare the coordinate system in which the point coordinates are valid. We'll assume that `drawPolygon()` expects WPF coordinates. So when we want to tell it about the tip of the minute hand, we'll need the numbers $(48, 4.8)$ rather than $(0, 9)$.

Here's a strawman algorithm for converting a scene graph into a sequence of `drawPolygon()` calls. We'll work with $3 \times k$ arrays of coordinates, because we'll represent the point $(0, 9)$ as a homogeneous triple $(0, 9, 1)$, which we'll write vertically as a column of the matrix that represents the geometry.

```
1  for each polygonal geometry element, g
2     let v be the 3 × k array of vertices of g
3     let n be the parent node of g
4     let M be the 3 × 3 identity matrix
5     while (n is not the root)
6        if n is a transformation with matrix S
7           M = SM
8        n = parent of n
9
10    w = Mv
11    drawPolygon(w)
```

As you can see, we multiply together several matrices, and then multiply the result (the **composite transformation matrix**) by the vertex coordinates to get the WPF coordinates for each polygon, which we then draw.

Inline Exercise 10.21: (a) How many elementary operations are needed, approximately, to multiply a 3×3 matrix by a $3 \times k$ matrix?
(b) If $\mathbf{A}$ and $\mathbf{B}$ are 3×3 and $\mathbf{C}$ is 3×1000, would you rather compute $(\mathbf{AB})\mathbf{C}$ or $\mathbf{A}(\mathbf{BC})$, where the parentheses are meant to indicate the order of calculations that you perform?
(c) In the code above, should we have multiplied the vertex coordinates by each matrix in turn, or was it wiser to accumulate the matrix product and only multiply by the vertex array at the end? Why?

If we hand-simulate the code in the clock example, the circle template coordinates are multiplied by the matrix

$$\begin{bmatrix} 1 & 0 & 48 \\ 0 & 1 & 48 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4.8 & 0 & 0 \\ 0 & 4.8 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 10 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{10.86}$$

The minute-hand template coordinates are multiplied by the matrix

$$\begin{bmatrix} 1 & 0 & 48 \\ 0 & 1 & 48 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4.8 & 0 & 0 \\ 0 & 4.8 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{10.87}$$

And the hour-hand template coordinates are multiplied by the matrix

$$\begin{bmatrix} 1 & 0 & 48 \\ 0 & 1 & 48 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4.8 & 0 & 0 \\ 0 & 4.8 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.96 & -0.26 & 0 \\ 0.26 & 0.96 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1.7 & 0 & 0 \\ 0 & 0.7 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{10.88}$$

Inline Exercise 10.22: Explain where each of the matrices for the minute hand arose.

Notice how much of this matrix multiplication is *shared.* We could have computed the product for the circle and reused it in each of the others, for instance. For a large scene graph, the overlap is often much greater. If there are 70 transformations applied to an object with only five or six vertices, the cost of multiplying matrices together far outweighs the cost of multiplying the composite matrix by the vertex coordinate array.

We can avoid duplicated work by revising our strawman algorithm. We perform a depth-first traversal of the scene graph, maintaining a stack of matrices as we do so. Each time we encounter a new transformation with matrix **M**, we multiply **M** by the current transformation matrix **C** (the one at the top of the stack) and push the result, **MC**, onto the stack. Each time our traversal rises up through a transformation node, we pop a matrix from the stack. The result is that whenever we encounter geometry (like the coordinates of the hand points, or of the ellipse points), we can multiply the coordinate array on the left by the current transformation to get the WPF coordinates of those points. In the pseudocode below, we assume that the scene graph is represented by a `Scene` class with a method that returns the root node of the graph, and that a transformation node has a `matrix` method that returns the matrix for the associated transformation, while a geometry node has a `vertexCoordinateArray` method that returns a $3 \times k$ array containing the homogeneous coordinates of the k points in the polygon.

```
1  void drawScene(Scene myScene)
2     s = empty Stack
3     s.push( 3×3 identity matrix )
4     explore(myScene.rootNode(), s)
5
6
7  void explore(Node n, Stack& s)
8     if n is a transformation node
9       push n.matrix() * s.top() onto s
10
11    else if n is a geometry node
12      drawPolygon(s.top() * n.vertexCoordinateArray())
13
14    foreach child k of n
15      explore(k, s)
16
17    if n is a transformation node
18      pop top element from s
```

In some complex models, the cost of matrix multiplications can be enormous. If the same model is to be rendered over and over, and none of the transformations change (e.g., a model of a building in a driving-simulation game), it's often worth it to use the algorithm above to create a list of polygons in world coordinates that can be redrawn for each frame, rather than reparsing the scene once per frame. This is sometimes referred to as **prebaking** or **baking** a model.

The algorithm above is the core of the standard one used for scene traversals in scene graphs. There are two important additions, however.

First, geometric transformations are not the only things stored in a scene graph—in some cases, attributes like color may be stored as well. In a simple

version, each geometry node has a color, and the `drawPolygon` procedure is passed both the vertex coordinate array and the color. In a more complex version, the color attribute may be set at some node in the graph, and that color is used for all the geometry "beneath" that node. In this latter form, we can keep track of the color with a parallel stack onto which colors are pushed as they're encountered, just as transformations are pushed onto the transformation stack. The difference is that while transformations are multiplied by the previous composite transformation before being pushed on the stack, the colors, representing an absolute rather than a relative attribute, are pushed without being combined in any way with previous color settings. It's easy to imagine a scene graph in which color-alteration nodes are allowed (e.g., "Lighten everything below this node by 20%"); in such a structure, the stack would have to accumulate color transformations. Unless the transformations are quite limited, there's no obvious way to combine them except to treat them as a sequence of transformations; matrix transformations are rather special in this regard.

Second, we've studied an example in which the scene graph is a *tree,* but depth-first traversal actually makes sense in an arbitrary directed acyclic graph (DAG). And in fact, our clock model, in reality, *is* a DAG: The geometry for the two clock hands is shared by the hands (using a WPF `StaticResource`). During the depth-first traversal we arrive at the hand geometry twice, and thus render two different hands. For a more complex model (e.g., a scene full of identical robots) such repeated encounters with the same geometry may be very frequent: Each robot has two identical arms that refer to the same underlying arm model; each arm has three identical fingers that refer to the same underlying finger model, etc. It's clear that in such a situation, there's some lost effort in retraversal of the arm model. Doing some analysis of a scene graph to detect such retraversals and avoid them by prebaking can be a useful optimization, although in many of today's graphics applications, scene traversal is only a tiny fraction of the cost, and lighting and shading computations (for 3D models) dominate. You should avoid optimizing the scene-traversal portions of your code until you've verified that they are the expensive part.

10.11.1 Coordinate Changes in Scene Graphs

Returning to the scene graph and the matrix products, the transformations applied to the minute hand to get WPF coordinates,

$$\begin{bmatrix} 1 & 0 & 48 \\ 0 & 1 & 48 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4.8 & 0 & 0 \\ 0 & 4.8 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{10.89}$$

represent the transformation from minute-hand coordinates to WPF coordinates. To go from WPF coordinates to minute-hand coordinates, we need only apply the inverse transformation. Remembering that $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$, this inverse transformation is

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1/4.8 & 0 & 0 \\ 0 & 1/4.8 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -48 \\ 0 & 1 & -48 \\ 0 & 0 & 1 \end{bmatrix}. \tag{10.90}$$

You can similarly find the coordinate transformation matrix to get from any one coordinate system in a scene graph to any other. Reading upward, you accumulate

the matrices you encounter, with the first matrix being farthest to the right; reading downward, you accumulate their inverses in the opposite order. When we build scene graphs in 3D, exactly the same rules apply.

For a 3D scene, there's the description not only of the model, but also of how to transform points of the model into points on the display. This latter description is provided by specifying a camera. But even in 2D, there's something closely analogous: The `Canvas` in which we created our clock model corresponds to the "world" of a 3D scene; the way that we transform this world to make it appear on the display (scale by $(4.8, 4.8)$ and then translate by $(48, 48)$) corresponds to the viewing transformation performed by a 3D camera.

Typically the polygon coordinates (the ones we've placed in templates) are called modeling coordinates. Given the analogy to 3D, we can call the canvas coordinates world coordinates, while the WPF coordinates can be called image coordinates. These terms are all in common use when discussing 3D scene graphs.

As an exercise, let's consider the tip of the hour hand; in modeling coordinates (i.e., in the clock-hand template) the tip is located at $(0, 9)$. In the same way, the tip of the minute hand, in modeling coordinates, is at $(0, 9)$. What are the `Canvas` coordinates of the tip of the hour hand? We must multiply (reading from leaf toward root) by all the transformation matrices from the hour-hand template up to the `Canvas`, resulting in

$$\begin{bmatrix} 0.96 & -0.26 & 0 \\ 0.26 & 0.96 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1.7 & 0 & 0 \\ 0 & 0.7 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 9 \\ 1 \end{bmatrix} \tag{10.91}$$

$$= \begin{bmatrix} -1.64 & -.18 & 0 \\ -0.44 & -0.68 & 0 \\ 001 & & \end{bmatrix} \begin{bmatrix} 0 \\ 9 \\ 1 \end{bmatrix} = \begin{bmatrix} 1.63 \\ -6.09 \\ 1 \end{bmatrix}, \tag{10.92}$$

where all coordinates have been rounded to two decimal places for clarity. The `Canvas` coordinates of the tip of the minute hand are

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 9 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 9 \\ 1 \end{bmatrix}. \tag{10.93}$$

We can thus compute a vector from the hour hand's tip to the minute hand's tip by subtracting these two, getting $\begin{bmatrix} -1.63 & 15.08 & 0 \end{bmatrix}^{\mathbf{T}}$. The result is the homogeneous-coordinate representation of the vector $\begin{bmatrix} -1.63 & 15.08 \end{bmatrix}^{\mathbf{T}}$ in `Canvas` coordinates.

Suppose that we wanted to know the direction from the tip of the minute hand to the tip of the hour hand *in minute-hand coordinates.* If we knew this direction, we could add, within the minute-hand part of the model, a small arrow that pointed toward the hour-hand. To find this direction vector, we need to know the coordinates of the tip of the hour hand in minute-hand coordinates. So we must go from hour-hand coordinates to minute-hand coordinates, which we can do by working up the tree from the hour hand to the `Canvas`, and then back down to the minute hand. The location of the hour-hand tip, in minute-hand coordinates, is given by

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0.96 & -0.26 & 0 \\ 0.26 & 0.96 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1.7 & 0 & 0 \\ 0 & 0.7 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 9 \\ 1 \end{bmatrix}. \tag{10.94}$$

We subtract from this the coordinates, $(0, 9)$, of the tip of the minute hand (in the minute-hand coordinate system) to get a vector from the tip of the minute hand to the tip of the hour hand.

As a final exercise, suppose we wanted to create an animation of the clock in which someone has grabbed the minute hand and held it so that the rest of the clock spins around the minute hand. How could we do this?

Well, the reason the minute hand moves from its initial 12:00 position on the `Canvas` (i.e., its position *after* it has been rotated 180° the first time) is that a sequence of further transformations have been applied to it. This sequence is rather short: It's just the varying rotation. If we apply the *inverse* of this varying rotation to each of the clock elements, we'll get the desired result. Because we apply both the rotation and its inverse to the minute hand, we could delete both, but the structure is more readable if we retain them. We could also apply the inverse rotation as part of the `Canvas`'s render transform.

Inline Exercise 10.23: If we want to implement the second approach—inserting the inverse rotation in the `Canvas`'s render transform—should it appear (in the WPF code) before or after the scale-and-translate transforms that are already there? Try it!

10.12 Transforming Vectors and Covectors

We've agreed to say that the point $(x, y) \in E^2$ corresponds to the 3-space vector $\begin{bmatrix} x & y & 1 \end{bmatrix}^{\mathbf{T}}$, and that the vector $\begin{bmatrix} u \\ v \end{bmatrix}$ corresponds to the 3-space vector $\begin{bmatrix} u & v & 0 \end{bmatrix}^{\mathbf{T}}$. If we use a 3×3 matrix $\mathbf{M}$ (with last row $\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$) to transform 3-space via

$$T : \mathbf{R^3} \to \mathbf{R^3} : \mathbf{x} \mapsto \mathbf{Mx}, \tag{10.95}$$

then the restriction of T to the $w = 1$ plane has its image in E^2 as well, so we can write

$$(T|E^2) : E^2 \to E^2 : \mathbf{x} \mapsto \mathbf{Mx}. \tag{10.96}$$

But we also noted above that we could regard T as transforming **vectors,** or displacements of two-dimensional Euclidean space, which are typically written with two coordinates but which we represent in the form $\begin{bmatrix} u & v & 0 \end{bmatrix}^{\mathbf{T}}$. Because the last entry of such a "vector" is always 0, the last column of $\mathbf{M}$ has no effect on how vectors are transformed. Instead of computing

$$\mathbf{M}\begin{bmatrix} u \\ v \\ 0 \end{bmatrix}, \tag{10.97}$$

we could equally well compute

$$\begin{bmatrix} m_{1,1} & m_{1,2} & 0 \\ m_{2,1} & m_{2,2} & 0 \\ 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} u \\ v \\ 0 \end{bmatrix}, \tag{10.98}$$

and the result would have a 0 in the third place. In fact, we could transform such vectors directly as two-coordinate vectors, by simply computing

$$\begin{bmatrix} m_{1,1} & m_{1,2} \\ m_{2,1} & m_{2,2} \end{bmatrix}\begin{bmatrix} u \\ v \end{bmatrix}. \tag{10.99}$$

For this reason, it's sometimes said for an affine transformation of the Euclidean plane represented by multiplication by a matrix $\mathbf{M}$ that the associated transformation of vectors is represented by

$$\overline{\mathbf{M}} = \begin{bmatrix} m_{1,1} & m_{1,2} \\ m_{2,1} & m_{2,2} \end{bmatrix}. \tag{10.100}$$

What about covectors? Recall that a typical covector could be written in the form

$$\phi_{\mathbf{w}} : \mathbf{R}^2 \to \mathbf{R}^2 : \mathbf{v} \mapsto \mathbf{w} \cdot \mathbf{v}, \tag{10.101}$$

where $\mathbf{w}$ was some vector in $\mathbf{R}^2$. We'd like to transform $\phi_{\mathbf{w}}$ in a way that's consistent with T. Figure 10.19 shows why: We often build a geometric model of some shape and compute all the normal vectors to the shape. Suppose that $\mathbf{n}$ is one such surface normal. We then place that shape into 3-space by applying some "modeling transformation" $T_{\mathbf{M}}$ to it, and we'd like to know the normal vectors to that transformed shape so that we can do things like compute the angle between a light-ray $\mathbf{v}$ and that surface normal. If we call the transformed surface normal $\mathbf{m}$,

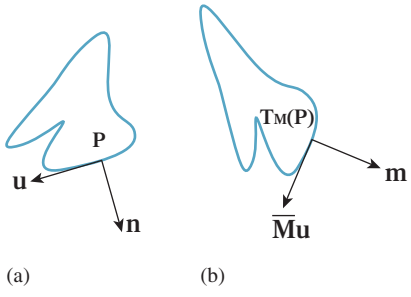


Figure 10.19: (a) A geometric shape that's been modeled using some modeling tool; the normal vector $\mathbf{n}$ *at a particular point P has been computed too. The vector* $\mathbf{u}$ *is tangent to the shape at P. (b) The shape has been translated, rotated, and scaled as it was placed into a scene. At the transformed location of P, we want to find the normal vector* $\mathbf{m}$ *with the property that its inner product with the transformed tangent* $\overline{\mathbf{M}}\mathbf{u}$ *is still 0.*

we want to compute $\mathbf{v} \cdot \mathbf{m}$. How is $\mathbf{m}$ related to the surface normal $\mathbf{n}$ of the original model?

The original surface normal $\mathbf{n}$ was defined by the property that it was orthogonal to every vector $\mathbf{u}$ that was tangent to the surface. The new normal vector $\mathbf{m}$ must be orthogonal to all the transformed tangent vectors, which are tangent to the transformed surface. In other words, we need to have

$$\mathbf{m} \cdot \overline{\mathbf{M}}\mathbf{u} = 0 \tag{10.102}$$

for every tangent vector $\mathbf{u}$ to the surface. In fact, we can go further. For *any* vector $\mathbf{u}$, we'd like to have

$$\mathbf{m} \cdot \overline{\mathbf{M}}\mathbf{u} = \mathbf{n} \cdot \mathbf{u}, \tag{10.103}$$

that is, we'd like to be sure that the angle between an untransformed vector and $\mathbf{n}$ is the same as the angle between a transformed vector and $\mathbf{m}$.

Before working through this, let's look at a couple of examples. In the case of the transformation T_1, the vector perpendicular to the bottom side of the house (we'll use this as our vector $\mathbf{n}$) should be transformed so that it's still perpendicular to the bottom of the transformed house. This is achieved by rotating it by 30° (see Figure 10.20).

If we just *translate* the house, then $\mathbf{n}$ again should be transformed just the way we transform ordinary vectors, that is, not at all.

But what about when we shear the house, as with example transformation T_3? The associated vector transformation is still a shearing transformation; it takes a vertical vector and tilts it! But the vector $\mathbf{n}$, if it's to remain perpendicular to the bottom of the house, must not be changed at all (see Figure 10.21). So, in this case, we see the necessity of transforming covectors differently from vectors.

Let's write down, once again, what we want. We're looking for a vector $\mathbf{m}$ that satisfies

$$\mathbf{m} \cdot (\overline{\mathbf{M}}\mathbf{u}) = \mathbf{n} \cdot \mathbf{u} \tag{10.104}$$

for every possible vector $\mathbf{v}$. To make the algebra more obvious, let's swap the order of the vectors and say that we want

$$(\overline{\mathbf{M}}\mathbf{u}) \cdot \mathbf{m} = \mathbf{u} \cdot \mathbf{n}. \tag{10.105}$$

Recalling that $\mathbf{a} \cdot \mathbf{b}$ can be written $\mathbf{a}^\mathbf{T}\mathbf{b}$, we can rewrite this as

$$(\overline{\mathbf{M}}\mathbf{u})^\mathbf{T}\mathbf{m} = \mathbf{u}^\mathbf{T}\mathbf{n}. \tag{10.106}$$

Remembering that $(\mathbf{AB})^\mathbf{T} = \mathbf{A}^\mathbf{T}\mathbf{B}^\mathbf{T}$, and then simplifying, we get

$$(\overline{\mathbf{M}}\mathbf{u})^\mathbf{T}\mathbf{m} = \mathbf{u}^\mathbf{T}\mathbf{n} \tag{10.107}$$

$$(\mathbf{u}^\mathbf{T}\overline{\mathbf{M}}^\mathbf{T})\mathbf{m} = \mathbf{u}^\mathbf{T}\mathbf{n} \tag{10.108}$$

$$\mathbf{u}^\mathbf{T}(\overline{\mathbf{M}}^\mathbf{T}\mathbf{m}) = \mathbf{u}^\mathbf{T}\mathbf{n}, \tag{10.109}$$

where the last step follows from the associativity of matrix multiplication. This last equality is of the form $\mathbf{u} \cdot \mathbf{a} = \mathbf{u} \cdot \mathbf{b}$ for all $\mathbf{u}$. Such an equality holds if and only if $\mathbf{a} = \mathbf{b}$, that is, if and only if

$$\overline{\mathbf{M}}^\mathbf{T}\mathbf{m} = \mathbf{n}, \tag{10.110}$$

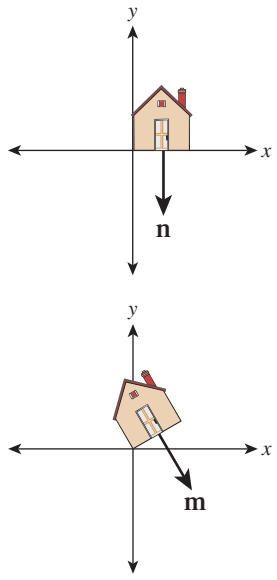


Figure 10.20: For a rotation, the normal vector rotates the same way as all other vectors.

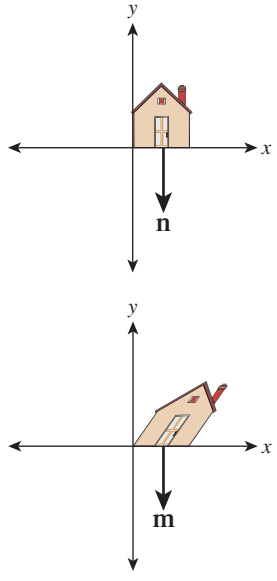


Figure 10.21: While the vertical sides of the house are sheared, the normal vector to the house's bottom remains unchanged.

so

$$\mathbf{m} = (\overline{\mathbf{M}}^{\mathbf{T}})^{-1}\mathbf{n}, \tag{10.111}$$

where we are assuming that $\overline{\mathbf{M}}$ is invertible.

We can therefore conclude that the covector $\phi_{\mathbf{n}}$ transforms to the covector $\phi_{(\overline{\mathbf{M}}^{\mathbf{T}})^{-1}\mathbf{n}}$. For this reason, the inverse transpose is sometimes referred to as the **covector transformation** or (because of its frequent application to normal vectors) the **normal transform.** Note that if we choose to write covectors as row vectors, then the transpose is not needed, but we have to multiply the row vector on the *right* by $\overline{\mathbf{M}}^{-1}$.

The normal transform, in its natural mathematical setting, goes in the opposite direction: It takes a normal vector in the codomain of $T_{\mathbf{M}}$ and produces one in the domain; the matrix for this **adjoint transformation** is $\mathbf{M}^{\mathbf{T}}$. Because we need to use it in the other direction, we end up with an inverse as well.

Taking our shearing transformation, T_3, as an example, when written in *xyw*-space the matrix $\mathbf{M}$ for the transformation is

$$\begin{bmatrix} 1 & 2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{10.112}$$

and hence $\overline{\mathbf{M}}$ is

$$\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}, \tag{10.113}$$

while the normal transform is

$$(\overline{\mathbf{M}}^{-1})^{\mathbf{T}} = \begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}^{\mathbf{T}} = \begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix}. \tag{10.114}$$

Hence the covector $\phi_{\mathbf{n}}$, where $\mathbf{n} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$, for example, becomes the covector $\phi_{\mathbf{m}}$, where $\mathbf{m} = \begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix}\mathbf{n} = \begin{bmatrix} 2 \\ -3 \end{bmatrix}$.

Inline Exercise 10.24: (a) Find an equation (in coordinates, not vector form) for a line passing through the point $P = (1, 1)$, with normal vector $\mathbf{n} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$. (b) Find a second point Q on this line. (c) Find $P' = T_3(P)$ and $Q' = T_3(Q)$, and a coordinate equation of the line joining P' and Q'. (d) Verify that the normal to this second line is in fact proportional to $\mathbf{m} = \begin{bmatrix} 2 \\ -3 \end{bmatrix}$, confirming that the normal transform really did properly transform the normal vector to this line.

Inline Exercise 10.25: We assumed that the matrix $\mathbf{M}$ was invertible when we computed the normal transform. Give an intuitive explanation of why, if $\mathbf{M}$ is degenerate (i.e., not invertible), it's impossible to define a normal transform. Hint: Suppose that $\mathbf{u}$, in the discussion above, is sent to $\mathbf{0}$ by $\mathbf{M}$, but that $\mathbf{u} \cdot \mathbf{n}$ is nonzero.

10.12.1 Transforming Parametric Lines

All the transformations of the $w = 1$ plane we've looked at share the property that they send lines into lines. But more than that is true: They send *parametric* lines to *parametric* lines, by which we mean that if ℓ is the parametric line $\ell = \{P + t\mathbf{v} : t \in \mathbf{R}\}$, and $Q = P + \mathbf{v}$ (i.e., ℓ starts at P and reaches Q at $t = 1$), and T is the transformation $T(\mathbf{v}) = \mathbf{M}\mathbf{v}$, then $T(\ell)$ is the line

$$T(\ell) = \{T(P) + t(T(Q) - T(P)) : t \in \mathbf{R}\}, \tag{10.115}$$

and in fact, the point at parameter t in ℓ (namely $P + t(Q - P)$) is sent by T to the point at parameter t in $T(\ell)$ (namely $T(P) + t(T(Q) - T(P))$).

This means that for the transformations we've considered so far, transforming the plane commutes with forming affine or linear combinations, so you can either transform and then average a set of points, or average and then transform, for instance.

10.13 More General Transformations

Let's look at one final transform, T, which is a prototype for transforms we'll use when we study projections and cameras in 3D. All the essential ideas occur in 2D, so we'll look at this transformation carefully. The matrix $\mathbf{M}$ for the transformation T is

$$\mathbf{M} = \begin{bmatrix} 2 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}. \tag{10.116}$$

It's easy to see that $T_{\mathbf{M}}$ doesn't transform the $w = 1$ plane into the $w = 1$ plane.

Inline Exercise 10.26: Compute $T(\begin{bmatrix} 2 & 0 & 1 \end{bmatrix}^{\mathbf{T}})$ and verify that the result is not in the $w = 1$ plane.

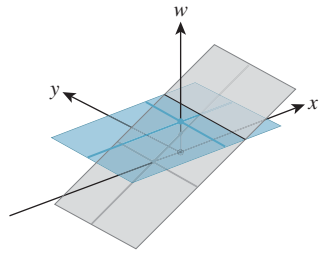


Figure 10.22: The blue $w = 1$ plane transforms into the tilted gray plane under $T_{\mathbf{M}}$.

Figure 10.22 shows the $w = 1$ plane in blue and the transformed $w = 1$ plane in gray. To make the transformation T useful to us in our study of the $w = 1$ plane, we need to take the points of the gray plane and "return" them to the blue plane somehow. To do so, we introduce a new function, H, defined by

$$H : \mathbf{R}^3 - \left\{ \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} : x, y, \in \mathbf{R} \right\} \to \mathbf{R}^3 : \begin{bmatrix} x \\ y \\ w \end{bmatrix} \mapsto \begin{bmatrix} x/w, y/w, 1 \end{bmatrix}. \tag{10.117}$$

Figure 10.23 show how the analogous function in two dimensions sends every point except those on the $w = 0$ line to the line $w = 1$: For a typical point P, we connect P to the origin O with a line and see where this line meets the $w = 1$ plane. Notice that even a point in the negative-w half-space on the same line gets sent to the same location on the $w = 1$ line. This connect-and-intersect operation isn't defined, of course, for points on the x-axis, because the line connecting them to the origin is the axis itself, which never meets the $w = 1$ line. H is often called **homogenization** in graphics.

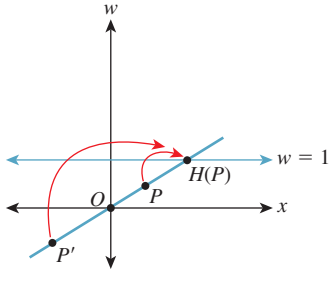


Figure 10.23: Homogenization $\begin{bmatrix} x \\ w \end{bmatrix} \mapsto \begin{bmatrix} x/w \\ 1 \end{bmatrix}$ in two dimensions.

With H in hand, we'll define a new transformation on the $w = 1$ plane by

$$S(\mathbf{v}) = H(T_{\mathbf{M}}(\mathbf{v})). \tag{10.118}$$

This definition has a serious problem: As you can see from Figure 10.22, some points in the image of T are in the $w = 0$ plane, on which H is not defined so that S cannot be defined there. For now, we'll ignore this and simply not apply S to any such points.

Inline Exercise 10.27: Find all points $\mathbf{v} = \begin{bmatrix} x & y & 1 \end{bmatrix}^{\mathbf{T}}$ of the $w = 1$ plane such that the w-coordinate of $T_{\mathbf{M}}(\mathbf{v})$ is 0. These are the points on which S is undefined.

The transformation S, defined by multiplication by the matrix $\mathbf{M}$, followed by homogenization, is called a **projective transformation.** Notice that if we followed either a linear or affine transformation with homogenization, the homogenization would have no effect. Thus, we have three nested classes of transformations: linear, affine (which includes linear *and* translation and combinations of them), and projective (which includes affine *and* transformations like S).

Figure 10.24 shows several objects in the $w = 1$ plane, drawn as seen looking down the w-axis, with the y-axis, on which S is undefined, shown in pale green. Figure 10.25 shows these objects after S has been applied to them. Evidently, S takes lines to lines, mostly: A line segment like the blue one in the figure that meets the y-axis in the segment's interior turns into two rays, but the two rays both lie in the same line. We say that the line $y = 0$ has been "sent to infinity." The red vertical line at $x = 1$ in Figure 10.24 transforms into the red vertical line at $x = 0$ in Figure 10.25. And every ray through the origin in Figure 10.24 turns into a horizontal line in Figure 10.25. We can say even more: Suppose that P_1 denotes radial projection onto the $x = 1$ line in Figure 10.24, while P_2 denotes horizontal projection onto the $z = 0$ line in Figure 10.25. Then

$$S(P_1(X)) = P_2(S(X)) \tag{10.119}$$

for any point X that's not on the y-axis. In other words, S converts radial projection into parallel projection. In Chapter 13 we'll see exactly the same trick in 3-space: We'll convert radial projection toward the eye into parallel projection. This is useful because in parallel projection, it's *really* easy to tell when one object obscures another by just comparing "depth" values!

Let's look at how S transforms a *parameterized* line. Consider the line ℓ starting at a point P and passing through a point Q when $t = 1$,

$$\ell(t) = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} + t \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} \tag{10.120}$$

$$= P + t(Q - P), \tag{10.121}$$

where $P = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}^{\mathbf{T}}$ and $Q = \begin{bmatrix} 3 & 1 & 1 \end{bmatrix}^{\mathbf{T}}$ so that in the $w = 1$ plane, the line starts at $(x, y) = (1, 0)$ when $t = 0$ and goes to the right and slightly upward, arriving at $(x, y) = (3, 1)$ when $t = 1$ (see Figure 10.26).

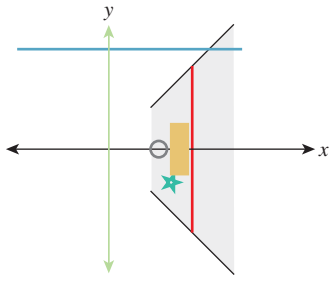


Figure 10.24: Objects in the $w = 1$ plane before transformation.

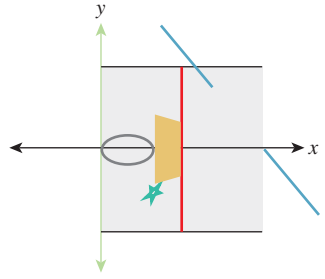


Figure 10.25: The same objects after transformation by S.

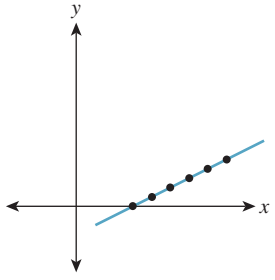


Figure 10.26: The line ℓ passes through P at $t = 0$ and Q at $t = 1$; the black points are equispaced in the interval $0 \leq t \leq 1$.

The function T transforms this to the line ℓ' that starts at $T(P) = \begin{bmatrix}1 & 0 & 1\end{bmatrix}^{\mathbf{T}}$ when $t = 0$ and arrives at $T(Q) = \begin{bmatrix}5 & 1 & 3\end{bmatrix}^{\mathbf{T}}$ when $t = 1$, and whose equation is

$$\ell'(t) = \begin{bmatrix}1 & 0 & 1\end{bmatrix} + t\begin{bmatrix}4 & 1 & 2\end{bmatrix} \tag{10.122}$$

$$= T(P) + t(T(Q) - T(P)). \tag{10.123}$$

Figure 10.27 shows the line in 3-space, after transformation by $T_{\mathbf{M}}$; the point spacing remains constant.

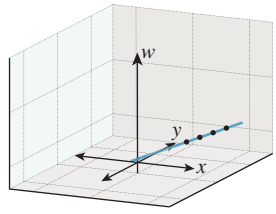


Figure 10.27: After transformation by $T_{\mathbf{M}}$, the points are still equispaced.

We know that this is the parametric equation of the line, because *every* linear transformation transforms parametric lines to parametric lines. But when we apply H, something interesting happens. Because the function H is *not linear,* the parametric line is *not* transformed to a parametric line. The point $\ell'(t) = \begin{bmatrix}1+4t & t & 1+2t\end{bmatrix}^{\mathbf{T}}$ is sent to

$$m(t) = \begin{bmatrix}(1+4t)/(1+2t) \\ t/(1+2t) \\ 1\end{bmatrix} \tag{10.124}$$

$$= \begin{bmatrix}1\\0\\1\end{bmatrix} + \frac{t}{1+2t}\begin{bmatrix}2\\1\\0\end{bmatrix} \tag{10.125}$$

Equation 10.125 has *almost* the form of a parametric line, but the coefficient of the direction vector, which is proportional to $S(Q) - S(P)$, has the form

$$\frac{at+b}{ct+d}, \tag{10.126}$$

which is called a **fractional linear transformation** of t. This nonstandard form is of serious importance in practice: It tells us that if we interpolate a value at the midpoint M of P and Q, for instance, from the values at P and Q, and then transform all three points by S, then $S(M)$ will generally *not* be at the midpoint of $S(P)$ and $S(Q)$, so the interpolated value will not be the correct one to use if we need post-transformation interpolation. Figure 10.28 shows how the equally spaced points in the domain have become unevenly spaced after the projective transformation.

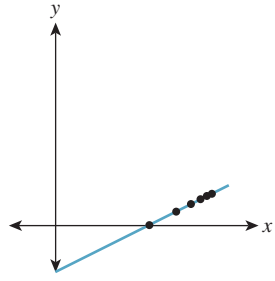


Figure 10.28: After homogenization, the points are no longer equispaced.

In other words, transformation by S and interpolation are not commuting operations. When we apply a transformation that includes the homogenization operation H, we cannot assume that interpolation will give the same results pre- and post-transformation. Fortunately, there's a solution to this problem (see Section 15.6.4).

Inline Exercise 10.28: (a) Show that if n and f are distinct nonzero numbers, the transformation defined by the matrix

$$\mathbf{N} = \begin{bmatrix}\frac{f}{f-n} & 0 & \frac{fn}{n-f} \\ 0 & 1 & 0 \\ 1 & 0 & 0\end{bmatrix}, \tag{10.127}$$

when followed by homogenization, sends the line $x = 0$ to infinity, the line $x = n$ to $x = 0$, and the line $x = f$ to $x = 1$.
(b) Figure out how to modify the matrix to send $x = f$ to $x = -1$ instead.

Inline Exercise 10.29: (a) Show that if T is any linear transformation on $\mathbf{R}^3$, then for any nonzero $\alpha \in \mathbf{R}$ and any vector $\mathbf{v} \in \mathbf{R}^3$, $H(T(\alpha\mathbf{v})) = H(T(\mathbf{v}))$.
(b) Show that if $\mathbf{K}$ is any matrix, then $H(T_{\mathbf{K}}(\mathbf{v})) = H(T_{\alpha\mathbf{K}}(\mathbf{v}))$ as well.
(c) Conclude that in a sequence of matrix operations in which there's an H at the end, matrix scale doesn't matter, that is, you can multiply a matrix by any nonzero constant without changing the end result.

Suppose we have a matrix transformation on 3-space given by $T(\mathbf{v}) = \mathbf{K}\mathbf{v}$, and T is nondegenerate (i.e., $T(\mathbf{v}) = \mathbf{0}$ only when $\mathbf{v} = \mathbf{0}$). Then T takes lines through the origin to lines through the origin, because if $\mathbf{v} \neq \mathbf{0}$ is any nonzero vector, then $\{\alpha\mathbf{v} : \alpha \in \mathbf{R}\}$ is the line through the origin containing $\mathbf{v}$, and when we transform this, we get $\{\alpha T(\mathbf{v}) : \alpha \in \mathbf{R}\}$, which is the line through the origin containing $T(\mathbf{v})$. Thus, rather than thinking of the transformation T as moving around points in $\mathbf{R}^3$, we can think of it as acting on the set of lines through the origin. By intersecting each line through the origin with the $w = 1$ plane, we can regard T as acting on the $w = 1$ plane, but with a slight problem: A line through the origin in 3-space that meets the $w = 1$ plane may be transformed to one that does not (i.e., a horizontal line), and vice versa. So using the $w = 1$ plane to "understand" the lines-to-lines version of the transformation T is a little confusing.

The idea of considering linear transformations as transformations on the set of lines through the origin is central to the field of **projective geometry.** An understanding of projective geometry can lead to a deeper understanding of the transformations we use in graphics, but is by no means essential. Hartshorne [Har09] provides an excellent introduction for the student who has studied abstract algebra.

Transformations of the $w = 1$ plane like the ones we've been looking at in this section, consisting of an arbitrary matrix transformation on $\mathbf{R}^3$ followed by H, are called **projective transformations.** The class of projective transformations includes all the more basic operations like translation, rotation, and scaling of the plane (i.e., *affine* transformations of the plane), but include many others as well. Just as with linear and affine transformations, there's a uniqueness theorem: If P, Q, R, and S are four points of the plane, no three collinear, then there's exactly one projective transformation sending these points to $(0,0)$, $(1,0)$, $(0,1)$, and $(1,1)$, respectively. (Note that this one transformation might be described by two *different* matrices. For example, if $\mathbf{K}$ is the matrix of a projective transformation S, then $2\mathbf{K}$ defines exactly the same transformation.)

For all the affine transformations we discussed in earlier sections, we've determined an associated transformation of vectors and of normal vectors. For projective transformations, this process is messier. Under the projective transformation shown in Figures 10.24 and 10.25, we can consider the top and bottom edges of the tan rectangle as vectors that point in the same direction. After the transformation, you can see that they have been transformed to point in *different* directions. There's no single "vector" transformation to apply. If we have a vector $\mathbf{v}$ starting at the point P, we have to apply "the vector transformation at P" to $\mathbf{v}$ to find out where it will go. The same idea applies to normal vectors: There's a different

normal transformation at every point. In both cases, it's the function H that leads to problems. The "vector" transformation for any function U is, in general, the derivative DU. In the case of a matrix transformation $T_{\mathbf{M}}$ that's being applied only to points of the $w = 1$ plane, the "vectors" lying in that plane all have $w = 0$, and so the matrix used to transform these vectors can have its third column set to be all zeroes (or can be just written as a 2×2 matrix operating on vectors with two entries), as we have seen earlier. But since

$$S = H \circ T_{\mathbf{M}}, \tag{10.128}$$

we have (using the multivariable chain rule)

$$DS(P) = DH(T_{\mathbf{M}}(P)) \cdot DT_{\mathbf{M}}(P). \tag{10.129}$$

Now, since $H(\begin{bmatrix} x \\ y \\ w \end{bmatrix}) = \begin{bmatrix} x/w \\ y/w \\ 1 \end{bmatrix}$, we know that

$$DH(\begin{bmatrix} x \\ y \\ w \end{bmatrix}) = \begin{bmatrix} 1/w & 0 & -x/w^2 \\ 0 & 1/w & -y/w^2 \\ 0 & 0 & 0 \end{bmatrix} \tag{10.130}$$

$$= \frac{1}{w^2} \begin{bmatrix} w & 0 & -x \\ 0 & w & -y \\ 0 & 0 & 0 \end{bmatrix} \tag{10.131}$$

and

$$DT_{\mathbf{M}}(P) = \mathbf{M} = \begin{bmatrix} 2 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}. \tag{10.132}$$

So, if $P = \begin{bmatrix} x, y, 1 \end{bmatrix}$ is a point of the $w = 1$ plane and $\mathbf{v} = \begin{bmatrix} s \\ t \\ 0 \end{bmatrix}$ is a vector in that plane, then $S(P) = \begin{bmatrix} 2x - 1 \\ y \\ x \end{bmatrix}$ and

$$DS(P)(\mathbf{v}) = DH(\begin{bmatrix} 2x - 1 \\ y \\ x \end{bmatrix}) \cdot DT(P)\mathbf{v} = \begin{bmatrix} s \\ t \\ 0 \end{bmatrix} \tag{10.133}$$

$$= \frac{1}{x^2} \begin{bmatrix} x & 0 & -(2x+1) \\ 0 & x & -y \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} s \\ t \\ 0 \end{bmatrix} \tag{10.134}$$

$$= \frac{1}{x^2} \begin{bmatrix} -1 & 0 & -x \\ -y & x & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} s \\ t \\ 0 \end{bmatrix} = \begin{bmatrix} -s/x^2 \\ (tx - sy)/x^2 \\ 0 \end{bmatrix}. \tag{10.135}$$

Evidently, the "vector" transformation depends on the point $(x, y, 1)$ at which it's applied. The normal transform, being the inverse transpose of the vector transform, has the same dependence on the point of application.

10.14 Transformations versus Interpolation

When you rotate a book on your desk by 30° counterclockwise, the book is rotated by each intermediate amount between zero and 30°. But when we "rotate" the house in our diagram by 30°, we simply compute the final position of each point of the house. In no sense has it passed through any intermediate positions. In the more extreme case of rotation by 180°, the resultant transformation is exactly the same as the "uniform scale by -1" transformation. And in the case of rotation by 360°, the resultant transformation is the identity.

This reflects a *limitation in modeling.* The use of matrix transformations to model transformations of ordinary objects captures only the relationship between initial and final positions, and not the means by which the object got from the initial to the final position.

Much of the time, this distinction is unimportant: We want to put an object into a particular pose, so we apply some sequence of transformations to it (or its parts). But sometimes it can be quite significant: We might instead want to show the object being transformed from its initial state to its final state. An easy, but rarely useful, approach is to linearly interpolate each point of the object from its initial to its final position. If we do this for the "rotation by 180°" example, at the halfway point the entire object is collapsed to a single point; if we do it from the "rotation by 360°" example, the object never appears to move at all! The problem is that what we *really* want is to find interpolated versions of our *descriptions* of the transformation rather than of the transformations themselves. (Thus, to go from the initial state to "rotated by 360" we'd apply "rotate by s" to the initial state, for each value of s from 0 to 360.)

But sometimes students confuse a transformation like "multiplication by the identity matrix" with the way it was specified, "rotate by 360°," and they can be frustrated with the impossibility of "dividing by two" to get a rotation by 180°, for instance. This is particularly annoying when one has access only to the matrix form of the transformation, rather than the initial specification; in that case, as the examples show, there's no hope for a general solution to the problem of "doing a transformation partway." On the other hand, there *is* a solution that often gives reasonable results in practice, especially for the problem of interpolating two rather similar transformations (e.g., interpolating between rotating by 20° and rotating by 30°), which often arises. We'll discuss this in Chapter 11.

10.15 Discussion and Further Reading

We've introduced three classes of basic transformations: *linear,* which you've already encountered in linear algebra; *affine,* which includes translations and can be seen as a subset of the linear transformations in xyw-space, restricted to the $w = 1$ plane; and *projective,* which arises from general linear transformations on xyw-space, restricted to the $w = 1$ plane and then followed by the homogenization operation that divides through by w. We've shown how to represent each kind of transformation by matrix multiplication, but we urge you to separate the idea of a transformation from the matrix that represents it.

For each category, there's a theorem about uniqueness: A linear transformation on the plane is determined by its values on two independent vectors; an affine transformation is determined by its values on any three noncollinear points;

a projective transformation is determined by its values on any four points, no three of which are collinear. In the next chapter we'll see analogous results for 3-space, and in the following one we'll see how to use these theorems to build a library for representing transformations so that you don't have to spend a lot of time building individual matrices.

Even though matrices are not as easy for humans to interpret as "This transformation sends the points A, B, and C to A', B', and C'," the matrix representation of a transformation is very valuable, mostly because composition of transformation is equivalent to multiplication of matrices; performing a complex sequence of transformations on many points can be converted to multiplying the points' coordinates by a *single* matrix.

10.16 Exercises

Exercise 10.1: Use the 2D test bed to write a program to demonstrate windowing transforms. The user should click and drag two rectangles, and you should compute the transform between them. Subsequent clicks by the user within the first rectangle should be shown as small dots, and the corresponding locations in the second rectangle should also be shown as dots. Provide a Clear button to let the user restart.

Exercise 10.2: Multiply $\mathbf{M} = \begin{bmatrix} a & c \\ b & d \end{bmatrix}$ by the expression given in Equation 10.17 for its inverse to verify that the product really is the identity.

Exercise 10.3: Suppose that $\mathbf{M}$ is an $n \times n$ square matrix with SVD $\mathbf{M} = \mathbf{UDV^T}$.
(a) Why is $\mathbf{V^TV}$ the identity?
(b) Let i be any number from 1 to n. What is $\mathbf{V^T}\mathbf{v}_i$, where $\mathbf{v}_i$ denotes the ith column of $\mathbf{V}$? Hint: Use part (a).
(c) What's $\mathbf{DV^T}\mathbf{v}_i$?
(d) What's $\mathbf{M}\mathbf{v}_i$ in terms of $\mathbf{u}_i$ and d_i, the ith diagonal entry of D?
(e) Let $\mathbf{M}' = d_1\mathbf{u}_1\mathbf{v}_1^\mathbf{T} + \ldots + d_n\mathbf{u}_n\mathbf{v}_n^\mathbf{T}$. Show that $\mathbf{M}'\mathbf{v}_i = d_i\mathbf{u}_i$.
(f) Explain why $\mathbf{v}_i, i = 1, \ldots, n$ are linearly independent, and thus span $\mathbf{R}^n$.
(g) Conclude that $\mathbf{w} \mapsto \mathbf{Mw}$ and $\mathbf{w} \mapsto \mathbf{M}'\mathbf{w}$ agree on n linearly independent vectors, and hence must be the same linear transformation of $\mathbf{R}^n$.
(h) Conclude that $\mathbf{M}' = \mathbf{M}$. Thus, the singular-value decomposition proves the theorem that every matrix can be written as a sum of **outer products** (i.e., matrices of the form $\mathbf{vw^T}$).

Exercise 10.4: (a) If P, Q, and R are noncollinear points in the plane, show that $Q - P$ and $R - P$ are linearly independent vectors.
(b) If $\mathbf{v}_1$ and $\mathbf{v}_2$ are linearly independent points in the plane, and A is any point in the plane, show that $A, B = A + \mathbf{v}_1$ and $C = A + \mathbf{v}_2$ are noncollinear points. This shows that the two kinds of affine frames are equivalent.
(c) Two forms of an affine frame in 3-space are (i) four points, no three coplanar, and (ii) one point and three linearly independent vectors. Show how to convert from one to the other, and also describe a third possible version (Three points and one vector? Two points and two vectors? You choose!) and show its equivalence as well.

Exercise 10.5: We said that if the columns of the matrix $\mathbf{M}$ are $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k \in R^n$, and they are pairwise orthogonal unit vectors, then $\mathbf{M^TM} = \mathbf{I}_k$, the $k \times k$ identity matrix.

(a) Explain why, in this situation, $k \leq n$.
(b) Prove the claim that $\mathbf{M}^{\mathbf{T}}\mathbf{M} = \mathbf{I}_k$.

Exercise 10.6: An image (i.e., an array of grayscale values between 0 and 1, say) can be thought of as a large matrix, $\mathbf{M}$ (indeed, this is how we usually represent images in our programs). Use a linear algebra library to compute the SVD $\mathbf{M} = \mathbf{UDV}^{\mathbf{T}}$ of some image $\mathbf{M}$. According to the decomposition theorem described in Exercise 10.3, this describes the image as a sum of outer products of many vectors. If we replace the last 90% of the diagonal entries of $\mathbf{D}$ with zeroes to get a new matrix $\mathbf{D}'$, then the product $\mathbf{M}' = \mathbf{UD}'\mathbf{V}$ deletes 90% of the terms in this sum of outer products. In doing so, however, it deletes the *smallest* 90% of the terms. Display $\mathbf{M}'$ and compare it to $\mathbf{M}$. Experiment with values other than 90%. At what level do the two images become indistinguishable? You may encounter values less than 0 and greater than 1 during the process described in this exercise. You should simply clamp these values to the interval $[0, 1]$.

Exercise 10.7: The **rank** of a matrix is the number of linearly independent columns of the matrix.
(a) Explain why the outer product of two nonzero vectors always has rank one.
(b) The decomposition theorem described in Exercise 10.3 expresses a matrix $\mathbf{M}$ as a sum of rank one matrices. Explain why the sum of the first p such outer products has rank p (assuming $d_1, d_2, \ldots, d_p \neq 0$). In fact, this sum $\mathbf{M}_p$ is the rank p matrix that's closest to $\mathbf{M}$, in the sense that the sum of the squares of the entries of $\mathbf{M} - \mathbf{M}_p$ is as small as possible. (You need not prove this.)

Exercise 10.8: Suppose that $T : \mathbf{R^2} \to \mathbf{R^2}$ is a linear transformation represented by the 2×2 matrix $\mathbf{M}$, that is, $T(\mathbf{x}) = \mathbf{Mx}$. Let $K = \max_{\mathbf{x} \in \mathbf{S^1}} \|T(\mathbf{x})\|^2$, that is, K is the maximum squared length of all unit vectors transformed by $\mathbf{M}$.
(a) If the SVD of $\mathbf{M}$ is $\mathbf{M} = \mathbf{UDV}^{\mathbf{T}}$, show that $K = d_1^2$.
(b) What is the *minimum* squared length among all such vectors, in terms of $\mathbf{D}$?
(c) Generalize to $\mathbf{R^3}$.

Exercise 10.9: Show that three distinct points P, Q, and R in the Euclidean plane are collinear if and only if the corresponding vectors ($\mathbf{v}_P = \begin{bmatrix} P_x \\ P_y \\ 1 \end{bmatrix}$, etc.) are linearly dependent, by showing that if $\alpha_P \mathbf{v}_P + \alpha_Q \mathbf{v}_Q + \alpha_R \mathbf{v}_R = \mathbf{0}$ with not all the αs being 0, then
(a) *none* of the αs are 0, and
(b) the point Q is an affine combination of P and R; in particular, $Q = -\frac{\alpha_P}{\alpha_Q} P - \frac{\alpha_R}{\alpha_P} R$, so Q must lie on the line between P and R.
(c) Argue why dependence and collinearity are trivially the same if two or more of the points P, Q, and R are identical.

Exercise 10.10: It's good to be able to recognize the transformation represented by a matrix by looking at the matrix; for instance, it's easy to recognize a 3×3 matrix that represents a translation in homogeneous coordinates: Its last row is $\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$ and its upper-left 2×2 block is the identity. Given a 3×3 matrix representing a transformation in homogeneous coordinates,
(a) how can you tell whether the transformation is affine or not?
(b) How can you tell whether the transformation is linear or not?
(c) How can you tell whether it represents a rotation about the origin?
(d) How can you tell if it represents a uniform scale?

Exercise 10.11: Suppose we have a linear transformation $T : \mathbf{R^2} \to \mathbf{R^2}$, and two coordinate systems with bases $\{\mathbf{u}_1, \mathbf{u}_2\}$ and $\{\mathbf{v}_1, \mathbf{v}_2\}$; all four basis vectors are

unit vectors, $\mathbf{u}_2$ is 90° counterclockwise from $\mathbf{u}_1$, and similarly $\mathbf{v}_2$ is 90° counterclockwise from $\mathbf{v}_1$. You can write down the matrix $\mathbf{M}_u$ for T in the u-coordinate system and the matrix $\mathbf{M}_v$ for T in the v-coordinate system.

(a) If $\mathbf{M}_u$ is a rotation matrix $\begin{bmatrix} \cos\theta & \sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$, what can you say about $\mathbf{M}_v$?

(b) If $\mathbf{M}_u$ is a uniform scaling matrix, that is, a multiple of the identity, what can you say about $\mathbf{M}_v$?

(c) If $\mathbf{M}_u$ is a *non*uniform scaling matrix of the form $\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}$, with $a \neq b$, what can you say about $\mathbf{M}_v$?

Chapter 11

Transformations in Three Dimensions

11.1 Introduction

Transformations in 3-space are in many ways analogous to those in 2-space.

- Translations can be incorporated by treating three-dimensional space as the subset E^3 defined by $w = 1$ in the four-dimensional space of points (x, y, z, w). A linear transformation whose matrix has the form $\begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix}$, when restricted to E^3, acts as a translation by $\begin{bmatrix} a & b & c \end{bmatrix}^{\mathbf{T}}$ on E^3.
- If T is any continuous transformation that takes lines to lines, and $\mathbf{O}$ denotes the origin of 3-space, then we can define

$$\hat{T}(\mathbf{x}) = T(\mathbf{x}) - T(\mathbf{O}) \tag{11.1}$$

  and the result is a line-preserving transformation $\hat{T}$ that takes the origin to the origin. Such a transformation is represented by multiplication by a 3×3 matrix $\mathbf{M}$. Thus, to understand line-preserving transformations on 3-space, we can decompose each into a translation (possibly the identity) and a linear transformation of 3-space.
- Projective transformations are similar to those in 2-space; instead of being undefined on a line, they are undefined on a whole plane. Otherwise, they are completely analogous.
- Scale transformations can again be uniform or nonuniform; those that are nonuniform are characterized by three orthogonal invariant directions and three scale factors rather than just two, but nothing else is significantly different. The matrix for an axis-aligned scale by amounts a, b, and c along the x-, y-, and z-axes, respectively, is

$$\begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{11.2}$$

Scale transformations in which one or three of a, b, and c are negative reverse orientation: A triple of vectors $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$ that form a right-handed coordinate system will, after transformation by such a matrix, form a left-handed coordinate system. A *uniform* scale by a negative number has all three diagonal entries negative, and hence reverses orientation.

- Similarly, shearing transformations continue to leave a line fixed. Points not on this line are moved by an amount that depends on their position relative to the line, but this position is now measured in *two* dimensions instead of just one. There are also shears that leave a plane fixed.
- Reflections in 2D were either reflections *through a point* (the transformation $\mathbf{x} \mapsto -\mathbf{x}$), which turns out to be the same as rotation by an angle π, or reflections *through a line*. In 3D, there are reflections through a point, a line, or a plane. Reflection through a line corresponds to rotation about the line by π. Reflection through a point is still given by the map $\mathbf{x} \mapsto -\mathbf{x}$; in contrast to the two-dimensional case, this map is orientation-reversing. Finally, reflection through a plane is given by the map

$$\mathbf{x} \mapsto \mathbf{x} - 2(\mathbf{x} \cdot \mathbf{n})\mathbf{n}, \tag{11.3}$$

where $\mathbf{n}$ is the unit normal vector to the plane. This is algebraically analogous to reflection through a line in two dimensions, but in three dimensions it is *orientation-preserving*. The matrix for this map is

$$\mathbf{I} - 2\mathbf{n}\mathbf{n}^{\mathbf{T}} = \begin{bmatrix} 1 - 2n_x^2 & -2n_x n_y & -2n_x n_z & 0 \\ -2n_x n_y & 1 - 2n_y^2 & -2n_y n_z & 0 \\ -2n_x n_z & -2n_y n_z & 1 - 2n_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{11.4}$$

but it should come as no surprise at this point that we recommend that you use the expression $\mathbf{I} - 2\mathbf{n}\mathbf{n}^{\mathbf{T}}$ to create a reflection matrix rather than explicitly typing in the matrix entries, which is prone to error.

The most important difference between two and three dimensions arises when we consider *rotations*. In two dimensions, the set of rotations about the origin corresponds nicely with the unit circle: If R is a rotation, we look at $R(\mathbf{e}_1)$, which is a point on the unit circle. This gives a mapping from rotations to the circle; the inverse mapping is given by taking each point $[x, y]^{\mathbf{T}}$ on the unit circle and associating to it the rotation whose matrix is

$$\begin{bmatrix} x & -y \\ y & x \end{bmatrix}, \tag{11.5}$$

for which it's easy to verify that $\mathbf{e}_1$ is sent to $[x, y]^{\mathbf{T}}$. Thus, we can say that the set of rotations in two dimensions is a one-dimensional shape: Knowing a single

number (the angle of rotation) completely determines the rotation.[1] By contrast, we'll see in Section 11.2 that in 3-space, the set of rotations is *three*-dimensional. Furthermore, there's no nice one-to-one correspondence with a familiar object like a circle.

Although you should, in general, use code like that described in the next chapter to perform transformations, during debugging you'll often find yourself looking at matrices. It's a good skill to be able to recognize translations and scales instantly, and to be able to guess quickly that the upper-left 3×3 block of a matrix is a rotation: If all entries are between -1 and 1, and the sum of the squares of the entries in a column looks like it's approximately one, it's probably a rotation. Finally, if the bottom row is not [0 0 0 1], you generally know that your transformation is projective rather than affine.

11.1.1 Projective Transformation Theorems

While projective transformations on 3-space are analogous to those in two dimensions, it's worth explicitly writing down a few of their properties.

A projective transformation is completely specified by its action on a **projective frame,** which consists of five points in space, no four of them coplanar. (The proof is exactly analogous to the two-dimensional proof.)

Every projective transformation on 3-space is determined by a linear transformation of the $w = 1$ subspace of 4-space, represented by a 4×4 matrix $\mathbf{M}$, followed by the **homogenizing transformation**

$$H(x, y, z, w) = (\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1). \tag{11.6}$$

If the last row of the matrix $\mathbf{M}$ is $\begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix}$, then the transformation takes the $w = 1$ plane to itself, so H has no effect, and the projective transformation is in fact an affine transformation whose matrix is $\mathbf{M}$.

> **Inline Exercise 11.1:** Suppose the last row of $\mathbf{M}$ is $\begin{bmatrix}0 & 0 & 0 & k\end{bmatrix}$ for some $k \neq 0, 1$. Show that in this case, the projective transformation defined by $\mathbf{M}$ is *still* affine. What is the matrix for this affine transformation? Hint: It's not $\mathbf{M}$!

The matrix $\mathbf{M}$ representing a projective transformation is not unique (as you can conclude from Inline Exercise 11.1). If $\mathbf{M}$ represents some transformation, so does $c\mathbf{M}$ for any nonzero constant c, because if $\mathbf{k} = \mathbf{M}\mathbf{v}$, then $(c\mathbf{M})\mathbf{v} = c\mathbf{k}$; homogenizing $c\mathbf{k}$ involves divisions like $\frac{ck_x}{ck_w} = \frac{k_x}{k_w}$, which produce the same results as homogenizing $\mathbf{k}$ itself.

The last row of the matrix $\mathbf{M}$ for a projective transformation determines the equation of the plane on which the transformation is undefined (i.e., the "plane sent to infinity"). If the last row is $\begin{bmatrix}A & B & C & D\end{bmatrix}$, then a point $\begin{bmatrix}x & y & z & 1\end{bmatrix}^{\mathbf{T}}$

1. Formally, we should say that $\mathbf{SO}(2)$, the set of 2×2 rotation matrices, is a one-dimensional **manifold,** which is, informally, a smooth shape with the property that at every point, there is essentially only one direction in which to move; in the case of the circle, this "direction" is that of increasing or decreasing angle. By contrast, the surface of the Earth is a 2-manifold, because at each point there are two independent directions for motion—at any point except the poles, one can take these to be north-south and east-west; any other direction is a combination of these two.

will be sent to infinity exactly if, after transformation, its w-coordinate is zero, that is, if

$$Ax + By + Cz + D = 0. \tag{11.7}$$

That equation defines a plane in 3-space.

Inline Exercise 11.2: In the case described above in which a projective transformation is actually affine, which points in $xyzw$-coordinates form the "plane sent to infinity"? It's important to include w in your computation.

11.2 Rotations

Rotations in 3-space are much more complicated than those in the plane, but much of that complexity is of little significance for the casual user. We therefore present the essentials in this section, but we provide a much longer discussion of rotations in the web materials for this chapter.

We begin with some easily derived formulas that you're likely to use often. Then we'll discuss how to use notions like pitch, roll, and yaw (which are called Euler angles) to describe rotations, and how to describe a rotation by giving an axis of rotation and an angle through which to rotate (Rodrigues' formula), as well as how to find the axis and angle for a rotation (a computation that's also due to Euler). Both of these descriptions of rotations have limitations that make them unsuitable for interpolating between rotations, so we'll consider a third way to describe rotations: To each point $\mathbf{q}$ of the sphere $\mathbf{S^3}$ in four-dimensional space $\mathbf{R^4}$, we can associate a rotation $K(\mathbf{q})$ in a very natural way. There's a small problem, however: The points $\mathbf{q}$ and $-\mathbf{q}$ of $\mathbf{S^3}$ correspond to the *same* rotation, so our correspondence is two-to-one. It turns out to still be easy to use this description of rotations to perform interpolation.

11.2.1 Analogies between Two and Three Dimensions

Rotations in two dimensions given by matrices of the form

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{11.8}$$

generalize nicely to three dimensions and higher. For instance, we can take the matrix for rotation through the angle θ in two dimensions and expand it to get

$$R_{xy}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{11.9}$$

which is the **rotation by angle θ in the xy-plane of 3-space.** As we mentioned in Chapter 10, this is also sometimes called **rotation about z by the angle θ.** The advantage of calling it rotation in the xy-plane is that there is an easy mnemonic associated to it: For small values of θ, the unit vector in the x-direction is rotated *toward* the unit vector in the y-direction. Corresponding statements are true of R_{yz}

and R_{zx}, which are written below. There's another advantage: While rotations in 3-space always have an axis (see the web material for a proof) those in 2-space do not (e.g., there's no vector in $\mathbf{R^2}$ left invariant by rotation through 30°), and neither do those in 4-space. But in all cases rotations can be described in terms of planes of rotation.

The analogous rotations in the *yz*- and *zx*-planes are given by

$$R_{yz}(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \text{ and} \tag{11.10}$$

$$R_{zx}(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}, \tag{11.11}$$

which can also be called rotation about *x* and rotation about *y*, respectively.

In contrast to the two-dimensional situation, where we found that the set of 3×3 rotation matrices was one-dimensional, in three dimensions the set $\mathbf{SO}(3)$ of 3×3 rotation matrices is *three*-dimensional. It is not, however, just a three-dimensional Euclidean space. One way to see it's three-dimensional is to find a mostly one-to-one mapping from an easy-to-understand three-dimensional object to $\mathbf{SO}(3)$ (just as the latitude-longitude parameterization of the 2-sphere shows us that the 2-sphere is two-dimensional). We'll actually describe three such mappings, each with its own advantages and disadvantages. The first of these mappings is through **Euler angles.** This mapping is "mostly one-to-one," in much the same way that the mapping of latitude and longitude to points on the globe is mostly one-to-one: Each point on the international date line has two longitudes (180E and 180W), and each pole has infinitely many longitudes, but each other sphere point corresponds to a single latitude-longitude pair.

11.2.2 Euler Angles

Euler angles are a mechanism for creating a rotation through a sequence of three simpler rotations (called roll, pitch, and yaw). This decomposition into three simpler rotations can be done in several ways (yaw first, roll first, etc.); unfortunately, just about every possible way is used in *some* discipline. You'll need to get used to the idea that there's no single correct definition of Euler angles.

The most commonly used definition in graphics describes a rotation by Euler angles (ϕ, θ, ψ) as a product of three rotations. The matrix $\mathbf{M}$ for the rotation is therefore a product of three others:

$$\mathbf{M} = R_{yz}(\psi)R_{zx}(\theta)R_{xy}(\phi). \tag{11.12}$$

Thus, objects are first rotated by angle ϕ in the *xy*-plane, then by angle θ in the *zx*-plane, and then by angle ψ in the *yz*-plane. The number ϕ is called pitch, θ is called yaw, and ψ is called roll. If you imagine yourself flying in an airplane (see Figure 11.1) along the *x*-axis (with the *y*-axis pointing upward) there are three direction changes you can make: Turning left or right is called **yawing,** pointing up or down is called **pitching,** and rotating about the direction of travel is called **rolling.** These three are independent in the sense that you can apply any one without the others. You can, of course, also apply them in sequence.

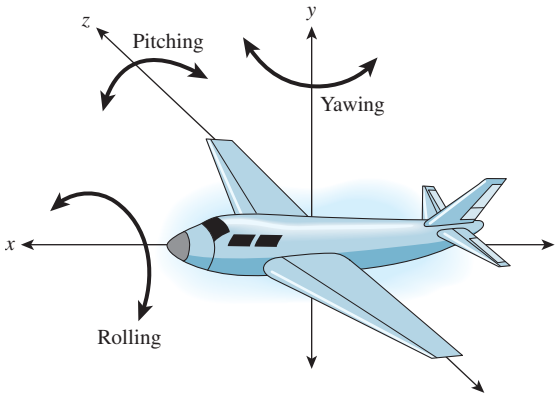


Figure 11.1: An airplane that flies along the x-axis can change direction by turning to the left or right (yawing), pointing up or down (pitching), or simply spinning about its axis (rolling).

Writing this out in matrices, we have

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & -\sin\psi \\ 0 & \sin\psi & \cos\psi \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{11.13}$$

$$= \begin{bmatrix} \cos\theta\cos\phi & -\cos\theta\sin\phi & \sin\theta \\ * & * & -\sin\psi\cos\theta \\ * & * & \cos\psi\cos\theta \end{bmatrix}. \tag{11.14}$$

With the proper choice of ϕ, θ, and ψ, such products represent all possible rotations. To see this, we'll show how to find ϕ, θ, and ψ from a rotation matrix $\mathbf{M}$. In other words, having shown how to convert a (ϕ, θ, ψ) triple to a matrix, we'll show how to convert a matrix $\mathbf{M}$ to a triple (ϕ', θ', ψ'), a triple with the property that if we convert it to a matrix, we'll get $\mathbf{M}$.

The $(1, 3)$ entry of $\mathbf{M}$, according to Equation 11.14, must be $\sin\theta$, so θ is just the arcsine of this entry; the number thus computed will have a non-negative cosine. When $\cos\theta \neq 0$, the $(1, 1)$ and $(1, 2)$ entries of $\mathbf{M}$ are positive multiples of $\cos\phi$ and $-\sin\phi$ by the same multiplier; that means $\phi = \text{atan2}(-m_{21}, m_{11})$. We can similarly compute ψ from the last entries in the second and third rows. In the case where $\cos\theta = 0$, the angles ϕ and ψ are not unique (much as the longitude of the North Pole is not unique). But if we pick $\phi = 0$, we can use the lower-left corner and atan2 to compute a value for ψ. The code is given in Listing 11.1, where we are assuming the existence of a 3×3 matrix class, `Mat33`, which uses zero-based indexing. The angles returned are in radians, not degrees.

Listing 11.1: Code to convert a rotation matrix to a set of Euler angles.

```
1 void EulerFromRot(Mat33 m, out double psi,
2                            out double theta,
3                            out double phi)
4 {
5   theta = Math.asin(m[0,2]) //using C# 0-based indexing!
6   double costheta = Math.cos(th);
7   if (Math.abs(costheta) == 0){
8       phi = 0;
```

```
9       psi = Math.atan2(m[2,1], m[1,1]);
10    }
11    else
12    {
13        phi = atan2(-m[0,1], m[0,0]);
14        psi = atan2(-m[1,2], m[2,2]);
15    }
16  }
```

It remains to verify that the values of θ, ϕ, and ψ determined produce matrices which, when multiplied together, really *do* produce the given rotation matrix $\mathbf{M}$, but this is a straightforward computation.

Inline Exercise 11.3: Write a short program that creates a rotation matrix from Rodrigues' formula (Equation 11.17 below) and computes from it the three Euler angles. Then use Equation 11.14 to build a matrix from these three angles, and confirm that it is, in fact, your original matrix. Use a random unit direction vector and rotation amount in Rodrigues' formula.

Aside from the special case where $\cos\theta = 0$ in the code above, we have a one-to-one mapping from rotations to (θ, ϕ, ψ) triples with $-\pi/2 < \theta \leq \pi/2$ and $-\pi < \phi, \psi \leq \pi$. Thus, the set of rotations in 3-space is three-dimensional.

In general, you can imagine controlling the attitude of an object by specifying a rotation using θ, ϕ, and ψ. If you change any one of them, the rotation matrix changes a little, so you have a way of maneuvering around in $\mathbf{SO}(3)$. The $\cos\theta = 0$ situation is tricky, though. If $\theta = \pi/2$, for instance, we find that multiple (ϕ, ψ) pairs give the same result; varying ϕ and ψ turns out to not produce independent changes in the attitude of the object. This phenomenon, in various forms, is called **gimbal lock,** and is one reason that Euler angles are not considered an ideal way to characterize rotations.

11.2.3 Axis-Angle Description of a Rotation

One way to rotate 3-space is to pick a particular axis (i.e., a unit vector) and rotate about that direction by some amount. The matrix R_{xy} does this when the axis is the z-axis, for instance. We show, in the web materials, that *every* rotation in 3-space is rotation about some axis by some angle. Rodrigues [Rod16] discovered a formula to build a rotation for any axis and angle of rotation, and thus to produce *any* rotation matrix. We let

$$\boldsymbol{\omega} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \tag{11.15}$$

denote the unit-vector axis of rotation and θ the amount of rotation about $\boldsymbol{\omega}$ (measured counterclockwise as viewed from the tip of $\boldsymbol{\omega}$ looking toward the origin).

To express the rotation we seek, we'll need to use cross products a good deal. The function $\mathbf{v} \mapsto \boldsymbol{\omega} \times \mathbf{v}$ is a linear transformation from $\mathbf{R}^3$ to itself; the matrix for this transformation is

$$\mathbf{J}_{\boldsymbol{\omega}} = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}. \tag{11.16}$$

Inline Exercise 11.4: (a) What's $\omega \times \omega$?
(b) Show that $\mathbf{J}_\omega \omega = \mathbf{0}$ as expected.
(c) Suppose that $\mathbf{v}$ is a unit vector perpendicular to ω. Explain why $\omega \times \mathbf{v}$ is perpendicular to both, and why $\omega \times (\omega \times \mathbf{v}) = -\mathbf{v}$.

The rotation matrix we seek is then

$$\mathbf{M} = \mathbf{I} + \sin(\theta)\mathbf{J}_\omega + (1 - \cos\theta)\mathbf{J}_\omega^2. \tag{11.17}$$

From Inline Exercise 11.4, it's clear that $\mathbf{M}\omega = \omega$. And if $\mathbf{v}$ is perpendicular to ω, then

$$\begin{aligned}
\mathbf{M}\mathbf{v} &= \mathbf{I}\mathbf{v} + \sin(\theta)\mathbf{J}_\omega\mathbf{v} + (1 - \cos\theta)\mathbf{J}_\omega^2\mathbf{v} && (11.18)\\
&= \mathbf{v} + \sin(\theta)\omega \times \mathbf{v} + (1 - \cos\theta)(\omega \times (\omega \times \mathbf{v})) && (11.19)\\
&= \mathbf{v} + \sin(\theta)\omega \times \mathbf{v} + (1 - \cos\theta)(-\mathbf{v}) && (11.20)\\
&= \sin(\theta)\omega \times \mathbf{v} + \cos(\theta)(\mathbf{v}), && (11.21)
\end{aligned}$$

which is just the rotation of $\mathbf{v}$ in the plane perpendicular to ω by an angle θ. Since $\mathbf{M}$ does the right thing to ω and to vectors perpendicular to ω, it must be the right matrix, as per the Transformation Uniqueness principle (see Figure 11.2).

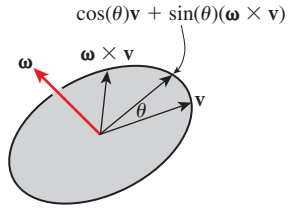


Figure 11.2: When $\mathbf{v}$ *is orthogonal to* ω, $\mathbf{v}$ *and* $\omega \times \mathbf{v}$ *form a basis for the plane perpendicular to* ω.

In coordinate form, it's

$$\mathbf{M} = \sin\theta \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} \tag{11.22}$$

$$+(1 - \cos\theta) \begin{bmatrix} -\omega_y^2 - \omega_z^2 & \omega_x\omega_y & \omega_z\omega_x \\ \omega_x\omega_y & -\omega_z^2 - \omega_x^2 & \omega_y\omega_z \\ \omega_z\omega_x & \omega_y\omega_z & -\omega_x^2 - \omega_y^2 \end{bmatrix} + \mathbf{I}, \tag{11.23}$$

where we've used the fact that ω is a unit vector to simplify things a little. But the earlier form is far easier to program correctly.

11.2.4 Finding an Axis and Angle from a Rotation Matrix

As we said earlier, it's a theorem that every rotation of 3-space has an axis (i.e., a vector that it leaves untouched). We can use Rodrigues' formula to recover the axis from the matrix. We'll follow the approach of Palais and Palais [PP07].

We know every rotation matrix has an axis ω and an amount of rotation, θ, about ω; Rodrigues' formula tells us the matrix must be

$$\mathbf{M} = \mathbf{I} + \sin(\theta)\mathbf{J}_\omega + (1 - \cos\theta)\mathbf{J}_\omega^2 \tag{11.24}$$

for some unit vector ω and some angle θ.

The trace of this matrix (the sum of the diagonal entries) is

$$\begin{aligned}
\mathbf{tr}(\mathbf{M}) &= \mathbf{tr}(\mathbf{I} + \sin(\theta)\mathbf{J}_\omega + (1 - \cos\theta)\mathbf{J}_\omega^2)\\
&= \mathbf{tr}(\mathbf{I}) + \sin(\theta)\mathbf{tr}(\mathbf{J}_\omega) + (1 - \cos\theta)\mathbf{tr}(\mathbf{J}_\omega^2)\\
&= 3 + (1 - \cos\theta)(-2(\omega_x^2 + \omega_y^2 + \omega_z^2))\\
&= 3 + (1 - \cos\theta)(-2)\\
&= 1 + 2\cos\theta,
\end{aligned}$$

so we can recover the angle of rotation by computing

$$\theta = \cos^{-1}\left(\frac{\mathbf{tr}(\mathbf{M}) - 1}{2}\right). \tag{11.25}$$

Two special cases arise at this point, corresponding to the two ways that $\sin\theta$ can be zero.

1. If $\theta = 0$, then *any* unit vector serves as an "axis" for the rotation, because the rotation is the identity matrix.

2. If $\theta = \pi$, then *twice* the rotation has angle 2π, and thus is the identity, that is, our rotation $\mathbf{M}$ must satisfy $\mathbf{M}^2 = \mathbf{I}$. From this we find that

$$\mathbf{M}(\mathbf{M} + \mathbf{I}) = \mathbf{M}^2 + \mathbf{M} = \mathbf{I} + \mathbf{M} = \mathbf{M} + \mathbf{I}. \tag{11.26}$$

   That means that when we multiply $\mathbf{M}$ by $\mathbf{M} + \mathbf{I}$, every column of $\mathbf{M} + \mathbf{I}$ remains unchanged. So any nonzero column of $\mathbf{M} + \mathbf{I}$, when normalized, can serve as the axis of rotation. We know that at least one column of $\mathbf{M} + \mathbf{I}$ is nonzero; otherwise, $\mathbf{M} = -\mathbf{I}$. But this is impossible, because the determinant of $-\mathbf{I}$ is -1, while that of $\mathbf{M}$ is $+1$.

In the general case, when $\sin\theta \neq 0$, we can compute $\mathbf{M} - \mathbf{M}^{\mathbf{T}}$ to get

$$\begin{aligned}\mathbf{M} - \mathbf{M}^{\mathbf{T}} = \; & \mathbf{I} + \sin(\theta)\mathbf{J}_\omega + (1 - \cos\theta)\mathbf{J}_\omega^2 - \\ & (\mathbf{I}^{\mathbf{T}} + \sin(\theta)\mathbf{J}_\omega^{\mathbf{T}} + (1 - \cos\theta)(\mathbf{J}_\omega^2)^{\mathbf{T}}).\end{aligned} \tag{11.27}$$

Because $\mathbf{J}_\omega^{\mathbf{T}} = -\mathbf{J}_\omega$ and $(\mathbf{J}_\omega^2)^{\mathbf{T}} = \mathbf{J}_\omega^2$, this simplifies to

$$\mathbf{M} - \mathbf{M}^{\mathbf{T}} = 2\sin(\theta)\mathbf{J}_\omega. \tag{11.28}$$

Dividing by $2\sin\theta$ gives the matrix $\mathbf{J}_\omega$, from which we can recover ω. The code is given in Listing 11.2.

Listing 11.2: Code to find the axis and angle from a rotation matrix.

```
void RotationToAxisAngle(
   Mat33 m,
   out Vector3D omega,
   out double theta)
{
// convert a 3x3 rotation matrix m to an axis-angle representation

  theta = Math.acos( (m.trace()-1)/2);
  if (θ is near zero)
  {
      omega = Vector3D(1,0,0); // any vector works
      return;
  }
  if (θ is near π)
  {
      int col = column with largest entry of m in absolute value;
      omega = Vector3D(m[0, col], m[1, col], m[2, col]);
      return;
  }
  else
  {
      mat 33 s = m - m.transpose();
      double x = -s[1,2], y = s[0,2]; z = -s[1,1];
```

```
24      double t = 2 * Math.Sin(theta);
25      omega = Vector3D(x/t, y/t, z/t);
26      return;
27    }
28  }
```

A few observations are in order.

- For small θ, $\mathbf{M}$ is nearly the identity.
- For small θ, the coefficient of the last term is approximately θ, while the coefficient of the middle term is $1 - \cos(\theta) \approx -\frac{\theta^2}{2}$; thus, the middle term is far smaller than the last term. So to first order, $\mathbf{M} \approx I + \sin\theta \mathbf{J}_\omega$.

11.2.5 Body-Centered Euler Angles

Suppose we have a model of an airplane whose vertices are stored in an $n \times 3$ array $\mathbf{V}$. We've rotated the model to some position that we like by multiplying all the vertices by some rotation matrix $\mathbf{M}$, that is, we've computed

$$\mathbf{W} = \mathbf{MV}. \tag{11.29}$$

We now decide that we'd like to have the airplane model pitch up a little more (as if the pilot had pulled on the joystick). We *could* apply some Euler-angle rotations to the rotated vertices, that is, we could compute

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & -\sin\psi \\ 0 & \sin\psi & \cos\psi \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{W}. \tag{11.30}$$

The problem is that this would take the already rotated vertices and rotate them first about the world z-axis, which might point diagonally through the airplane, and then about the world y-axis, and then about the world x-axis. It would be very hard to choose ψ, θ, and ϕ to have the effect we are seeking. Such a transformation would be called a **world-centered rotation,** because the description of the rotation is in terms of the axes of the world coordinate system. We could instead compute

$$\mathbf{M} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & -\sin\psi \\ 0 & \sin\psi & \cos\psi \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{V}, \tag{11.31}$$

that is, apply some rotations to the object's vertices *before* applying the rotation $\mathbf{M}$ to the object. Such an operation is called a **body-centered rotation** or **object-centered rotation.** In this case, performing the rotation we seek is easy: We simply adjust the pitch angle ϕ. Of course, if we then want to adjust it further, we have to apply another body-centered rotation, and it appears that we're destined to accumulate a huge sequence of matrices. One solution is to explicitly compute the product so that we always have at most one matrix, plus three others temporarily being adjusted until they too can be folded into the matrix. Another approach is to represent matrices with quaternions, as we'll see below. In general, if $\mathbf{M}$ is the current transformation applied to the vertex set $\mathbf{V}$, and we alter it to $\mathbf{M}_1 = \mathbf{MA}$, then $\mathbf{A}$ is called a **body-centered** operation, while if we alter it to $\mathbf{M}_2 = \mathbf{CM}$, then $\mathbf{C}$ is called a **world-centered** operation.

11.2.6 Rotations and the 3-Sphere

The set $\mathbf{SO}(3)$ of all 3×3 rotation matrices can be difficult to understand. It is, in some sense, a subset of $\mathbf{R}^9$: Just read the nine entries of the matrix $\mathbf{M}$ in order to get the point in $\mathbf{R}^9$ that corresponds to $\mathbf{M}$. In a web extra, we give considerable detail on this set and its properties. Here, we'll give just the essentials. The main tool used to understand $\mathbf{SO}(3)$, to make computations involving $\mathbf{SO}(3)$ more numerically robust, and to let us reason about operations in $\mathbf{SO}(3)$ (like interpolation) by means of a familiar space, is $\mathbf{S^3}$, the 3-sphere, or the set of all points $\begin{bmatrix} w & x & y & z \end{bmatrix}^{\mathbf{T}}$ in 4-space whose distance from the origin is 1. We've shuffled the coordinates on purpose to make some of what we say below involve *less* shuffling. We'll also, in this section, talk about *points* of $\mathbf{S^3}$, but we'll always write them as *vectors* so that we can form linear combinations of them.

Just as you can wrap a line segment into a circle (joining the two boundary points into one point of the circle, as in Figure 11.3), or wrap a disk into a sphere (with the whole boundary circle becoming one point of the sphere, as in Figure 11.4), you can wrap a solid ball in 3-space into a 3-sphere, by collapsing the whole boundary sphere to a point. To do so, you must work in the fourth dimension, but the idea is to reason about the 3-sphere by analogy.

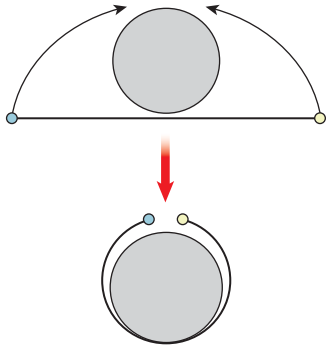


Figure 11.3: Wrapping a line onto a circle.

For instance, if we take two perpendicular unit vectors $\mathbf{u}$ and $\mathbf{v}$ in the unit circle and construct all points of the form $\cos(\theta)\mathbf{u} + \sin(\theta)\mathbf{v}$, these points cover the whole circle (see Figure 11.5). Similarly, in the 2-sphere, if we have two perpendicular unit vectors, their cosine-sine combinations form a **great circle,** that is, the intersection of the sphere with a plane through its center (see Figure 11.6). In fact, the same thing is true for the 3-sphere as well: Cosine-sine combinations of perpendicular vectors traverse a great circle on the 3-sphere. And the arc of this circle from $\mathbf{u}$ to $\mathbf{v}$ (i.e, from $\theta = 0$ to $\pi/2$) is the shortest path between them, just as in the lower dimensions.

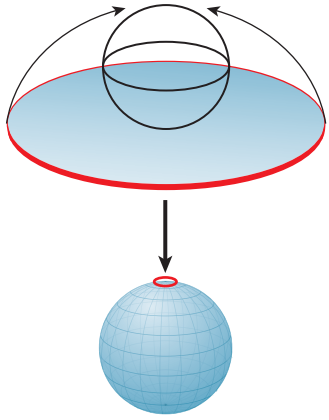


Figure 11.4: Wrapping a disk onto a sphere; all points of the circular edge of the disk are sent to the North Pole.

There's a mapping from $\mathbf{S^3}$ to $\mathbf{SO}(3)$, given by

$$K : \mathbf{S^3} \to \mathbf{SO}(3) : \tag{11.32}$$

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \mapsto \begin{bmatrix} a^2+b^2-c^2-d^2 & 2(bc-ad) & 2(ac+bd) \\ 2(ad+bc) & a^2-b^2+c^2-d^2 & 2(cd-ab) \\ 2(bd-ac) & 2(ab+cd) & a^2-b^2-c^2+d^2 \end{bmatrix}. \tag{11.33}$$

The map K has several nice properties.

- It's *almost* one-to-one. In fact, it's a two-to-one map; for any $\mathbf{q} \in \mathbf{S^3}$, $K(\mathbf{q}) = K(-\mathbf{q})$, as you can see by looking at the formula.
- Great circles on $\mathbf{S^3}$ are sent, by K, to geodesics (paths of shortest length) in $\mathbf{SO}(3)$.
- $K(\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}^{\mathbf{T}}) = \mathbf{I}$.

This mapping arises from a definition of a kind of "multiplication" on $\mathbf{R^4}$, closely analogous to the way we can treat points of $\mathbf{R^2}$ as complex numbers and multiply them together. The multiplication on $\mathbf{R^4}$ is *not* commutative, which causes some difficulties, but otherwise it's closely analogous to multiplication of complex numbers. The set $\mathbf{R^4}$, together with this multiplication operation, is called the **quaternions** (which is why we use a boldface $\mathbf{q}$ for a typical element of $\mathbf{S^3}$).

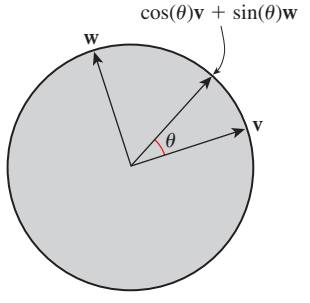


Figure 11.5: The set of all cosine-sine combinations of $\mathbf{v}$ *and* $\mathbf{w}$ *wraps around the whole circle.*

The web material for this chapter describes the quaternions in detail, derives the mapping K above, and shows how it's related to Rodrigues' formula. For most purposes in graphics, it's sufficient to know the three properties above, and one more fact, which we now develop.

If we have a point $\mathbf{q} = \begin{bmatrix} a & b & c & d \end{bmatrix}^{\mathbf{T}} \in \mathbf{S^3}$, we know $-1 \leq a \leq 1$, so a is the cosine of some number. We let

$$\theta = \arccos(a). \tag{11.34}$$

Furthermore, since $\begin{bmatrix} a & b & c & d \end{bmatrix}^{\mathbf{T}} \in \mathbf{S^3}$, we know that $a^2 + b^2 + c^2 + d^2 = 1$. Thus,

$$1 = a^2 + b^2 + c^2 + d^2 \tag{11.35}$$

$$= \cos^2(\theta) + b^2 + c^2 + d^2 \tag{11.36}$$

so that $\begin{bmatrix} b & c & d \end{bmatrix}^{\mathbf{T}}$ is a vector of squared length $\sin^2(\theta)$. If $a \neq \pm 1$, then $\sin(\theta) \neq 0$, and we can let

$$\boldsymbol{\omega} = \begin{bmatrix} 0 & \frac{b}{\sin(\theta)} & \frac{c}{\sin(\theta)} & \frac{d}{\sin(\theta)} \end{bmatrix}^{\mathbf{T}}, \tag{11.37}$$

and say that

$$\mathbf{q} = \cos(\theta) \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}^{\mathbf{T}} + \sin(\theta)\boldsymbol{\omega}. \tag{11.38}$$

In the case where $\sin(\theta) = 0$, we can choose any unit vector for $\boldsymbol{\omega}$. In short, every element of $\mathbf{S^3}$ can be written in the form

$$\mathbf{q} = \cos(\theta) \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}^{\mathbf{T}} + \sin(\theta)\boldsymbol{\omega}, \tag{11.39}$$

where $0 \leq \theta \leq \pi$ and $\boldsymbol{\omega}$ is a unit vector in the xyz-subspace of $\mathbf{S^3}$, that is, perpendicular to $\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}^{\mathbf{T}}$.

With a good deal of algebra, you can plug in $a = \cos(\theta)$, $b = \sin(\theta)\omega_x$, $c = \sin(\theta)\omega_y$, and $d = \sin(\theta)\omega_z$ in Equation 11.32, and discover that it's *exactly* the same matrix you get if you apply Rodrigues' formula to build a rotation about the xyz-vector $\boldsymbol{\omega}$ by angle 2θ (note the factor of two!).

The map K has a great deal in common with the map $K_1 : \mathbf{S^1} \to \mathbf{S^1} : (\cos(\theta), \sin(\theta)) \mapsto (\cos(2\theta), \sin(2\theta))$. Like K, the map K_1 is also two-to-one. For instance, the points at $\theta = 0$ and $\theta = \pi$ both get sent to the point at $\theta = 0$ by K_1. In fact, the points at θ and $\theta + \pi$ are both sent to the point at θ, for any angle θ; in other words, K_1 maps each pair of antipodal points to the same point. If you wanted to interpolate between, say, $\pi/4$ and $3\pi/4$ in the codomain, you could pick the points $\pi/8$ and $3\pi/8$ in the domain, interpolate between them, and then apply K_1 to the interpolated angles and get the result you expect. Of course, if instead of picking $3\pi/8$, you pick $11\pi/8$, then the interpolation will run along the *long* path between $\pi/4$ and $3\pi/4$, as shown in Figure 11.7.

By the way, although K is not invertible, it's easy to build a *kind* of inverse: Given $\mathbf{M} \in \mathbf{SO}(3)$, we can find *an* element $\mathbf{q} \in \mathbf{S^3}$ with $K(\mathbf{q}) = \mathbf{M}$; we just cannot claim that it is *the* element with this property. Recall that Rodrigues' formula tells us that every rotation matrix has the form

$$\mathbf{M} = \mathbf{I} + \sin(\theta)\,\mathbf{J}_{\boldsymbol{\omega}} + (1 - \cos\theta)\mathbf{J}_{\boldsymbol{\omega}}^2, \tag{11.40}$$

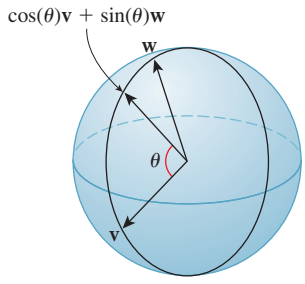


Figure 11.6: All cosine-sine combinations of two perpendicular unit vectors in the sphere again form a unit circle, called a great circle.

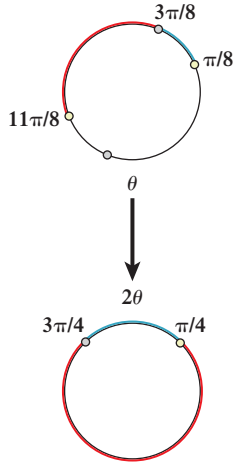


Figure 11.7: The blue path in the domain transforms to the short arc between $\pi/4$ and $3\pi/4$ in the codomain, while the red one transforms to the long arc.

where ω is a unit vector that's the axis of rotation of the matrix and θ is the angle of rotation. And we discussed how to recover the axis ω and the angle θ from an arbitrary rotation matrix (except that when the matrix was $\mathbf{I}$, the axis could be any unit vector and the angle was 0). The associated element $\mathbf{q}$ of $\mathbf{S^3}$ has $\cos(\theta/2)$ as its first coordinate and $\sin(\theta/2)\omega$ as its last three coordinates. The case where $\theta = 0$ and ω is indeterminate presents no problem, because $\sin(\theta/2) = 0$, so the last three entries are all zeroes. There *is* an ambiguity, however: When we found the axis ω and the angle θ we could instead have found $-\omega$ and $-\theta$; those two would have produced $-\mathbf{q}$ instead of $\mathbf{q}$. So our "inverse" to K really can produce one of two opposite values, depending on choices made in the axis-and-angle computation. To make all this concrete, we'll give pseudocode for a function L whose domain is $\mathbf{SO}(3)$ and whose codomain is *pairs* of antipodal points in $\mathbf{S^3}$; L will act as an inverse to K, in the sense that if $\mathbf{M} \in \mathbf{SO}(3)$ is a rotation matrix and $L(\mathbf{M}) = \{\mathbf{q}_1, -\mathbf{q}_1\}$ are two elements of $\mathbf{S^3}$, then $K(\mathbf{q}_1) = K(-\mathbf{q}_1) = \mathbf{M}$. In the pseudocode in Listing 11.3, neither `q1` nor `q2` is guaranteed to be a continuous function of the entries of the matrix `m`.

Listing 11.3: Code to convert a rotation matrix to the two corresponding quaternions.

```
void RotationToQuaternion(Mat33 m, out Quaternion q1, out Quaternion q2)
{
// convert a 3x3 rotation matrix m to the two quaternions
// q1 and q2 that project to m under the map K.
  if (m is the identity)
  {
    q1 = Quaternion(1,0,0,0);
    q2 = -q1;
    return;
  }

  Vector3D omega;
  double theta;
  RotationToAxisAngle(m, omega, theta);

  q1 = Quaternion(Math.cos(theta/2), Math.sin(theta/2)*omega);
  q2 = -q1;
}
```

We now have a method for going from $\mathbf{S^3}$ to $\mathbf{SO}(3)$ and for going from $\mathbf{SO}(3)$ back to *pairs* of elements of $\mathbf{S^3}$. To interpolate between rotations in $\mathbf{SO}(3)$, we'll interpolate between points of $\mathbf{S^3}$.

11.2.6.1 Spherical Linear Interpolation

Suppose we have two points $\mathbf{q}_1$ and $\mathbf{q}_2$ of the unit sphere, and that $\mathbf{q}_1 \neq -\mathbf{q}_2$, that is, they're not antipodal. Then there's a unique shortest path between them, just as on the Earth there's a unique shortest path from the North Pole to any point except the South Pole. (There *is* a shortest path from the North to the South Pole; the problem is that it's no longer unique—any line of longitude is a shortest path.)

We'll now construct a path γ that starts at $\mathbf{q}_1$ (i.e., $\gamma(0) = \mathbf{q}_1$), ends at $\mathbf{q}_2$ (i.e., $\gamma(1) = \mathbf{q}_2$), and goes at constant speed along the shorter great arc between them. This is called **spherical linear interpolation** and was first described for use in computer graphics by Shoemake [Sho85], who called it **slerp.** There are three steps.

1. Find a vector $\mathbf{v} \in \mathbf{S^3}$ that's in the $\mathbf{q}_1$–$\mathbf{q}_2$ plane and is perpendicular to $\mathbf{q}_1$. We subtract the projection $(\mathbf{q}_2 \cdot \mathbf{q}_1)\mathbf{q}_1$ of $\mathbf{q}_2$ onto $\mathbf{q}_1$ from $\mathbf{q}_2$ to get a vector perpendicular to $\mathbf{q}_1$, and normalize:
$$\mathbf{v} = \frac{\mathbf{q}_2 - (\mathbf{q}_2 \cdot \mathbf{q}_1)\mathbf{q}_1}{\|\mathbf{q}_2 - (\mathbf{q}_2 \cdot \mathbf{q}_1)\mathbf{q}_1\|}. \tag{11.41}$$

2. Find a unit-speed path along the great circle from $\mathbf{q}_1$ through $\mathbf{v}$. That's just $\gamma(t) = \cos(t)\mathbf{q}_1 + \sin(t)\mathbf{v}$. This path reaches $\mathbf{q}_2$ at $t = \theta$, where $\theta = \cos^{-1}(\mathbf{q}_1 \cdot \mathbf{q}_2)$ is the angle between the two vectors.

3. Adjust the path so that it reaches $\mathbf{q}_2$ at time 1 rather than at time θ, by multiplying t by θ.

The resultant code is shown in Listing 11.4.

Listing 11.4: Code for spherical linear interpolation between two quaternions.

```
double[4] slerp(double[4] q1, double[4] q2, double t)
{
  assert(dot(q1, q1) == 1);
  assert(dot(q2, q2) == 1);
  // build a vector in q1-q2 plane that's perp. to q1
  double[4] u = q2 - dot(q1, q2) * q2;
  u = u / length(u); // ...and make it a unit vector.
  double angle = acos(dot(q1, q2));
  return cos(t * angle) * q1 + sin(t * angle) * u;
}
```

As the argument to the cosine ranges from `0` to `angle`, the result varies from `q1` to `q2`.

11.2.6.2 Interpolating between Rotations

We now have all the tools we need to interpolate between rotations. Suppose that $\mathbf{M}_1$ and $\mathbf{M}_2$ are rotation matrices, with $\mathbf{M}_1$ corresponding to the quaternions $\pm\mathbf{q}_1$ and $\mathbf{M}_2$ corresponding to the quaternions $\pm\mathbf{q}_2$, as indicated schematically in Figure 11.8. Starting with $\mathbf{q}_1$, we determine which of $\mathbf{q}_2$ and $-\mathbf{q}_2$ is closer, and then interpolate along an arc between $\mathbf{q}_1$ and this point; to find interpolating rotations, we project to $\mathbf{SO}(3)$ via K.

Listing 11.5 shows the pseudocode.

Listing 11.5: Code to interpolate between two rotations expressed as matrices.

```
Mat33 RotInterp(Mat33 m1, Mat33 m2, double t)
// find a rotation that's t of the way from m1 to m2 in SO(3).
// m1 and m2 must be rotation matrices.
{
   if ( m1*m2^T == -I) {
      Report error; can't interpolate between opposite rotations.
   }
   Quaternion q1, q1p, q2, q2p;
   RotationToQuaternion(m1, q1, q1p);
   RotationToQuaternion(m2, q2, q2p);
   if (Dot(q1, q2) < 0) q2 = q2p;
   Quaternion qi = Quaternion.slerp(q1, q2, t);
   return K(qi); // K is the projection from S3 to SO(3)
}
```

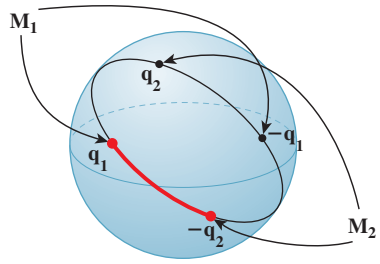


Figure 11.8: We have two rotation matrices, $\mathbf{M}_1$ and $\mathbf{M}_2$; the first corresponds to a pair of antipodal quaternions, $\pm\mathbf{q}_1$, and the second to a different pair of antipodal quaternions, $\pm\mathbf{q}_2$. Starting at $\mathbf{q}_1$, we choose whichever of $\mathbf{q}_2$ and $-\mathbf{q}_2$ is closer (in this case, $-\mathbf{q}_2$) and interpolate between them (as indicated by the thick red arc); we then can project the interpolated points to $\mathbf{SO}(3)$ to interpolate between $\mathbf{M}_1$ and $\mathbf{M}_2$.

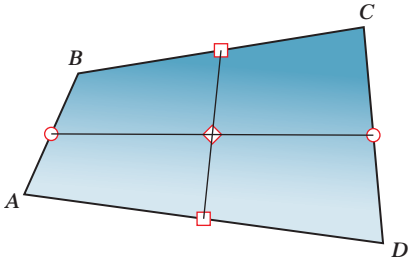


Figure 11.9: We can compute the midpoint of the quadrilateral ABCD by finding the midpoints of AB and CD (marked by circles), and then the midpoint of the segment between them, or *by doing the same process to the edges AD and BC (whose midpoints are marked by squares); the resultant quadrilateral midpoint (indicated by the diamond) is the same in both cases. This does not happen when we work with quaternions.*

With this notion of "interpolate between rotations" or "interpolate between quaternions" in hand, other operations, like blending together three or four rotations, also become possible. Indeed, even operations like the curve subdivision we did in Chapter 4 become possible in $\mathbf{SO}(3)$ instead of $\mathbf{R}^2$. One must be careful, however. While it's nice to be able to interpolate between quaternions in the same way we construct a segment between points in the plane, the analogy has some weaknesses: In the plane, we can construct points $(1-t)A + tB$ for t between 0 and 1, and they lie between A and B. If we use $t > 1$, we get a point "beyond B." On the other hand, if we do spherical linear interpolation between quaternions $\mathbf{q}_1$ and $\mathbf{q}_2$, as we increase t further and further, the result eventually wraps around the sphere and returns to $\mathbf{q}_1$.

Other "obvious" things fail as well. In the plane, we can find the center of a quadrilateral by bisecting opposite sides and finding the midpoint of the edge between these points. It doesn't matter which pair of opposite sides we choose—the result is the same, as shown in Figure 11.9. But with quaternions, it's generally not the same.

Inline Exercise 11.5: On the 2-sphere, let $A = B = (0,1,0)$, $C = (1,0,0)$, and $D = (0,0,1)$. Compute (by drawing—you should not need to perform any algebra) the center of the quadrilateral $ABCD$ twice, once using each pair of opposite sides, to verify that the results are not the same.

Buss [BF01] discusses thoroughly the challenges of working with such "affine combinations" of quaternions.

11.2.7 Stability of Computations

You probably recall solving problems in calculus where you know a position $\mathbf{x}(t)$ at time $t = 0$, and you're given a formula for velocity $\mathbf{x}'(t)$ for all t, and you need to find $\mathbf{x}$ at times other than $t = 0$. The simple method is to say that at $t = 0.1$, your position is approximately

$$\mathbf{x}(0.1) \approx \mathbf{x}(0) + 0.1\mathbf{x}'(0) \tag{11.42}$$

and then compute $\mathbf{x}(0.2) \approx \mathbf{x}(0.1) + 0.1\mathbf{x}'(0.1)$, etc. This is called **Euler integration** of the position with a time step of 0.1, and it gives a crude approximation of the solution to the problem. Picking a time step smaller than 0.1 produces better results, but at a cost of greater computational effort. Chapter 35 discusses this, and better approximations, extensively.

The analogy for the attitude (i.e., the rotation matrix currently applied to some model) is that you're given the attitude at time $t = 0$, and the change in attitude for all t, and you have to "integrate" to find the attitude at all times. The "update" step is

$$\mathbf{M}(0.1) = \mathbf{M}(0)(\mathbf{I} + 0.1\mathbf{M}'(0)) \tag{11.43}$$

so that it's multiplicative rather than additive. One problem is that $\mathbf{I} + 0.1\mathbf{M}'(0)$ is not actually a rotation matrix. It's very close, but not quite. So, after the update, we have to convert $\mathbf{M}$ into a rotation matrix, typically by applying the Gram-Schmidt process to the columns of $\mathbf{M}$ (i.e., normalize the first column; make the second perpendicular to it; normalize the second; make the third perpendicular to both; normalize the third column). This is computationally fairly expensive.

As an alternative, we can store the current attitude as a unit *quaternion* $\mathbf{q}$. The update, in that case, looks like this:

$$\mathbf{q}(0.1) = \mathbf{q}(0) + 0.1\mathbf{q}'(0). \tag{11.44}$$

Once again, the resultant vector at $t = 0.1$ is not *quite* what we want: It may not be a unit vector, so we have to normalize it. Notice, however, that normalizing a vector requires much less work than performing the Gram-Schmidt process on a whole matrix. And while a matrix can fail to be a rotation in many ways (one or more columns not unit length, various pairs of columns not perpendicular), a quaternion can fail to be a unit quaternion in only one way. For this reason, animation systems often represent attitude with a quaternion, converting it to a rotation matrix with the map K only when necessary. Computations on a quaternion tend to be more numerically stable than those on a rotation matrix as well.

11.3 Comparing Representations

We've now seen four ways to represent a 3D rigid reference frame (i.e., a set of three orthogonal unit vectors forming a right-handed coordinate system, based at some location P).

1. A 4×4 matrix $\mathbf{M}$, which we apply to the standard basis vectors at the origin to get the three vectors, and to the origin itself to get the basepoint P. The last row of the matrix must be $\begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix}$, and the upper-left 3×3 submatrix $\mathbf{S}$ must be a rotation matrix.

2. A 3×3 rotation matrix $\mathbf{S}$ and a translation vector $\mathbf{t} = P -$ origin.

3. Three Euler angles and a translation vector.

4. A unit quaternion and a translation vector.

Each has its advantages.

Option 1 is nice because it's easy to consider it as a transformation, which can be combined with other transformations by matrix multiplication. It makes sense to use this in a geometric modeling system. Converting between options 1 and 2 is straightforward, but option 2 has the advantage that it's easy to check that the matrix $\mathbf{S}$ is orthogonal by checking $\mathbf{S^T S} = \mathbf{I}$. If several such matrices have been multiplied, accumulating round-off errors, you can apply the Gram-Schmidt orthogonalization process to the result to adjust it back into orthogonal form, although this involves many multiplications and divisions, and several square roots.

Option 3 is useful, especially in a body-centered form, for representing things like aircraft or other first-person-control situations. Converting to and from matrix form is somewhat messy, however.

Option 4 is much favored in rigid-body animation. Converting to matrix form is easy; converting from matrix form is slightly messier because of the two-to-one nature of the quaternion-to-rotation map. Interpolation is particularly easy in quaternion form, and the equivalent of "reorthogonalization" in the matrix form is vector normalization for the quaternion, which is very fast. This is discussed further in Section 35.5.2.

11.4 Rotations versus Rotation Specifications

We've defined a rotation as a transformation having certain properties; in particular, it's a linear function from $\mathbf{R^3}$ to $\mathbf{R^3}$ represented by multiplication by some matrix. Thus, for instance, the transformation represented by multiplication by

$$\begin{bmatrix}0 & -1 & 0\\ 1 & 0 & 0\\ 0 & 0 & 1\end{bmatrix} \tag{11.45}$$

can be called rotation about the z-axis (or in the xy-plane) by 90°. But it's important to understand that multiplying a vector $\mathbf{v}$ by this matrix doesn't rotate $\mathbf{v}$ about the z-axis in the sense that this phrase is commonly used. The vector $\mathbf{v}$ is at no point rotated by 10°, or 20°, or 30°. The function simply takes the coordinates of $\mathbf{v}$ and returns the coordinates of the rotated-by-90° version of $\mathbf{v}$. Indeed, looking at the coordinates returned by the rotation, there's no way to tell whether they arose as the result of rotation by 90°, −270°, or 450°. This might seem irrelevant in the sense that we got the rotation we wanted, but when we consider the problem of *interpolating* rotations it's really quite significant. If we attempt to build an interpolation procedure `interp(M1, M2, t)` that takes as input two matrices

$\mathbf{M}_1$ and $\mathbf{M}_2$ and a fraction t by which to interpolate them, what should happen when $\mathbf{M}_1$ arises from the operation "rotate not at all" and so does $\mathbf{M}_2$? Obviously, the interpolated rotation will not rotate at all (i.e., it'll be the identity rotation). But what if $\mathbf{M}_1$ comes from "rotate not at all" and $\mathbf{M}_2$ comes from "rotate 360° about the z-axis"? We know that we want the halfway interpolation ($t = 0.5$) to be "rotate 180° about the z-axis," but there's no way for our procedure to compute this: In both our cases, the matrices $\mathbf{M}_1$ and $\mathbf{M}_2$ are the identity!

In many situations, the underlying desire is not to interpolate rotation matrices or rotation transformations, but to interpolate the rotation *specifications* and then compute the transformation associated to the interpolated specification. Unfortunately, interpolating specifications is not so easy. In the case of axis-angle specifications, it's easy when the axis doesn't change: One simply interpolates the angle. But interpolating the axis is a trickier matter. And suppose we consider two instances of the problem.

1. Rotation 1: Rotate by 0 about the x-axis. Rotation 2: Rotate by 90° about the x-axis.

2. Rotation 1: Rotate by 0 about the y-axis. Rotation 2: Rotate by 90° about the x-axis.

Should the halfway interpolated rotation in these two cases be the same or different? The initial and final rotations are identical in the two cases. The only difference is in the specification of the irrelevant direction in the no-angle rotation. Should that make a difference?

Yahia and Gagalowicz [YG89] describe a method for axis-angle interpolation in which the axis always has an effect, even when the angle is a multiple of 2π; aside from this artifact, the method is quite reasonable.

11.5 Interpolating Matrix Transformations

Despite the claim of the preceding section that, in general, we want to interpolate rotation *specifications* rather than the transformations themselves, there are several circumstances in which directly acting on the transformations makes sense. For instance, if we've written a physics simulator that computes the orientations and positions of bodies at certain times, but we'd like to fill in orientations and positions at intermediate times, we can ensure that the values provided at the key times by the simulator will be fairly close to the neighboring values (i.e., an object isn't going to spin 720° between key times) by working with small enough time steps. Given two such "nearby" transformations, can we interpolate between them?

Alexa et al. [Ale02] describe a method for interpolating transformations and more generally for forming linear combinations of transformations, provided that the transformations being combined are "near enough." The web material for this chapter describes this method, but since it involves the matrix exponential and other mathematical topics we don't wish to delve into, we omit it here.

11.6 Virtual Trackball and Arcball

As an application of our study of the space of rotations, we'll now examine two methods for controlling the attitude of a 3D object that we're viewing. These two

user-interface techniques are really a part of the general topic of 3D interaction (see Chapter 21, where we give actual implementations), but we discuss them here because they are so closely related to the study of $\mathbf{SO}(3)$.

Our standard 3D test program can display geometric objects modeled as meshes. Imagine that we have a fixed mesh K with vertices $P_0, P_1, \ldots, P_k$. We can, before displaying the mesh K, apply a transformation to each of the points P_i to create a new mesh. By displaying this new mesh, we see a transformed version of K. If we repeatedly vary the transformation applied to the mesh K, we'll see a sequence of new meshes that appears to move over time.

An alternative view of this is that we leave the mesh K fixed, but repeatedly change our virtual camera's position and orientation. We'll take the first approach here, however. Not surprisingly, the two are closely related: Moving the object in one direction is equivalent to moving the virtual camera in the opposite direction, for instance (as long as there's only one object in the world).

Suppose we want to be able to view the object from all sides. We'll assume that it's positioned at the origin so that applying rotations to it keeps it in the same location, but with a varying attitude. How can we control the viewing direction?

One easy-to-understand metaphor is to imagine the object as being encased in a large spherical block of glass (see Figure 11.10). This glass ball is so large that if it were drawn on the display, it would fill up as much of the display as possible. (For a square display, it would touch all four sides of the display.) We can now imagine interacting with this virtual sphere by clicking on some point of the sphere, dragging some distance, and releasing. If we click first at a point P and release at the point Q, this is supposed to rotate the sphere so as to make the point P move to the point Q along a great-circle arc (i.e., to rotate in the plane defined by P, Q, and the origin).

Of course, when we click on a point of the display with the mouse cursor we're not actually clicking on the *sphere*—what we get are the coordinates of the point on the display surface. This in turn must be used to determine a point of the sphere itself. Suppose, for now, that we know the position C of the virtual camera, and that in response to a mouse-click, we are given the position of a corresponding point S on a plane in 3-space that corresponds to our display (see Figure 11.11).

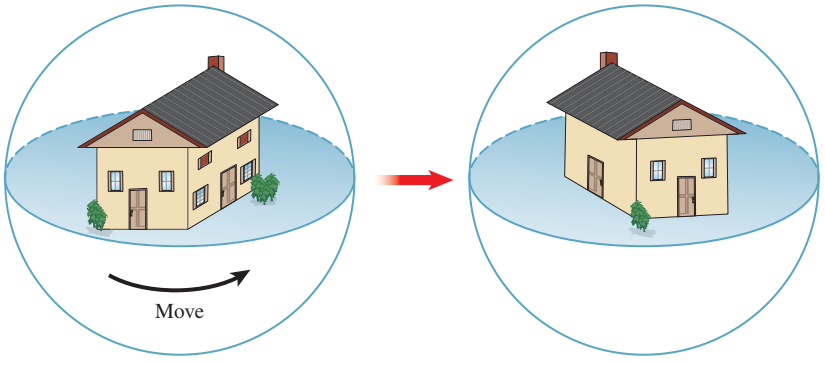


Figure 11.10: The object being viewed is imagined as lying in a large glass sphere. Moving a point on the surface of the sphere moves the object inside.

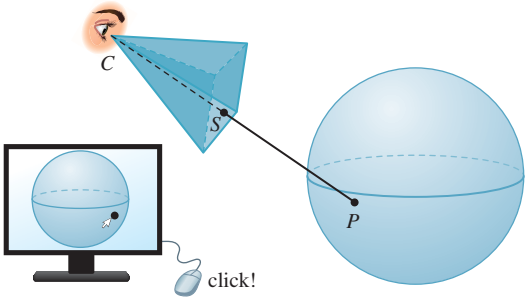


Figure 11.11: When the user clicks near the lower-right corner of the display, we can recover the 3-space coordinates of a corresponding point S of the imaging plane in 3-space; we'll use this to determine where a ray from the eye through this point hits the virtual sphere.

To determine the point P corresponding to this click, we ask where the ray parameterized by

$$R(t) = C + t(S - C) \tag{11.46}$$

meets the virtual sphere, which we'll assume, for simplicity, is the unit sphere defined by $x^2 + y^2 + z^2 = 1$. In other words, the unit sphere, if displayed, would just touch two sides of our display rectangle. For the point $R(t)$ to lie on the sphere, its coordinates (which we'll call r_x, r_y, and r_z) must satisfy the defining equation of the sphere, that is,

$$r_x^2 + r_y^2 + r_z^2 = 1. \tag{11.47}$$

Alternatively, we can consider the vector from the origin $\mathbf{O}$ to $R(t)$, that is, $C + t(S - C) - \mathbf{O}$; this vector must have unit length, which means it must satisfy $(R(t) - \mathbf{O}) \cdot (R(t) - \mathbf{O}) = 1$. Letting $\mathbf{u}$ denote $S - C$, this becomes

$$(C - \mathbf{O} + t\mathbf{u}) \cdot (C - \mathbf{O} + t\mathbf{u}) = 1, \tag{11.48}$$

which we can simplify and expand; letting $\mathbf{c} = C - \mathbf{O}$, we get

$$(\mathbf{u} \cdot \mathbf{u})t^2 + (2\mathbf{c} \cdot \mathbf{u})t + \mathbf{c} \cdot \mathbf{c} = 1, \tag{11.49}$$

which is a quadratic in t; we solve to get

$$t = \frac{-\mathbf{c} \cdot \mathbf{u} \pm \sqrt{(\mathbf{c} \cdot \mathbf{u})^2 - (\mathbf{u} \cdot \mathbf{u})(\mathbf{c} \cdot \mathbf{c})}}{\mathbf{u} \cdot \mathbf{u}}. \tag{11.50}$$

The smaller t value—call it t_1—corresponds to the first intersection of the ray with the sphere; using this, we can compute the sphere point

$$P = C + t_1(S - C). \tag{11.51}$$

(It's possible that both solutions for t are not real numbers, in which case the ray does not intersect the sphere: The user did not click on the image of the virtual sphere on the display.)

As the user drags the mouse, we can, at each instant, compute the corresponding sphere point Q in the same way. From P and Q, we compute a rotation of the

sphere that takes P to Q along a great circle; this rotation must be about a unit vector orthogonal to P and Q, and it must have magnitude $\cos^{-1}(P \cdot Q)$. Rodrigues' formula provides the matrix.

We use this matrix to multiply all vertex coordinates of the original mesh to get a new mesh for display; the resultant operation feels completely natural to many people.

Two problems remain: What happens when the user drags to a point outside the virtual sphere? And what happens when the user's initial click is outside the virtual sphere?

Various solutions have been implemented. When the user drags outside the virtual sphere, one good solution is to treat the point Q as being the nearest point on the sphere to the ray that the user is describing; this corresponds to using $t = -\mathbf{c} \cdot \mathbf{u}/\mathbf{u} \cdot \mathbf{u}$ in the quadratic solution.

When the user clicks outside the virtual sphere, one can treat subsequent mouse-drags as instructions to rotate about the view direction, like the "rotate object" interaction in most 2D drawing programs.

One problem with the virtual-sphere controller described so far is that the action of the controller depends on the first point the user clicked; in a long interaction sequence, this may be gradually forgotten. An improved approach is to treat each mouse-drag event as defining a new motion of the sphere, taking the start point to the endpoint. Thus, a click and drag becomes a sequence of mouse positions $P_0 = P, P_1, P_2, \ldots, P_n = Q$, and the object is rotated by a sequence of virtual-sphere rotations from P_0 to P_1, followed by the rotation defined by P_1 and P_2, etc.

With this modified version of the virtual sphere, it can be difficult to return to one's starting position; in trade for this, one gets the advantage that a click-and-drag-in-small-circles motion causes the object to spin about the view direction, which users seem to learn instinctively.

There's a different approach to virtual-sphere rotation developed by Shoemake [Sho92], in which a click and drag from P to Q rotates the sphere from P toward Q, but by *double* the angle used in the virtual sphere. A click at the center of the virtual arcball followed by a drag to the edge of the arc ball produces not a 90° rotation, but a 180° rotation. This has the advantage that one can achieve any desired rotation of the object by a single click and drag (e.g., spins about the view direction are generated by dragging from one point near the boundary of the ball to another).

11.7 Discussion and Further Reading

For the mathematically inclined, the study of $\mathbf{SO}(n)$ is covered in several books [Che46, Hus93, Ste99], and some of the basic properties of $\mathbf{S^n}$ and $\mathbf{SO}(n)$ are discussed in many introductory books on manifolds [GP10, Spi79a].

The classic work on quaternions is by Hamilton [Ham53], but more modern expositions [Che46, Hus93] are much easier to read.

Quaternions are an instance of a more general phenomenon developed by Grassmann [Gra47] in which noncommutative forms of multiplication played a central role. Unfortunately, Grassmann's ideas were so confusingly expressed that they were largely ignored by his contemporaries. There has been some renewed

interest in them in physics (and some related interest in graphics), with recent developments being given the name **geometric algebra** [HS84, DFM07].

11.8 Exercises

Exercise 11.1: We computed the matrix for reflection through the line determined by the unit vector $\mathbf{u}$ by reasoning about dot products. But this reflection is also identical to rotation about $\mathbf{u}$ by 180°. Use the axis-angle formula for rotation to derive the reflection matrix directly.

Exercise 11.2: In $\mathbf{R}^n$, what is the matrix for reflection through the subspace spanned by $\mathbf{e}_1, \ldots, \mathbf{e}_k$, the first k standard basis vectors? In terms of k, tell whether this reflection is orientation-*preserving* or *reversing*.

Exercise 11.3: Write down the matrices for rotation by 90° in the xy-plane and rotation by 90° in the yz-plane. Calling these $\mathbf{M}$ and $\mathbf{K}$, verify that $\mathbf{MK} \neq \mathbf{KM}$, thus establishing that in general, if $\mathbf{R}_1$ and $\mathbf{R}_2$ are elements of $\mathbf{SO}(3)$, it's not true that $\mathbf{R}_1\mathbf{R}_2 = \mathbf{R}_2\mathbf{R}_1$; this is in sharp contrast to the set $\mathbf{SO}(2)$ of 2×2 rotation matrices, in which any two rotations commute.

Exercise 11.4: In Listing 11.2, we have a condition "if θ is near π" which handles the special case of very-large-angle rotations by picking a nonzero column $\mathbf{v}$ of $\mathbf{M} + \mathbf{I}$ as the axis. If θ is not exactly π, then $\mathbf{v}$ will not quite be parallel to an axis. Explain why $\mathbf{v} + \mathbf{Mv}$ will be much more nearly parallel to the axis. Adjust the code in Listing 11.2 to apply this idea repeatedly to produce a very good approximation of the axis.

Exercise 11.5: Consider the parameterization of rotations by Euler angles, with $\theta = \pi/2$. Show that simultaneously increasing ϕ and decreasing ψ by the same amount results in no change in the rotation matrix at all.

Exercise 11.6: The second displayed matrix in Equation 11.23 is the square of the first ($\mathbf{J}_\omega$); it's also symmetric. This is not a coincidence. Show that the square of a skew-symmetric matrix is always symmetric.

Exercise 11.7: Find the eigenvalues and all real eigenvectors for $\mathbf{J}_\omega$. Do the same for $\mathbf{J}_\omega^2$.

Exercise 11.8: Suppose that $\mathbf{A}$ is a rotation matrix in $\mathbf{R}^3$.

(a) How many eigenvalues does a 3×3 matrix have?

(b) Show that the only real eigenvalue that a rotation matrix can have is ± 1. Hint: A rotation preserves length.

(c) Recall that for a real matrix, nonreal eigenvalues come in pairs: If z is an eigenvalue, so is $\bar{z}$. Use this to conclude that $\mathbf{A}$ must have either one or three real eigenvalues.

(d) Use the fact that if z is a nonzero complex number, then $z\bar{z} > 0$, and the fact that the determinant is the product of the eigenvalues to show that if $\mathbf{A}$ has a nonreal eigenvalue, it also has a real eigenvalue which much be 1, and that if $\mathbf{A}$ has only real eigenvalues, at least one of them must be 1.

(e) Conclude that since 1 is always an eigenvalue of $\mathbf{A}$, there's always a nonzero vector $\mathbf{v}$ with $\mathbf{Av} = \mathbf{v}$, that is, the rotation $\mathbf{A}$ has an axis.

Exercise 11.9: The skew-symmetric matrix $\mathbf{J}_\omega$ associated to a vector ω is the matrix for the linear transformation $\mathbf{v} \mapsto \omega \times \mathbf{v}$.

(a) Show that every 3×3 skew-symmetric matrix $\mathbf{S}$ represents the cross product with some vector ω, that is, describe an inverse to the mapping $\omega \mapsto \mathbf{J}_\omega$.

◈ (b) Now use this to explain why every 3×3 skew-symmetric matrix has 0 as an eigenvalue, hence $\det \mathbf{S} = 0$.

Exercise 11.10: In the description of the virtual-sphere controller, we used the angle $\theta = \cos^{-1}(P \cdot Q)$, which involves a dot product of *points* rather than vectors. This worked only because we assumed that the center of our virtual sphere was the origin.
(a) Suppose the center was some other point B; how would we have computed θ?
(b) Now suppose that the virtual sphere was no longer assumed to be a unit sphere. How would we compute θ?

Exercise 11.11: Use the 3D test bed to implement virtual-sphere control for viewing.

Exercise 11.12: Given a point P and a direction $\mathbf{v}$, describe how to build a transformation on $\mathbf{R^3}$ that rotates by θ about the line determined by P and $\mathbf{v}$ so that the plane through P, orthogonal to $\mathbf{v}$, rotates counterclockwise by θ when viewed from the point $P + \mathbf{v}$.

Exercise 11.13: We saw in Chapter 7 that if f is the test function for the plane containing the point Q and with normal $\mathbf{n}$, that is, $f(P) = (P - Q) \cdot \mathbf{n}$, then we could compute the intersection of the ray $t \mapsto A + t\mathbf{w}$ with this plane by writing $g(t) = A + t\mathbf{w}$ and solving $f(g(t)) = 0$. Suppose that T is a linear transformation with 4×4 matrix $\mathbf{M}$—perhaps translation by some amount, or a rotation about the y-axis by $30°$. We can imagine applying T to every point of the plane defined by f to get a *new* plane, and then finding the intersection of our ray with the new plane.
(a) George proposes that a test function $\bar{f}$ for the new plane can be defined by $\bar{f}(P) = f(T(P))$. Is he correct? If not, adjust his claim to a correct one.
(b) Show that $\bar{f}(g(t)) = 0$ if and only if $f(\bar{g}(t)) = 0$, where $\bar{g}(t) = \mathbf{M}^{-1}A + t\mathbf{M}^{-1}\mathbf{w}$.
(c) Describe how you can use the idea of part (b) to compute a ray-object intersection between a ray and an object that has been transformed by some affine transformation, assuming you know how to do ray-object intersection for some standard form of the object. This allows you, for instance, to ray-trace a stretched, rotated unit sphere (i.e., an ellipsoid) if you know only how to ray-trace the standard unit sphere.
◈ (d) Often in ray tracing it's necessary to determine not only the intersection point, but also the normal vector at the intersection point. Suppose that instead of intersecting a ray R with a transformed sphere, you intersect an inverse-transformed version of R with the unit sphere at the origin, and find the intersection is at a point $P = (x, y, z)$ with normal vector $\mathbf{n} = \begin{bmatrix} x & y & z \end{bmatrix}^{\mathbf{T}}$. How can you find the normal vector to the transformed sphere? Note: This approach to ray tracing is somewhat out of favor for complex scenes, as traversing the modeling hierarchy for every ray is a big cost. Instead, flattening the hierarchy by converting everything to triangles and using a spatial data structure to accelerate intersection testing turns out to be faster in general, but not always: In a forest of 1 million identical trees, each a copy of a single million-triangle tree, the spatial data structure approach fails. Instead, we make a spatial data structure whose leaves are *bounding boxes* for the individual trees, use the transforming-rays trick of this exercise, and proceed to trace each ray in the archetype tree's modeling space, possibly using a spatial data structure there to accelerate the computation. This is another application of the Coordinate-System/Basis principle.

This page intentionally left blank

Chapter 12

A 2D and 3D Transformation Library for Graphics

12.1 Introduction

The ideas of the previous chapters can be nicely condensed into an implementation—a collection of cooperating classes that help to maintain the point/vector distinction, the distinction between a transformation T that acts on points and the associated transformations of vectors and covectors, and some of the other routine computations that are often done in graphics.

This chapter can be regarded as an instance of the Implementation principle: that if you understand a mathematical idea well enough, you can implement it in code, and thereafter be insulated from the need for further understanding.

The book's website has such an implementation in C#, starting with the predefined `Point`, `Vector`, `Point3D`, and `Vector3D` WPF classes that you've already seen. You should download the implementation and look at it as you read this chapter.

The implementation depends on a matrix library—one capable of inverting matrices, solving linear systems, multiplying matrices, etc. We've chosen to import the `MathNet.Numerics.LinearAlgebra` library [Mat], but if you prefer another one, it should be easy to substitute, as our use of the library is highly localized.

In most of the classes there are procedures that can fail under certain circumstances. For instance, if you ask for a linear transformation that sends $\mathbf{v}_1$ to $\mathbf{w}_1 \neq \mathbf{0}$, and *also* sends $\mathbf{v}_1$ to $2\mathbf{w}_1$, there is no satisfactory answer. All such failures amount to some matrix being noninvertible. We raise exceptions in these situations. They are discussed in the code and its documentation, but not in this chapter.

The approach taken in our implementation is not the only one possible. Our approach depends on *transformations* as the primitive notion, but it's quite possible and reasonable to think instead of *coordinate systems* as the fundamental entity. Just as in Chapter 10 we discussed the interpretation of a linear transformation as a change of coordinates, you can approach much of graphics with this point of view. You end up having coordinate frames for vector spaces (for a 2D space, you have two independent vectors; for a 3D space, you have three independent vectors), for affine spaces (typically a coordinate frame for a 2D affine space consists of three points, from which you determine barycentric coordinates, but you can also build a frame from one point and two vectors), and for projective spaces (where in 2D, a projective frame is represented by four points in "general position," which we'll discuss shortly). This coordinate frame approach is taken by Mann et al. [MLD97].

12.2 Points and Vectors

The predefined `Point` and `Vector` classes in WPF already implement the main ideas we've discussed (for two dimensions): There are operators defined so that you can add one `Point` and one `Vector` to get a new `Point`, but there is no operator for adding two `Point`s, for instance. Certain convenience operations have been included, like the dot product of `Vector`s.

There are idiosyncrasies in the design of the classes, however. The two coordinates of a point `P` are `P.X` and `P.Y`; there's no way to refer to them as elements of an array of length 2, nor even a predefined "cast" operation to convert to a `double[2]`. There is, however, a predefined `CrossProduct` operation for `Vector`s, which treats the vectors as lying in the *xy*-plane of 3-space, computes the 3D cross product (which always points along the *z*-axis), and returns the *z*-component of the resultant vector. In deference to the convenience of having data types that work well with the remainder of WPF, we've ignored these idiosyncrasies and simply used the parts of the `Point` and `Vector` classes (and their 3D analogs) that we like. We've also added some geometric functions to our `LIN_ALG` namespace (in which all the transformation classes reside) to compute things like the two-dimensional cross product of one vector.

12.3 Transformations

While WPF also has a class called `Matrix`, its peculiarities made it unsuitable for our use. Furthermore, we wanted to build a library in which the fundamental idea was that of a *transformation* rather than the matrix used to represent it. We therefore define four classes.

- `MatrixTransformation2`: A parent class for linear, affine, and projective transformations. Since all three can be represented by 3×3 matrices, a `MatrixTransformation` holds a 3×3 matrix and provides certain support procedures to multiply and invert such matrices.
- `LinearTransformation2`: A transformation that takes `Vector`s to `Vector`s.
- `AffineTransformation2`: A transformation that can operate on both `Point`s and `Vector`s.

- `ProjectiveTransformation2`: A transformation that operates on `Point`s in their homogeneous representation and includes a division by the last coordinate after the matrix multiplication.

(There are four corresponding classes for transformations in 3D.)

In each case, we've defined composition of transformations by overloading the $\star$ operator, and the application of a transformation to a `Point` or `Vector` by overloading the $\star$ operator again. To translate a point and then rotate it by $\pi/6$, we could write

```
1 Point P = new Point(...);
2 AffineTransformation2 T = AffineTransformation2.Translate(Vector(3,1));
3 AffineTransformation2 S = AffineTransformation2.RotateXY(Math.PI/6);
4 Point Q = (S * T) * P;
```

If we were planning to operate on many points with this composed transformation, we'd precompute the composed transformation and instead write

```
1 ...
2 AffineTransformation2 T2 = (S * T);
3 Point Q = T2 * P;
```

12.3.1 Efficiency

Applying a transformation to a point or vector involves some memory allocation, a method invocation, and a matrix multiplication. If you simply stored the matrix yourself, you could avoid most of this cost. And since graphics programs end up applying *lots* of transformations to *lots* of points and vectors, you might think that doing it yourself is the best possible approach. If you're writing a program that will be doing real-time graphics on a processor where computation is a real bottleneck (e.g., a game that runs on a battery-powered device), it may well be. But as a student of computer graphics, you're likely to write a lot of programs that get run just a few times. The great "cost" in your programs is *your* time as a developer. Using a high-level approach can help reduce bugs and even *increase* efficiency, as you notice ways to restructure your code that would be difficult to see if you were looking at every detail all the time. A profiler can help you to determine exactly *where* in your code it's worth the trouble of working with a low-level construct rather than a high-level one.

Nonetheless, there are places where one can get a certain amount of efficiency at no cost. For instance, the `LinearTransformation2` class uses a 3×3 matrix to represent a transformation, but that matrix always has the form

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{12.1}$$

so it's much easier to invert than a general 3×3 matrix (we just invert the upper-left 2×2 matrix); in the same way, multiplying two of these is much less work than multiplying two 3×3 matrices (we just multiply the upper-left 2×2 matrices). By overriding the `MatrixTransform2` methods for inversion and multiplication, we get a large efficiency improvement.

Inline Exercise 12.1: Without looking at the code, consider whether you can find a more efficient way to invert an affine transformation on $\mathbf{R}^2$ than by inverting its 3×3 matrix. Hint: The bottom row of the matrix is always $\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$.

12.4 Specification of Transformations

For each kind of transformation, the default constructor generates the identity transformation. (The constructor for `MatrixTransformation2` is `protected`, as only the derived classes are supposed to ever create a `MatrixTransformation`.) In general, though, transformations are constructed by static methods with mnemonic names. For the `AffineTransformation2` class, for instance, there are eight static methods (all `public static AffineTransform2`) that construct transformations.

```
1  RotateXY(double angle)
2  Translate(Vector v)
3  Translate(Point p, Point q)
4  AxisScale(double x_amount, double y_amount)
5  RotateAboutPoint(Point p, double angle)
6  PointsToPoints(Point p1, Point p2, Point p3,Point q1, Point q2, Point q3)
7  PointAndVectorsToPointAndVectors(Point p1, Vector v1, Vector v2,
8                                   Point q1, Vector w1, Vector w2)
9  PointsAndVectorToPointsAndVector(Point p1, Point p2, Vector v1,
10                                  Point q1, Point q2, Vector w1)
```

The naming convention is straightforward: "From" comes before "to" so that

```
Translate(Point p, Point q)
```

creates a translation that sends `p` to `q`, and within a collection of arguments, points come before vectors so that in

```
PointAndVectorsToPointAndVectors
```

the point `p1` is sent to the point `q1`, the vector `v1` is sent to the vector `w1`, and the vector `v2` is sent to the vector `w2`. The method name tells you that there is one point and more than one vector; since an affine transformation of the plane is determined by its values on three points, or one point and two vectors, or two points and one vector, you know that the arguments must be one point and two vectors.

Methods that produce particular familiar transformations—translations, rotations, axis-aligned scales—have names indicating these. While the names are sometimes cumbersome, they are also expressive; it's easy to understand code that uses them.

12.5 Implementation

Most of the transformations are easy to implement. For instance, we first implemented the

```
PointAndVectorsToPointAndVectors
```

method for `AffineTransformation2`; once we'd done this, the `PointsToPoints` method was straightforward:

```
1  public static AffineTransform2 PointsToPoints(
2    Point p1, Point p2, Point p3,
3    Point q1, Point q2, Point q3)
4  {
5    Vector v1 = p2 - p1;
6    Vector v2 = p3 - p1;
7    Vector w1 = q2 - q1;
8    Vector w2 = q3 - q1;
9    return AffineTransform2.PointAndVectorsToPointAndVectors(p1, v1, v2, q1, w1, w2);
10 }
```

The `PointAndVectorsToPointAndVectors` code is implemented in a relatively straightforward way: We know that the vectors `v1` and `v2` must be sent to the vectors `w1` and `w2`; in terms of the 3×3 matrix, that means that the upper-left corner must be the 2×2 matrix that effects this transformation of 2-space. So we invoke the `LinearTransformation2` method `VectorsToVectors` to produce the transformation. Unfortunately, the resultant transformation does not send `p1` to `q1` in general. To ensure this, we precede this vector transform with a translation that takes `p1` to the origin (the translation has no effect on vectors, of course); we then perform the linear transformation; we then follow this with a transformation that takes the origin to `q1`. The net result is that `p1` is sent to `q1`, and the vectors are transformed as required.

This approach relies on the `VectorsToVectors` method for `LinearTransformation2`. Writing that is straightforward: We place the vectors `v1` and `v2` in the first two columns of a 3×3 matrix, with a 1 in the lower right. This transformation T sends $\mathbf{e}_1$ to $\mathbf{v}_1$ and $\mathbf{e}_2$ to $\mathbf{v}_2$. Similarly, we can use the $\mathbf{w}$'s to build a transformation S that sends $\mathbf{e}_1$ to $\mathbf{w}_1$ and $\mathbf{e}_2$ to $\mathbf{w}_2$. The composition $T \circ S^{-1}$ sends $\mathbf{v}_1$ to $\mathbf{e}_1$ to $\mathbf{w}_1$, and similarly for $\mathbf{v}_2$, and hence solves our problem.

12.5.1 Projective Transformations

The only really subtle implementation problem is the `PointsToPoints` method for `ProjectiveTransformation2`. Explaining this code requires a bit of mathematics, but it's all mathematics that we've seen before in various forms.

We're given four points P_1, P_2, P_3, and P_4 in the Euclidean plane, and we are to find a projective transformation that sends them to the four points Q_1, Q_2, Q_3, and Q_4 of the Euclidean plane.

Before we go any further, we should mention a limitation. When we described the `VectorsToVectors` method of `LinearTransformation2`, we promised to send $\mathbf{v}_1$ and $\mathbf{v}_2$ to $\mathbf{w}_1$ and $\mathbf{w}_2$, but there was, in fact, a constraint. If $\mathbf{v}_1 = \mathbf{0}$ and $\mathbf{w}_1 \neq \mathbf{0}$, there's no linear transformation that accomplishes this. In fact, if $\mathbf{v}_1$ is a multiple of $\mathbf{v}_2$, in general there's no linear transformation that solves the problem (except in the very special case where $\mathbf{w}_1$ is the same multiple of $\mathbf{w}_2$, in which case there are an infinite number of solutions). The implicit constraint was that $\mathbf{v}_1$ and $\mathbf{v}_2$ must be *linearly independent* for our solution to work (or for the general problem to have a solution). In the case of `PointsToPoints`, we require something similar: The points P_i $(i = 1, \ldots, 4)$ must be in **general position,** which means that (a) no

two of them can be the same, and (b) none of them can lie on the line determined by two others (see Figure 12.1). In more familiar terms, this is equivalent to saying (a) that P_1, P_2, and P_3 form a nondegenerate triangle, and (b) that the barycentric coordinates of P_4 with respect to P_1, P_2, and P_3 are all nonzero. We'll further require that the Q_is are similarly in general position.[1]

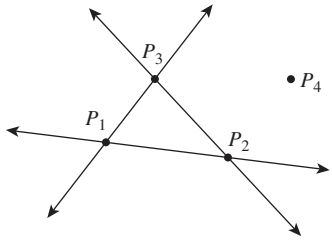


Figure 12.1: The four points are in general position, because (a) none of them lies on a line passing through another pair or (b) the first three form a nondegenerate triangle, and the fourth is not on the extensions of any of the sides of this triangle. These two descriptions are equivalent.

Returning to the main problem of sending the Ps to the Qs, when we express the points P_i and Q_i as elements of 3-space, we append a 1 to each of them to make it a vector whose tip lies in the $w = 1$ plane. We'll call these vectors $\mathbf{p}_1, \mathbf{p}_2$, etc. Our problem can then be expressed by saying that we seek a 3×3 matrix $\mathbf{M}$ with the property that

$$\mathbf{M}\mathbf{p}_1 = \alpha\mathbf{q}_1 \tag{12.2}$$
$$\mathbf{M}\mathbf{p}_2 = \beta\mathbf{q}_2 \tag{12.3}$$
$$\mathbf{M}\mathbf{p}_3 = \gamma\mathbf{q}_3 \tag{12.4}$$
$$\mathbf{M}\mathbf{p}_4 = \delta\mathbf{q}_4 \tag{12.5}$$

for some four nonzero numbers α, β, γ, and δ (because, for instance, $\alpha\mathbf{q}_1$, when we divide through by the last coordinate, will become $\mathbf{q}_1$). The problem is that we do not know the values of the multipliers.

This problem, as stated, is too messy. If there were no multipliers, we'd be looking for a 3×3 matrix with $\mathbf{M}\mathbf{p}_i = \mathbf{q}_i$ $(i = 1, \ldots, 4)$. We can only solve such problems for *three* vectors at a time, not four. Thus, the multipliers are essential—without them, there'd be no solution at all in general. But they also complicate matters: We're looking for the nine entries of the matrix, and the four multipliers, which makes 13 unknowns. But we have four equations, each of which has three components, so we have 12 equations and 13 unknowns, a large underdetermined system. It's easy to see why the system is underdetermined, though: If we found a solution $(\mathbf{M}, \alpha, \beta, \gamma, \delta)$, then we could double everything and get another equally good solution $(2\mathbf{M}, 2\alpha, 2\beta, 2\gamma, 2\delta)$ to Equations 12.2–12.5.

Inline Exercise 12.2: Verify this claim.

So the first step is to make the solution unique by declaring that we're looking for a solution with $\delta = 1$. This gives 13 equations in 13 unknowns. We *could* just solve this linear system. But there's a simpler approach that involves much less computation in this 3×3 case, and even greater savings in the 4×4 case.

Inline Exercise 12.3: Show that fixing $\delta = 1$ is reasonable by showing that if there is *any* solution to Equations 12.2–12.5, then there is a solution with $\delta = 1$. In particular, explain why any solution must have $\delta \neq 0$.

We'll follow the pattern established in Chapter 10 in simplifying the problem. To send the $\mathbf{p}$'s to the $\mathbf{q}$'s, we'll instead find a way to send four standard vectors to the $\mathbf{q}$'s, and the same four vectors to the $\mathbf{p}$'s, and then compose one of these transformations with the inverse of the other. The four standard vectors we'll use are $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$, and $\mathbf{u} = \mathbf{e}_1 + \mathbf{e}_2 + \mathbf{e}_3$. We'll start by finding a transformation that sends these to multiples of $\mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3$, and $\mathbf{q}_4$, respectively.

1. This latter condition is overly stringent, but it simplifies the analysis somewhat.

Step 1: The matrix whose columns are $\mathbf{q}_1, \mathbf{q}_2$, and $\mathbf{q}_3$ sends $\mathbf{e}_1$ to $\mathbf{q}_1$, $\mathbf{e}_2$ to $\mathbf{q}_2$, and $\mathbf{e}_3$ to $\mathbf{q}_3$. Unfortunately, it does not necessarily send $\mathbf{u}$ to $\mathbf{q}_4$; instead, it sends $\mathbf{u} = \mathbf{e}_1 + \mathbf{e}_2 + \mathbf{e}_3$ to $\mathbf{q}_1 + \mathbf{q}_2 + \mathbf{q}_3$, that is, the sum of the columns. If we scale up each column by a different factor, the resultant matrix will still send $\mathbf{e}_i$ to a multiple of $\mathbf{q}_i$ for $i = 1, 2, 3$, but it will send $\mathbf{u}$ to a sum of *multiples* of the $\mathbf{q}$'s. We therefore, as a first step, write $\mathbf{q}_4$ as a linear combination of $\mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3$,

$$\mathbf{q}_4 = \alpha\mathbf{q}_1 + \beta\mathbf{q}_2 + \gamma\mathbf{q}_3, \tag{12.6}$$

which is exactly the same thing as writing Q_4 in barycentric coordinates with respect to Q_1, Q_2, and Q_3. Note that because of the general position assumption, α, β, and γ are all nonzero.

Inline Exercise 12.4: Explain this last statement.

In terms of code, to find α, β, and γ we build a matrix[2] $\mathbf{Q} = \begin{bmatrix}\mathbf{q}_1; & \mathbf{q}_2; & \mathbf{q}_3\end{bmatrix}$ and let

$$\begin{bmatrix}\alpha \\ \beta \\ \gamma\end{bmatrix} = \mathbf{Q}^{-1}\mathbf{q}_4. \tag{12.7}$$

Inline Exercise 12.5: Explain why the solution to Equation 12.7 is in fact a solution to Equation 12.6.

Now consider the matrix

$$\mathbf{A} = [\alpha\mathbf{q}_1; \beta\mathbf{q}_2; \gamma\mathbf{q}_3]. \tag{12.8}$$

It's straightforward to verify that it sends $\mathbf{e}_i$ to a multiple of $\mathbf{q}_i$ for $i = 1, 2, 3$, and it sends $\mathbf{u}$ to the sum of its columns, which is, by Equation 12.7, exactly $\mathbf{q}_4$.

Step 2: Repeating this process, we can find a matrix transformation with matrix $\mathbf{B}$ sending $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \mathbf{u}$ to multiples of $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4$.

Step 3: The matrix $\mathbf{AB}^{-1}$ then sends the $\mathbf{p}$'s to multiples of the corresponding $\mathbf{q}$'s.

Note that in solving this problem we solved a 3×3 system of equations and inverted a 3×3 matrix—which required far less computation than solving a 13×13 system of equations.

Inline Exercise 12.6: Explain why the matrix $\mathbf{B} = \begin{bmatrix}\alpha'\mathbf{p}_1 & \beta'\mathbf{p}_2 & \gamma'\mathbf{p}_3\end{bmatrix}$, where α', β', γ' are the barycentric coordinates of P_4 with respect to P_1, P_2, and P_3, is invertible. Hint: Write $\mathbf{B} = \mathbf{PS}$, where $\mathbf{S}$ is diagonal and $\mathbf{P}$ has $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$ as columns. Now apply the general-position assumption about the points $P_i (i = 1, \ldots, 4)$.

12.6 Three Dimensions

The 3D portion of the library is completely analogous to the 2D one, except that rotations are somewhat more complicated. To implement rotation about an arbitrary vector, we use Rodrigues' formula; to implement rotation about an arbitrary

2. Recall that semicolons indicate that the items listed are the *columns* of the matrix.

ray (specified by a point and direction), we translate the point to the origin, rotate about the vector, and translate back. The projective `PointsToPoints` method uses the same general approach that we used in two dimensions, replacing the solution of 21 simultaneous equations with a solution of four simultaneous equations and a 4×4 matrix inversion.

12.7 Associated Transformations

When we have an affine transformation T on Euclidean space, we've said that we can transform either points or vectors, and we've incorporated this into our code. For an affine transformation, we've also seen how to transform covectors; in the code, we've defined a `Covector` structure (which is very similar to the `Vector` structure in the sense of storing an array of `double`s). And for affine transformations, there's an associated transformation, `T.NormalMap`, of covectors. We've bowed to convention here in calling this the "normal map" rather than the "covector map," since its use in graphics is almost entirely restricted to normal (co)vectors to triangles.

For a projective transformation, the associated map on vectors typically varies from point to point. In Figure 10.24, for instance, the top and bottom edges of the small square in the domain may be regarded as two identical vectors; after transformation, these vectors are no longer parallel. The associated map of vectors transformed them differently because they were based at different points. Thus, the associated vector transformation takes both a point and a vector as arguments. The details, and detailed rationale, are presented in a web addendum. The covector transform (for a projective transformation) similarly depends on the point of application.

One consequence of the point dependence for the vector and covector transformations of a projective transformation is that many operations—particularly those involving dot products of vectors, like computing how much light reflects from a surface—are best performed *before* the projective transformation near the end of the rendering pipeline.

12.8 Other Structures

Depending on how you plan to use the linear algebra library, it might make sense to create classes to represent other common geometric entities like rays, lines, planes in 3-space, ellipses, and ellipsoids (which are transformed to ellipses and ellipsoids, respectively, by nondegenerate linear and affine maps). A ray, for instance, might be represented by a `Point` and a direction `Vector`. It's then natural either to define

```
public static Ray operator*(AffineTransformation2 T, Ray r)
```

or to include in the `AffineTransformation2` class a method like

```
public Ray RayTransform(Ray r)
```

A good implementation would transform the ray's `Point` by T, and would transform its direction `Vector` and normalize the result, because many computations on rays are easier when their directions are expressed as unit vectors.

12.9 Other Approaches

We mentioned that there are other approaches to encapsulating the various transformation operations needed in graphics, including basing them on coordinate frames rather than linear transformations.

There's a particularly efficient way to create a *restricted* transformation library in the case that we only want to describe *rigid* transformations of objects. This eliminates all scaling—both uniform and nonuniform—and all nonaffine projective transformations. Thus, every transformation is simply a translation, rotation, or combination of the two. There are two advantages of considering only such transformations.

- There are no "degeneracies." In the case of the `PointsToPoints` transformations we discussed earlier, there was a possibility of failure if the starting points were not in general position. No such problem arises here.
- When we need to invert such a rigid transformation, no matrix inversion procedure is needed, because the inverse of a rotation matrix $\mathbf{A}$ is its transpose $\mathbf{A}^{\mathbf{T}}$.

There are disadvantages as well.

- We can no longer easily use a `PointsToPoints` specification for a transformation. The pairwise distances of the starting points must exactly match those of the target points, because a rigid transformation preserves distances between points. It's impractical to try to specify target points with this property, even when the source points are $(0,0)$, $(1,0)$, and $(0,1)$, for instance.
- We cannot make larger or smaller instances of objects in a scene using this design. (A typical solution is to provide a method for reading objects from a file *with a scale factor* so that you create a large sphere by reading a standard sphere with a scale factor of 6.0, for instance.)

G3D, a package we'll use in Chapter 32 for the implementation of two renderers, uses the rigid-motion approach. It contains a class `CFrame` (for "coordinate frame"); the standard instance of this is the standard coordinate frame based at the origin. The model for the scene in Figure 12.2 involves quite a lot of code, most of which describes material properties, etc. The essence of the *geometric* part of the modeling[3] is given in Listing 12.1, from which we've removed all modeling of light sources and materials.

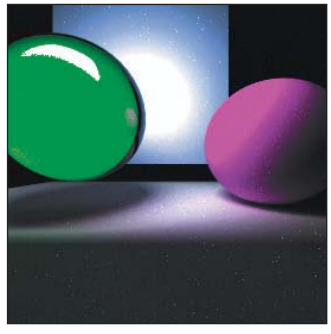


Figure 12.2: A simple scene.

Listing 12.1: Modeling a simple scene.

```
1  void World::loadWorld1() {
2    modeling of lights omitted
3    // A sphere, slightly to right, shiny and red.
4    addSphere(Point3(1.00f, 1.0f, -3.0f), 1.0f, material specification );
5    // LEFT sphere
6    addTransparentSphere(Point3(-0.95f, 0.7f, -3.0f), 0.7f, material specifications );
7
8    // And a ground plane...
9    addSquare(4.0, Point3(0.0f, -0.2f, -2.0f),
```

3. The camera is specified elsewhere in the program.

```
    Vector3(1.0f, 0.0f, 0.0f), Vector3(0.0f, 1.0f, 0.0f), material specifications );

  // And a back plane...
  addSquare(4.0, Point3(0.0f, 2.0f, -4.00f),
    Vector3(1.0f, 0.0f, 0.0f), Vector3(0.0f, 0.0f, 1.0f), material specifications );
   ...
}
```

A sphere is specified by its center point and radius; a square by its edge length, center point, a vector aligned with one axis of the square, and the normal vector to the square. These two vectors, as specified, must be perpendicular, or the code will fail. The code for adding a sphere or square to the scene is given in Listings 12.2 and 12.3.

Listing 12.2: The shape-adding methods.

```
void World::addTransparentSphere(const Point3& center, float radius,
  material parameters ){
  ArticulatedModel::Ref sphere =
    ArticulatedModel::fromFile(System::findDataFile("sphere.ifs"), radius);
  lots of material specification omitted
  insert(sphere, CFrame::fromXYZYPRDegrees(center.x, center.y, center.z, 0));
}

void World::addSquare(float edgeLength, const Point3& center, const Vector3&
    axisTangent, const Vector3& normal, const Material::Specification& material){
  ArticulatedModel::Ref square = ArticulatedModel::fromFile(
    System::findDataFile("squarex8.ifs"), edgeLength);

  material specification code omitted

  Vector3 uNormal = normal / normal.length();
  Vector3 firstTangent(axisTangent / axisTangent.length());
  Vector3 secondTangent(uNormal.cross(firstTangent));

  Matrix3 rotmat(
    firstTangent.x, secondTangent.x, uNormal.x,
    firstTangent.y, secondTangent.y, uNormal.y,
    firstTangent.z, secondTangent.z, uNormal.z);

  CoordinateFrame cFrame(rotmat, center);
  insert(square, cFrame);
}
```

Listing 12.3: The methods for inserting a shape into the scene.

```
void World::insert(const ArticulatedModel::Ref& model, const CFrame& frame) {
    Array<Surface::Ref> posed;
    model->pose(posed, frame);
    for (int i = 0; i < posed.size(); ++i) {
        insert(posed[i]);
        m_surfaceArray.append(posed[i]);
        Tri::getTris(posed[i], m_triArray, CFrame());
    }
}
```

As you can see, the sphere-adding code reads the model from a file and accepts a scaling parameter that says how much to scale the model up or down. The

returned object represents a shape; its coordinates are assumed to be in the standard coordinate frame. This shape is then added to our world (represented by the object `m_surfaceArray`) by specifying a coordinate frame using

```
CFrame::fromXYZYPRDegrees(center.x, center.y, center.z);
```

This method specifies a coordinate frame by saying how much to translate the standard frame in x, y, and z (the "XYZ" part of the name), and then how much yaw, pitch, and roll (the "YPR") to apply, with the amounts specified in degrees. The default values for yaw, pitch, and roll are 0, so we didn't specify them.

In the `insert` method, the model is "posed" in this new frame, that is, its coordinates are now taken to be relative to that new frame. Since that new frame is based at the specified `center`, we end up with a sphere translated to the specified location. This transformed surface is added to the world-description member variable `m_surfaceArray`, while its representation as a collection of triangles is added to another member variable, `m_triArray`, which is used in visibility testing.

The square-insertion code is slightly more complex. The standard unit square is read from a file, scaled up by the edge length. Since the standard square is centered at the origin, with edge length 1, and is aligned with the axes of the xy-plane, the x and y unit vectors are tangent to the square and the z unit vector is normal to it. We'll need to build a new coordinate frame whose first axis is the specified `axisTangent` and whose third axis is the specified `normal`. To do so, we build a matrix transformation that sends the x-, y-, and z-axes to the `axisTangent`, a second tangent, and the `normal`, respectively, although we assist the user slightly by not requiring that either specified vector be unit length. To build the new coordinate frame, we build a matrix that transforms the standard frame to the desired new one, and then construct the new frame using `cFrame(rotmat, center);`, one of the `CoordinateFrame` class's standard constructors.

12.10 Discussion

Which kind of linear algebra support you choose will depend on both personal taste and the kinds of programming you're doing. If peak efficiency in matrix operations is essential, you may choose to work directly with arrays of floats. If expressive convenience matters to you, you may choose the style we first described, with methods like `PointsToPoints`. If you'll frequently be inverting transformations, perhaps the G3D approach will work best for you.

In general, however, your programs will be easier to write and debug if you rely on a few carefully written and tested programs for building and applying transformations, and use your language's type system to help you keep track of the difference between points and vectors. The more your linear algebra module can support the expression of *what* rather than *how* (e.g., "I want the camera to look in *this* direction" rather than "I want to rotate the camera 37° in x, then 12. 3° in y") the easier it will be to both understand and maintain your programs.

12.11 Exercises

Exercise 12.1: Create a `Ray` class to represent a ray in the plane, and build the associated ray transformation in the `AffineTransformation2` class. Do the same for a

`Line`. What constructors should the `Line` class have? What about a `Segment` class? What methods should `Segment` have that `Line` lacks? Can you develop a way to make rays, lines, and segments cooperate with the `ProjectiveTransformation` class, or are there insurmountable problems? Think about what happens when a ray crosses the line on which a projective transformation is undefined.

Exercise 12.2: General position of the points $P_i (i = 1, \ldots, 4)$ was needed to invert the matrix **B** in the construction of the `PointsToPoints` method for projective maps. We also assumed that the points $Q_i (i = 1, \ldots, 4)$ were in general position, but that assumption was stronger than necessary. What is the weakest geometric condition on the Q_i that allows the `PointsToPoints` transformation to be built?

Exercise 12.3: Explain why the two characterizations of general position for four points in the plane—that (a) no point lies on a line passing through another pair and (b) the first three form a nondegenerate triangle, while the fourth is not on the extensions of any of the sides of this triangle—are equivalent. Pay particular attention to the failure cases, that is, show that if four points fail to satisfy condition (a), they also fail to satisfy condition (b), and vice versa.

Exercise 12.4: Enhance the library by defining one-dimensional transformations as well (`LinearTransformation1`, `AffineTransformation1`, `ProjectiveTransformation1`). The first two classes will be almost trivial. The third is more interesting; include a constructor `ProjectiveTransform1(double p, double q, double r)` that builds a projective map sending 0 to p, 1 to q, and ∞ to r (i.e., $\lim_{x \to \infty} T(x) = r$). From such a constructor it's easy to build a `PointsToPoints` transformation.

Exercise 12.5: Enhance the library we presented by adding a constructor `TransformXYZYPRDegrees(Point3 P, float yaw, float pitch, float roll)` to create a transformation that translates the origin to the point P, and applies the specified yaw, pitch, and roll to the standard basis for 3-space.

Exercise 12.6: By hand, find a transformation sending the points $P_1 = (\frac{1}{2}, 1), P_2 = (1, 1), P_3 = (\frac{1}{2}, -1)$, and $P_4 = (1, -1)$ to the points $Q_1 = (\frac{1}{2}, \frac{1}{2})$, $Q_2 = P_2, Q_3 = (\frac{1}{2}, -\frac{1}{2})$, and $Q_4 = P_4$.

Chapter 13

Camera Specifications and Transformations

13.1 Introduction

In this chapter, we briefly discuss camera specifications, which you already encountered in Chapter 6. Recall that we specified a camera in WPF in code like that shown below:

```
1   <PerspectiveCamera
2        Position="57, 247, 41"
3        LookDirection="-0.2, 0, -0.9"
4        UpDirection="0, 1, 0"
5        NearPlaneDistance="0.02" FarPlaneDistance="1000"
6        FieldOfView="45"
7             />
```

From such a specification, we will create a sequence of transformations that will transform the world coordinates of a point on some model to so-called "camera coordinates," and from there to image coordinates. We'll do so by repeatedly using the Transformation Uniqueness principle.

Since an affine coordinate frame in three dimensions consists of four noncoplanar points, this says that if we know where we want to send each of four noncoplanar points, we know that there's exactly one affine transformation that will do it for us. The corresponding theorem for the plane says that if we know where we want to send some *three* noncollinear points, then there's a unique affine transformation that will do it.

We start with an example of this kind of transformation in the plane. Next we discuss basic perspective camera specifications and how we can convert such specifications to a set of affine transformations, plus one projective transformation. We briefly treat the case of "parallel" cameras, and discuss the details of that case and of skewed projections, in this chapter's web materials.

13.2 A 2D Example

While Chapter 10 showed how to build transformations and compose them, it's often easiest to use a higher-level construction to build transformations: Rather than saying *how* to create a transformation as a sequence of elementary transformations, we say *what* we want the transformation to accomplish. Using our linear algebra package (see Chapter 12), we can simply say that we want a linear map that takes certain points to certain others, and let the package find the unique solution to this problem.

Suppose we want to map the square $-1 \leq u, v \leq 1$ (which we'll call the imaging rectangle in anticipation of a later use of such a transformation) to a display with square pixels, 1024 pixels wide, 768 pixels tall. The upper-left pixel is called $(0,0)$; the lower-left is $(0,767)$; and the lower-right is $(1023,767)$. We want to find a transformation T with the property that it sends the square $-1 \leq u, v \leq 1$ to a square region on the left-hand side of the display, one that fills as much of the display as possible. To do so, we need coordinates on the plane of the display. Because pixel coordinates refer to the upper-left corner of each pixel, as shown in Figure 13.1, that is, the center of pixel $(0,0)$ is at $(0.5, 0.5)$, the display coordinates range from $(0,0)$ to $(1024,768)$.

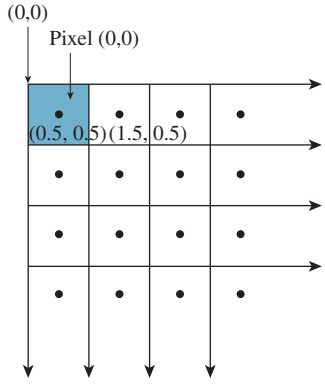


Figure 13.1: The center of the upper-left pixel has coordinates $(0.5, 0.5)$; the pixel is called pixel (0,0). In other words, it is named by its upper-left corner.

This means that we want the point $(-1,-1)$ (the lower-left corner of the imaging rectangle) to be sent to $(0,768)$ (the lower-left corner of the display), and we want $(-1,1)$ (the upper-left corner) to be sent to $(0,0)$. To completely specify an affine transformation on a two-dimensional space, we need to know where *three* independent points are sent. We've already determined where two are sent. For our third point, we choose the lower-right corner: $(1,-1)$ must go to $(768,768)$ (to keep the image square). The code that implements this is

```
1 Transform t =
2   Transform.PointsToPoints(
3     Point2(-1, -1),   Point2(-1, 1),    Point2(1, -1),
4     Point2(0, 768),   Point2(0, 0),     Point2(768, 768));
```

For a specification like this (specifying which points go to which points) we must be certain that the source points constitute a coordinate frame; in two dimensions, this means "noncollinear," which is clearly the case here: Any three corners of a square are noncollinear.

The obvious generalization of this to an arbitrary viewing window of r rows by k columns (without the constraint on squareness) is called the **windowing transformation,** with matrix $\mathbf{M}_{\text{wind}}$. It can be generated by code like

```
1 Transform t =
2   Transform.PointsToPoints(
3     Point2(-1, -1),   Point2(-1, 1),    Point2(1, -1),
4     Point2(0, k),     Point2(0, 0),     Point2(r, k));
```

or, for those who prefer the matrix to be expressed directly, as

$$\mathbf{M}_{\text{wind}} = \begin{bmatrix} r & 0 & 0 \\ 0 & k & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} r & 0 & r \\ 0 & -k & k \end{bmatrix}. \tag{13.1}$$

13.3 Perspective Camera Specification

The WPF camera specification uses six parameters to specify a camera: position (a point), look direction and up direction (two vectors), near-plane distance and far-plane distance (two scalars), and field of view (an angle in degrees).

> Why don't we just render everything we can see? Our field of view is about 180°. Even if you say, "Sure, I can detect stuff in my peripheral vision, but I really *see* stuff within about a 120° view, sort of a cone extending in front of my eyes," you'll find that if you specify a 120° field of view, your pictures will look peculiar and distorted. That's partly because we tend to view pictures so that they occupy a relatively small portion of our field of view. A computer monitor at a comfortable viewing distance may represent only a 25° field of view, for instance, while your cell-phone screen may be just a few degrees wide.
>
> If we *do* make a wide-angle picture and then display it in a way that makes the image occupy a large portion of our field of view, the results can seem less distorted, but there's some evidence that even this doesn't give the viewer a completely satisfactory sense of "seeing everything" [Koe11].
>
> So we compromise and take the approach used by photographers: We render only a modest portion of the eye's field of view.

Figure 13.2 shows these. You can think of the camera being specified as a pinhole camera so that all rays entering the camera do so through a single point, specified as the **position** of the camera. You could also think of this as the center of the camera lens in a more conventional camera. In setting up a real-world photograph, we generally establish several things: the camera position, its orientation, and the field of view (often adjusted with a zoom control on the camera). For fancier cameras, we may also be able to adjust the **focal distance** (the distance to the points that are most in focus in the image), the **depth of field** (how far in front of and behind the focal distance things will be in focus), and even the tilt and offset of the camera lens relative to the camera body. The WPF specification, and that of most basic graphics systems, omits these latter aspects, since an ideal pinhole camera is in focus at all depths, and the pinhole is typically centered over the film or imaging sensor. We'll return to these topics in Section 13.9.

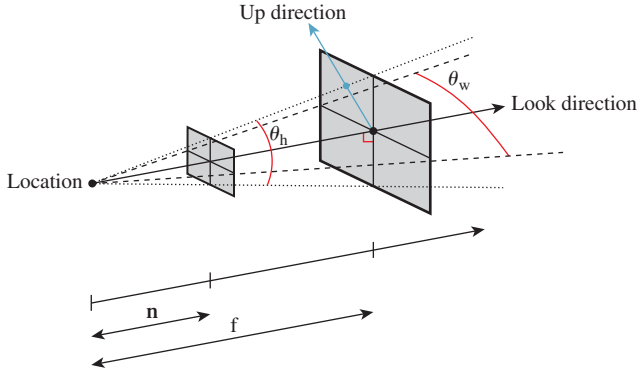


Figure 13.2: The parts of a WPF camera specification.

Returning to the WPF camera model, the look direction specifies in which direction the camera is pointed. If we trace a ray in the scene starting at the position and pointing in the look direction, we'll hit some object. That object will appear in the exact center of the "picture" taken by our camera. The field-of-view angles describe how far from the look direction, in degrees, the camera sees. The basic WPF camera produces a square picture, so the field of view is the same in both the horizontal and vertical directions. In some other systems, you get to specify both a horizontal and vertical field of view (see Exercise 13.1). Others, including WPF, let you specify both the aspect ratio and the horizontal field of view, and compute the vertical field of view for you. In fact, in WPF you specify the aspect ratio indirectly. You specify the width and height of the **viewport** (a rectangle on the display used for exhibiting the image), and the aspect ratio is determined by the ratio of the width to the height. In the remainder of our presentation, we'll specify the view in terms of *both* horizontal and vertical field of view, and we'll discuss the relationship to aspect ratio afterward.

The only subtle point is the specification of the **up direction**, which orients the camera about the look direction, just as you can look out a window with your head tilted either left or right. If we imagine a vertical unit vector **v** painted on the back of the camera body, then this vector, together with the look direction, determines a plane. You might think that we should require the user to specify the vector **v** directly, but that's rather difficult to do in general. Instead, we require the user to specify *any* nonzero vector in the plane (except the look direction), and from this we *compute* the vector **v**. Figure 13.3 shows this: Any of three different vectors can be used as the up direction, from which we can compute the vector **v** that is in the vertical plane of the camera and is perpendicular to the look direction. Often in practice the `UpDirection` is set to `Vector3D(0,1,0)`, that is, the *y*-direction. As long as the camera does not look straight up or down, this is the most natural direction in which to hold it.[1] If the camera *does* look straight up, then the computed vector **v** will be zero; if the camera looks nearly straight up, the computation of **v** from **vup** will involve division by a number that's almost zero, and hence will be numerically unstable.

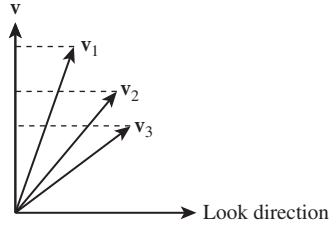


Figure 13.3: The vectors $\mathbf{v}_1, \mathbf{v}_2$, *and* $\mathbf{v}_3$ *all lie in the plane of* `LookDirection` *and* **v**, *and any one of them can be used as the* `UpDirection` *and result in the same view.*

At this point, the view specification tells where the camera is, in which direction it's looking, and how the camera body is rotated around that view direction; the field of view determines how wide an area the camera "sees." We've indirectly described a four-sided **view volume** with a rectangular cross section in space.

Two more parts of the specification remain: the near-plane and far-plane distances shown in Figure 13.4. The near and far planes cut out a *frustum* of the view cone. Objects within this frustum will be displayed in our image, but objects outside will not (see Figure 13.5).

This can be a useful device: By setting the near-plane distance to be almost the distance to the subject, we guarantee that objects between us and the subject do not interfere with the picture. We also avoid ever considering objects that are *behind* the camera, which can be a huge time-saver. And by setting the far-plane distance so that it is not too large, we can similarly save lots of time, by never considering all the objects in the world that are *potentially* within our view but would hardly affect it at all, such as a person standing 30 miles away.

1. In CAD, the horizontal plane is often *xy*, with *z* used for the vertical direction; in that case, of course, the up vector would be `Vector3D(0,0,1)`.

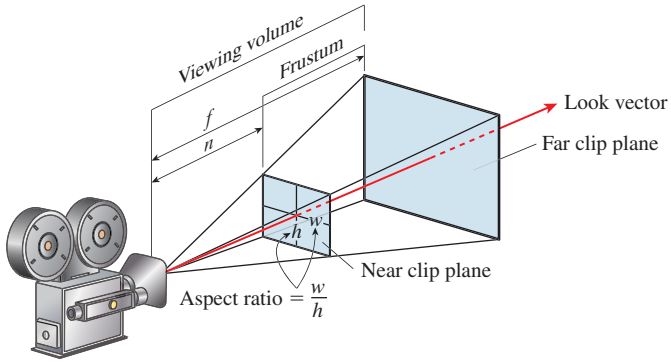


Figure 13.4: The distances to the near and far planes are measured along the look direction.

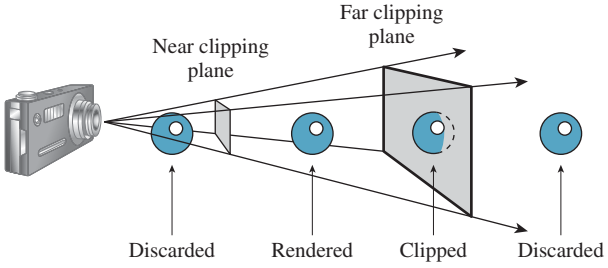


Figure 13.5: Objects outside the view frustum will not be rendered.

It is not only the utility of being able to set the near- and far-plane distances that motivates their use, as you'll see presently; they also allow a rasterizing renderer to avoid certain floating-point-comparison problems that generate errors in images.

Clipping planes have often been used in games to help reduce rendering time by removing distant objects, but sometimes as you move forward in a game, a distant object will "pop" into view, which can be distracting. The general solution to this problem was to render objects in the distance as being obscured by fog so that they appeared gradually as you approached them. In more modern games, we have better rendering systems, and many objects are represented with multiple different levels of detail (see Section 25.4) so that when they are distant, they can be rendered with fewer polygons, making the use of fog less common than it once was.

13.4 Building Transformations from a View Specification

We'll now convert from a view specification to some specific geometry. From a specification, we'll build (a) an orthonormal coordinate system based at the camera position, and (b) several points on the view cone, as shown in Figure 13.6. We'll use these in building the transformations we need. Having a coordinate frame based at the camera is very convenient, since we'll later be transforming the camera to the origin and aligning its coordinate frame with the standard *xyz*-coordinate system at the origin.

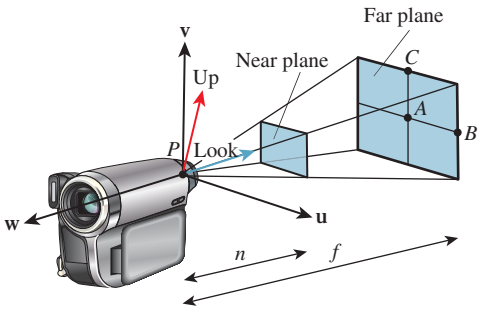


Figure 13.6: The **uvw** *frame for a camera, the* **look** *and* **vup** *vectors, and the points* P, A, B, *and* C.

In writing the equations, we'll use **vup** and **look** to indicate the up vector and look direction, respectively; these shorter names make the equations more comprehensible. The point P is just a shorter name for the camera's `Position`.

We'll build the orthonormal basis, $\mathbf{u}, \mathbf{v}, \mathbf{w}$, in reverse order. First, $\mathbf{w}$ is a unit vector pointing opposite the look direction, so

$$\mathbf{w} = \frac{-\mathbf{look}}{\|\mathbf{look}\|} = \mathcal{S}(\mathbf{look}). \tag{13.2}$$

To construct $\mathbf{v}$, we first project **vup** onto the plane perpendicular to $\mathbf{w}$, and hence perpendicular to the **look** direction as well, and then adjust its length:

$$\bar{\mathbf{v}} = \mathbf{vup} - (\mathbf{vup} \cdot \mathbf{w})\mathbf{w} \tag{13.3}$$

$$\mathbf{v} = \frac{\bar{\mathbf{v}}}{\|\bar{\mathbf{v}}\|} = \mathcal{S}(\bar{\mathbf{v}}). \tag{13.4}$$

Finally, to create a right-handed coordinate system, we let

$$\mathbf{u} = \mathbf{v} \times \mathbf{w}. \tag{13.5}$$

Inline Exercise 13.1: Some camera software (like Direct3D, but not OpenGL) starts by letting $\mathbf{w} = \mathcal{S}(\mathbf{look})$, without negation.
(a) Show that this makes no difference in the computation of $\mathbf{v}$.
(b) Show that in this case, if we want $\mathbf{u}, \mathbf{v}$ to retain the same orientation on the view plane (i.e., $\mathbf{u}$ pointing right, $\mathbf{v}$ pointing up), then the computation of $\mathbf{u}$ becomes $\mathbf{u} = \mathbf{w} \times \mathbf{v}$.
(c) Is the resultant *uvw*-coordinate system right- or left-handed?

Now we'll compute the four points P, A, B, and C. The only subtlety concerns determining the length of edges AB and AC. The edge AB subtends half the horizontal field of view at P, and is at distance f from P, so

$$\tan\left(\frac{\theta_\mathrm{h}}{2}\right) = \frac{AB}{f}, \text{ so} \tag{13.6}$$

$$AB = f \tan\left(\frac{\theta_\mathrm{h}}{2}\right), \tag{13.7}$$

where θ_h denotes the horizontal field of view angle, converted to radians,

$$\theta_\mathrm{h} = \mathrm{FieldOfView}\frac{\pi}{180}, \tag{13.8}$$

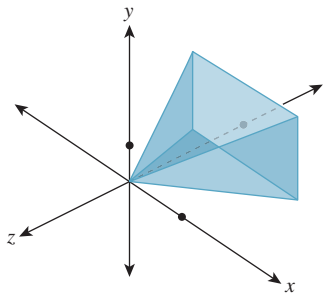


Figure 13.7: The standard perspective view volume is the pyramid that ranges from -1 *to* 1 *in x and y, and from* 0 *to* -1 *in z. The scale in z is exaggerated.*

and similar expressions determine θ_v, the vertical field of view, and the length of AC:

$$P = \text{Position} \tag{13.9}$$

$$A = P - f\mathbf{w} \tag{13.10}$$

$$B = A + f\tan\left(\frac{\theta_h}{2}\right)\mathbf{u} = P + f\tan\left(\frac{\theta_h}{2}\right)\mathbf{u} - f\mathbf{w} \tag{13.11}$$

$$C = A + f\mathbf{v} = P + f\tan\left(\frac{\theta_v}{2}\right)\mathbf{v} - f\mathbf{w}. \tag{13.12}$$

Notice that the near-plane distance n has not entered into our computations yet.

We now use the four points P, A, B, and C to transform the view frustum to the standard view frustum shown in Figure 13.7, and known as the **standard perspective view volume.**

All we need to do is say where the four points should be sent. We want to send P to the origin, A to the midpoint of the back face, which is $(0, 0, -1)$, B to the mid-right edge of the back face, which is $(1, 0, -1)$, and C to the mid-top edge, which is $(0, 1, -1)$. The matrix that performs this transformation is denoted $\mathbf{M}_{\text{per}}$ (for "perspective"), so we'll call the associated transformation `Tper`. The code that creates our transformation is

```
1 Transform3 Tper =
2   Transform3.PointsToPoints(
3     P, A, B, C,
4     Point3(0, 0, 0), Point3(0, 0, -1)), Point3(1, 0, -1), Point3(0, 1, -1));
```

Under this transformation, points of the far plane are transformed to the $z = -1$ plane. Since distances along the ray from P to A must transform linearly, points of the near plane are transformed to the plane $z = -n/f$. We're nearly done at this point: We've transformed the view volume to a standard view volume, and from this point onward, *almost* everything we'll do is independent of the camera parameters, the exception being that the ratio $-n/f$ will enter into some of our computations.

This may appear too simple to you. Indeed, in an earlier edition of this book, the development of the transformation took several pages. But it's an example of the power of proving one good theorem and writing the associated code.

The transformation can also be realized step-by-step. We can take the camera's view volume and apply a sequence of transformations to it: translate it so that P moves to the origin; rotate it several times around various coordinate axes so that the *uvw*-axes align with the *xyz*-axes; scale in z so that the far plane ends up at $z = -1$ instead of $z = -f$; and scale in x and y to make the view frustum have a width and height of 2. Letting P_x, P_y, and P_z denote the world coordinates of P, and similarly for $\mathbf{u}, \mathbf{v}$, and $\mathbf{w}$, the matrix for the transformation is then

$$\mathbf{M}_{\text{per}} = \begin{bmatrix} \frac{1}{f \tan \frac{\theta_h}{2}} & & & \\ & \frac{1}{f \tan \frac{\theta_v}{2}} & & \\ & & \frac{1}{f} & \\ & & & 1 \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & & & -P_x \\ & 1 & & -P_y \\ & & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{13.13}$$

The rightmost matrix effects the translation; the middle one transforms $\mathbf{u}$ to $\mathbf{e}_1$, $\mathbf{v}$ to $\mathbf{e}_2$, and $\mathbf{w}$ to $\mathbf{e}_3$; the leftmost scales each axis appropriately.

We present these merely for your interest and strongly advocate using the `PointsToPoints` method instead, as it's far less prone to errors in order of matrix multiplication, in copying of coordinates, etc.

At this point, it's easy to project points in the standard perspective view volume onto the back plane, for instance, using the nonlinear transformation $(x, y, z) \mapsto (x/z, y/z, 1)$. This is more or less the process we followed when we rendered the cube in Chapter 3: Our **uvw** basis was already aligned with the *xyz*-axes, and our center of projection was chosen to be the center of our coordinate system, so all we had to do was the projection step.

Inline Exercise 13.2: Review the rendering code in Chapter 3 to verify that it matches this description.

Rather than take that approach, however, we're going to apply *two* transformations—the first to "open up" our pyramidal view volume into a rectangular parallelepiped, and the second to project along the z-axis. There are two reasons for this.

- When we discuss "parallel" cameras and projections rather than the perspective cameras we've seen so far, the parallelepipedal volume will be a more natural target than the pyramidal one.
- When we project along the z-axis, it's especially easy to determine which objects obscure which other objects, that is, visibility testing becomes trivial. This property is essential in the design of the so-called z-buffer algorithm at the heart of most graphics hardware.

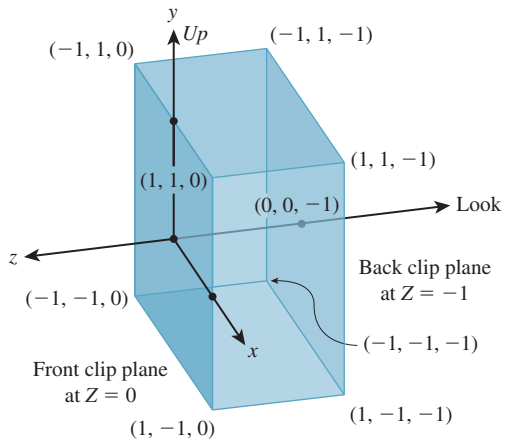


Figure 13.8: The standard parallel view volume.

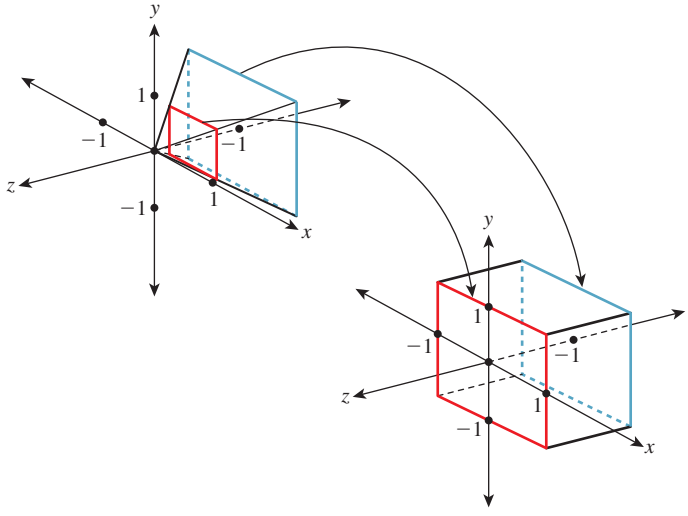


Figure 13.9: The unhinging transformation.

Our **standard parallel view volume** (see Figure 13.8) is a parallelepiped that ranges from -1 to 1 in x and y, and from 0 to -1 in z. Its near clipping plane is $z = 0$; its far clipping plane is $z = -1$. (This differs from the parallel view volume used in either Direct3D or OpenGL, but only slightly.)

Now we'll transform the portion of the standard perspective view volume between the transformed near and far planes (i.e., the portion between $z = -n/f$ and $z = -1$) to the standard parallel view volume. The transformation we use will be a *projective* transformation in which all the rays passing from the view volume toward the origin are transformed into rays passing from the view volume toward the xy-plane in the positive-z direction (see Figure 13.9). (This is sometimes called an **unhinging** transformation, because the planes defining opposite sides of the view frustum meet along a "hinge line," which this transformation "sends to infinity.")

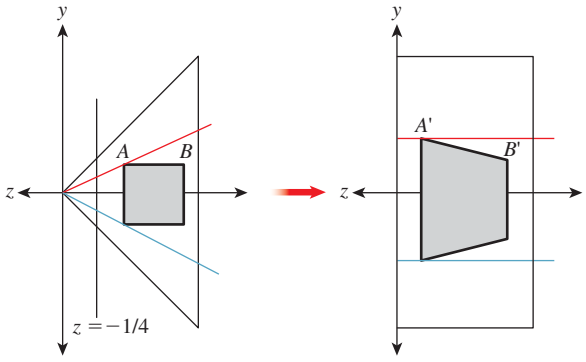


Figure 13.10: The standard perspective view volume at left (with a near clipping plane at $z = -1/4$) contains a small square, which is transformed into a parallelogram in the parallel view volume at right.

Applying this transformation has no impact on the final results of our rendering because the perspective projection (i.e., $(x, y, z) \mapsto (x/z, y/z, 1)$) of a shape in the pre-transformed volume is the same as the parallel projection ($(x, y, z) \mapsto (x, y, 1)$) of the transformed shape in the post-transformed volume. This is easy to see if we look at a two-dimensional slice of the situation, just the yz-plane. Consider, for instance, the small square shown in Figure 13.10 that occupies the middle half of a perspective view of the scene. **Occlusion** (which points are obscured by others) is determined by the ordering of points along rays from the viewpoint into the scene so that the point B is obscured by the near edge of the square. After transformation, that ray from the viewpoint into the scene becomes a ray in the $-z$ direction; once again, the point B' is obscured by the front edge of the square. And once again, the transformed square ends up filling the middle half of the parallel view of the scene. The essential underlying fact is that light (the underlying agent of vision) travels in straight lines, and the transformation we'll build converts straight lines to straight lines (and in particular, the projection rays from the perspective view to projection rays for the parallel view).

Recall from Section 11.1.1 that a *projective* transformation on $\mathbf{R}^3$ can be written as a *linear* transformation on $\mathbf{R}^4$ (the homogeneous-coordinate representation of points in 3-space), followed by the homogenizing transformation

$$H(x, y, z, w) = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1\right). \tag{13.14}$$

Letting $c = \frac{-n}{f}$ denote the z-coordinate of the front clipping plane after transformation to the standard perspective view volume (here at last the parameter n is being used!), we'll simply write down the linear transformation from perspective to parallel, which we call $\mathbf{M}_{\text{pp}}$:

$$\mathbf{M}_{\text{pp}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1/(1+c) & -c/(1+c) \\ 0 & 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & f/(f-n) & n/(f-n) \\ 0 & 0 & -1 & 0 \end{bmatrix}. \tag{13.15}$$

Because we'll be homogenizing in a moment, we can multiply through by $f - n$ and instead use the matrix

$$\mathbf{M}_{\text{pp}} = \begin{bmatrix} f-n & 0 & 0 & 0 \\ 0 & f-n & 0 & 0 \\ 0 & 0 & f & n \\ 0 & 0 & -(f-n) & 0 \end{bmatrix}. \tag{13.16}$$

The derivation of this matrix is slightly messy and not particularly informative; we cover it in this chapter's web materials. For now, all we need to do is verify that it does in fact transform the frustum between the near and far planes ($z = c$ and $z = -1$, respectively) in the standard perspective view volume to the standard parallel view volume. We'll do so by looking at corners.

The only interesting part of the unhinging matrix of Equation 13.15 is in the zw-plane, so it's worth looking at that more closely, remembering that after this transformation, all points will be homogenized (i.e., projected radially from the origin onto the $w = 1$ line in the diagram). Figure 13.11 shows the slice of the view volume before transformation. The thick blue segment at the right on the $w = 1$ line represents the zw-slice of the view frustum between the near and far clipping planes. The thick red segment at the left is the part of the view cone between the near plane and the eye. The red point on the y-axis is the eye. The transformation tilts and stretches the $w = 1$ line (see Figure 13.12). Points at $z = -1$ (the far clipping plane) remain fixed. The near clipping plane is transformed to lie on the $z = 0$ line. The eye is transformed onto the $w = 0$ line. After homogenization (see Figure 13.13), the front clipping plane remains at $z = 0$, while the eyepoint is sent to "infinity on the z-axis," causing the lines that used to meet at the eye to become parallel lines that "meet at infinity on the z-axis" (i.e., parallel lines that are parallel to the z-axis). These three constraints on the transformation are enough to uniquely determine the matrix (see Exercise 13.9).

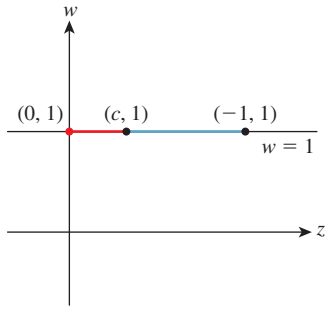


Figure 13.11: Side view of frustum and view volume in the zw-plane before unhinging.

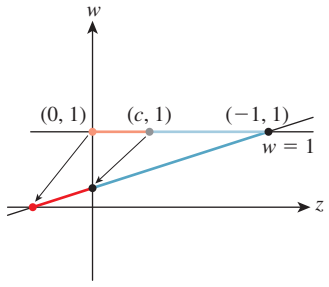


Figure 13.12: After applying $\mathbf{M}_{pp}$.

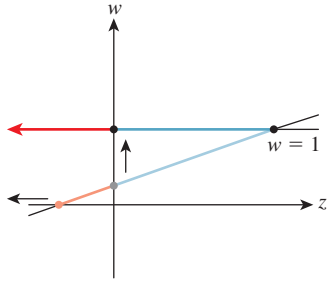


Figure 13.13: After homogenization.

Consider the upper-right front corner of the frustum. It's at the location $(-c, -c, c)$. (Recall that $c = -n/f$ is negative, so $-c$ is positive.) Under the transformation $\mathbf{M}_{\text{pp}}$ it becomes

$$\begin{bmatrix} f-n & 0 & 0 & 0 \\ 0 & f-n & 0 & 0 \\ 0 & 0 & f & n \\ 0 & 0 & -(f-n) & 0 \end{bmatrix} \cdot \begin{bmatrix} -c \\ -c \\ c \\ 1 \end{bmatrix} = \begin{bmatrix} -c(f-n) \\ -c(f-n) \\ cf+n \\ -(f-n)c \end{bmatrix}. \tag{13.17}$$

Homogenizing, we get

$$\begin{bmatrix} 1 & 1 & \frac{cf+n}{-(f-n)c} & 1 \end{bmatrix}^{\mathbf{T}} = \begin{bmatrix} 1 & 1 & 0 & 1 \end{bmatrix}^{\mathbf{T}}, \tag{13.18}$$

the upper-right front corner of the standard perspective view volume, as promised, where the last step depends on $cf + n = -\frac{n}{f}f + n = 0$.

Inline Exercise 13.3: Perform the corresponding computation for the lower-right rear vertex of the perspective view volume, and continue until you're convinced that the transformation works as promised.

Our unhinging transformation places the view volume in the standard parallel view volume, which extends from 0 to -1 in z; objects with *more positive* z-values obscure those with *more negative* z-values. In a hardware z-buffer, the z-values are often stored as unsigned integers; storing them as negative numbers wastes 1 bit for the sign. So, rather than unhinging to a view volume that ranges from 0 to -1, with -1 being "far away," they unhinge to a view volume that ranges from 0 to 1, with 1 being far away. To do this, you need only negate the z-row of the unhinging matrix $\mathbf{M}_{\text{pp}}$.

Alternatively, instead of having the standard parallel view volume extend from 0 to -1 in z, we could have it extend from 1 to 0 in z (i.e., we simply add one to each transformed z-value). Then, although most transformed values would cluster near zero, the problem would be minimized, because if we store them as floating-point numbers, there are many more floating-point numbers near zero than near one. This does in fact improve matters somewhat [AS06].

13.5 Camera Transformations and the Rasterizing Renderer Pipeline

We described in Chapter 1 how graphics processing is typically done. First, geometric models, like the ones created in Chapter 6, are placed in a 3D scene by various geometric transformations. Then these models are "viewed" by a camera, which amounts to transforming their world-space coordinates into coordinates in the standard perspective view volume, and then transforming to the standard *parallel* view volume. Finally, they are projected to a 2D image, and this image is transformed to the viewport where we see a picture.

Along the way, the geometric representation of each model must be processed as shown in Figure 13.14. The 3D world coordinates of primitives (typically triangles) are "clipped" against a view volume; in other words, those outside the view volume are removed from consideration. A triangle that's partly inside the

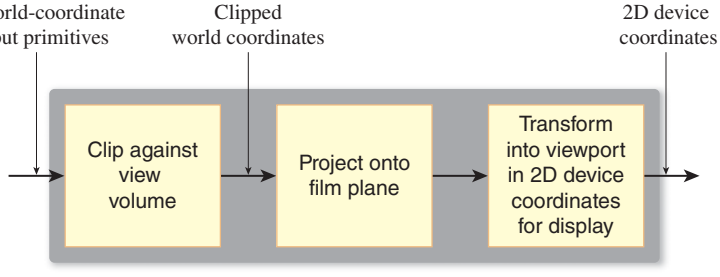


Figure 13.14: Processing of geometry to create images.

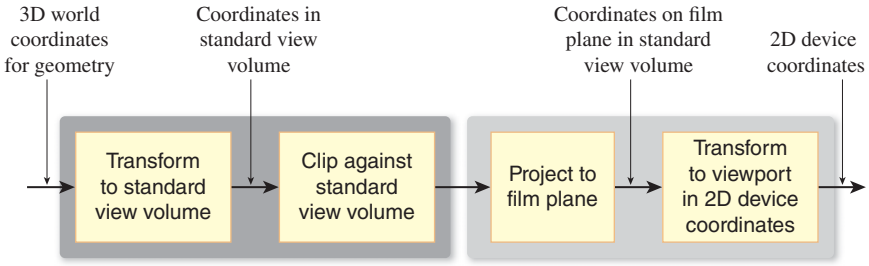


Figure 13.15: Transformation to standard view volume makes clipping simpler.

view volume and partly outside may be truncated to a quadrilateral (and then typically subdivided into two new triangles). Alternatively, a system may determine that the truncation and retriangulation is more expensive than handling the small amount of work of generating pixel data that will never be shown; this depends on the architecture of the hardware doing the rasterization. The clipping operation is discussed in more detail in Chapters 15 and 36.

We'll show how to implement this abstract rendering process in the context of camera transformations. Instead of clipping world coordinates against the camera's view volume, we'll transform the world coordinates to a standard view volume, where clipping is far simpler. In the standard view volume, we end up clipping against coordinate planes like $z = -1$, or against simple planes like $x = z$ or $y = -z$. The projection to the film plane in the second step of the sequence is no longer a generic projection onto a plane in 3-space, but a projection onto a standard plane in the standard parallel view volume, which amounts to simply forgetting the z-coordinate. The revised sequence of operations is shown in Figure 13.15.

The dark gray section (the left half) of the sequence shown in Figure 13.15 can be further expanded for the perspective camera, as shown in Figure 13.16. In this case, we multiply by $\mathbf{M}_{\text{pp}}$ to transform from the standard perspective view volume to the standard parallel view volume, but before homogenizing, we clip out objects with $z < 0$. Why? Because an object with $z < 0$ *and* $w < 0$, after homogeneous division, will transform to one with $z > 0$ and $w = 1$. In practice, this means that objects behind the camera can reappear in front of it, which is not what we want.

Following this first clipping phase, we can homogenize and clip against x and y and the far plane in z, all of which are simple because they involve clipping against planes parallel to coordinate planes.

To interpret the entire sequence of operations mathematically, we start with our world coordinates for triangle vertices and then do the following.

1. Multiply by $\mathbf{M}_{\text{pp}}\mathbf{M}_{\text{per}}$, transforming points into the standard perspective view volume with $\mathbf{M}_{\text{per}}$, and thence toward the standard parallel view volume ($\mathbf{M}_{\text{pp}}$), stopping just short of homogenization.
2. Clip to remove points with $z < 0$. This, and step 4, actually requires knowledge of *triangles* rather than just the vertex data.
3. Apply the homogenizing transformation $(x, y, z, w) \mapsto (\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1)$, at which point we can drop the w-coordinate.

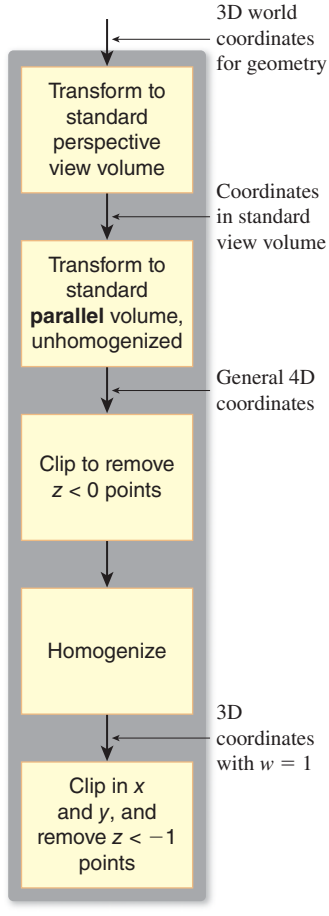


Figure 13.16: Clipping in the perspective case.

4. Clip against $x = \pm 1$, $y = \pm 1$, and $z = -1$.
5. Multiply by $\mathbf{M}_{\text{wind}}$ to transform the points to pixel coordinates.

This description omits two critical steps, which are the determination of the color at each vertex, and the interpolation of these colors across the triangles as we determine which pixels each triangle covers. The first of these was originally called **lighting**, as you learned in Chapter 2, but now the two together are often performed at each pixel by a small GPU program called a shader, and the whole process is therefore sometimes called "shading." These are discussed in several chapters later in this book. For efficiency, however, it's worth noting that lighting is an expensive process, so it's worth delaying as late as possible so that you do the lighting computation for a vertex (or pixel) only if it makes a difference in the final image. The clipping stage is an ideal place to do this. You can avoid work for all the objects that are not visible in the final output. And for many basic lighting rules, it's possible to do the lighting *after* transforming to the standard perspective view volume, or even after transforming to the standard parallel view volume, although *not* after homogenization.[2] Because of this, it makes sense to do *all* the clipping in the pre-homogenized parallel view volume, then do the lighting, and finally homogenize, convert to pixel coordinates, and draw filled polygons with interpolated colors.

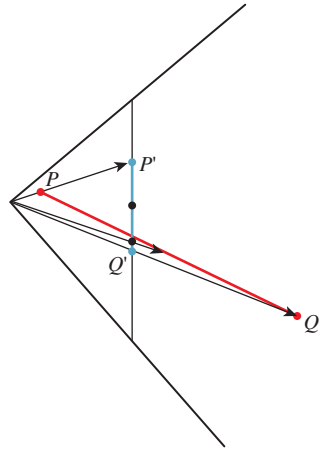


Figure 13.17: The projection of the midpoint of PQ is not the same as the midpoint of the segment $P'Q'$.

What about interpolation of colors over the interior of a triangle, given the color values at the corners? The answer is "It's not as simple as it looks at first." In particular, linearly interpolating in pixel coordinates will not work. To see this, look at the simpler problem shown in Figure 13.17: You've got a line segment PQ in the world, with a value—say, temperature—at each end, and the temperature is interpolated linearly along this line segment so that the midpoint is at a temperature exactly halfway between the endpoints, for instance. Suppose that line segment is transformed into the line segment $P'Q'$ in the viewport. If we take the midpoint $\frac{P+Q}{2}$ and compute the point it transforms to, it will in general *not* be $\frac{P'+Q'}{2}$, so the temperature assigned to $\frac{P'+Q'}{2}$ should *not* be the average of the temperatures for P' and Q'.

The only case where linear interpolation *does* work is when the endpoints P and Q are at the same depth in the scene (measured from the eye). The classic picture of train tracks converging to a point on the horizon provides a good instance of this. Although the crosspieces of the train track ("sleepers" in the United Kingdom, "ties" in the United States) are at constant spacing on the track itself, their spacing in the image is not constant: The distant ties appear very close together in the image. If we assign a number to each tie $(1, 2, 3, \ldots)$, then the tie number varies linearly in world space, but nonlinearly in image space.

This suggests that interpolation in image space may be very messy, but the truth is that it's also not as complicated as it looks at first. In Section 15.6.4.2 we will return to this topic and explain how to perform perspective-correct interpolation simply.

2. Many shading rules depend on dot products, and while linear transformations alter these in ways that are easy to undo, the homogenizing transformation's effects are not easy to undo.

13.6 Perspective and *z*-values

Suppose we consider the perspective-to-parallel transformation $\mathbf{M}_{\text{pp}}$ for the case $c = -\frac{1}{2}$. (Recall that $c = -n/f$ is the z-position of the near plane in the standard perspective view volume.) If we take a sequence of points equally spaced between c and -1 on the z-axis, and apply the transformation and homogenize, then we get a sequence of points between 0 and -1 in the parallel view volume, but they're no longer equally spaced. Figure 13.18 shows the relationship of the new coordinates (z') to the input coordinates (z) for several values of c. When c is near -1, the relationship is near linear; when c is near 0, the relationship is highly nonlinear. You can see how the output values all cluster near $z' = -1$.

Now suppose that the z'-values are to be multiplied by N for some integer N and discretized to integer values between 0 and $N - 1$, as is common in many z-buffers, which use these discretized z'-values to determine which polygon is visible at a given pixel. If c is very small, then all the z'-values will be so near to 1 that they almost all discretize to $N - 1$, and the z-buffer will be unable to determine occlusion. In consequence, if you choose a near plane that's too near the eye, or a far plane that's too distant, then your z-buffer may not perform as expected. The near-plane distance is by far the more important: To avoid so-called z-fighting, you always want to push the near plane as far from the eye as possible while still seeing everything you need to see.

13.7 Camera Transformations and the Modeling Hierarchy

Recall that in Section 10.11 we made a hierarchy of transformations to represent the clock face of Chapter 2, and we said that a similar hierarchy could be created for a 3D model. For a 3D model, the product of all the matrices representing

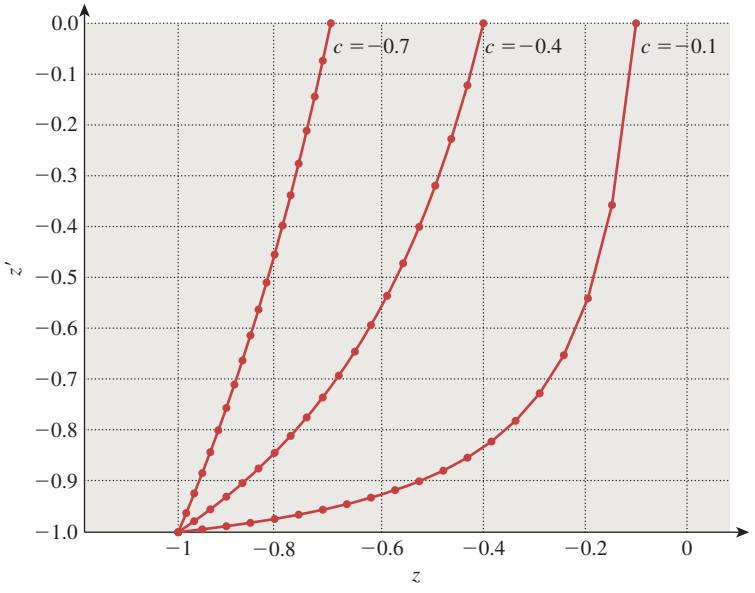


Figure 13.18: Points equispaced in depth in the perspective view volume transformed to unevenly spaced ones in the parallel view volume.

transformations from the world-coordinate system down to some primitive element (typically a triangle specified by its vertex coordinates) is called the **composite modeling transformation matrix** or **CMTM.**

This matrix, multiplied by the modeling coordinates of some vertex, produces the world coordinates of the corresponding point on the modeled object. (Remember that all coordinates need to be expressed homogeneously, to allow us to generate translations, so the CMTM is a 4×4 matrix.)

Inline Exercise 13.4: Explain why the last row of the CMTM, assuming that the transformations in the modeling hierarchy are all translations, rotations, and scaling transformations (i.e., they are all *affine* transformations), must be $\begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix}$.

To transform world coordinates to the standard parallel view volume, we must multiply these coordinates by $\mathbf{M}_{\text{per}}$ and then $\mathbf{M}_{\text{pp}}$, and then homogenize. The product

$$CTM = \mathbf{M}_{\text{pp}} \cdot \mathbf{M}_{\text{per}} \cdot CMTM \tag{13.19}$$

is called (in OpenGL) the **modelview projection matrix** or **composite transformation matrix** or **CTM**.

We can consider the **uvw** triple of vectors, determined by the camera specification, together with the camera location, as defining another coordinate system, **eye coordinates.** To transform a vertex from world to eye coordinates, we must multiply by the matrix

$$\mathbf{N} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{13.20}$$

Inline Exercise 13.5: Confirm that **N** transforms P to the origin, transforms the vector **u** to $\begin{bmatrix}1 & 0 & 0 & 0\end{bmatrix}^{\mathbf{T}}$, and similarly for **v** and **w**.

The product **N***CMTM* is called the **modelview** matrix in OpenGL.

Inline Exercise 13.6: Suppose you've modeled a scene—two robots talking—and have placed a camera so as to view the scene. You want to show a friend the "larger context"—a more distant view of both robots *and* a geometric representation of the camera: a small pyramid whose vertex is at the eye. Fortunately, you happen to have a vertex-and-triangle-table representation of the standard perspective view volume, shortened by a factor of two in the y direction so that it's twice as wide as it is tall. What transformation would you apply to this model to place it in the scene with its apex at the eye and its base parallel to the **uv**-plane, with its y-axis (the shorter one) aligned with **v**?

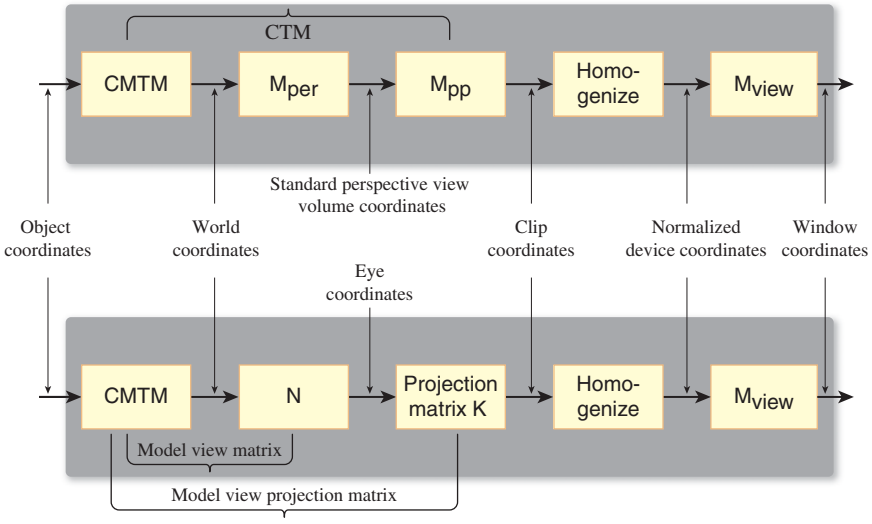


Figure 13.19: OpenGL transformations compared to our transformations.

OpenGL defines another matrix—the projection matrix—that performs the transformation to the pre-homogenization standard parallel view volume. If we call this **K**, then the correspondence with the matrices we've defined is

$$\mathbf{KN} = \mathbf{M}_{\text{pp}}\mathbf{M}_{\text{per}}. \tag{13.21}$$

A comparison of the OpenGL sequence of transformations and ours is shown in Figure 13.19.

Inline Exercise 13.7: Let's return to the two-robots-talking example. Assume that the first robot's right hand is modeled as a unit cube ($-\frac{1}{2} \leq x, y, z \leq \frac{1}{2}$), with its $z = -\frac{1}{2}$ face being attached to the wrist and its $y = \frac{1}{2}$ face being the one that's on top when the arms are held in front of the robot (see Figure 13.20), and that the CMTM for this cube is the matrix **H**. Now suppose that the hand is actually a camera whose eye position is at the center of the $z = \frac{1}{2}$ face (in modeling coordinates), shown as a red dot in the figure. The camera has a 90° field of view, both vertical and horizontal, a near-plane distance of 0. 5, and a far-plane distance of 10. Describe how to find $\mathbf{M}_{\text{per}}$ for this camera so that you can show what the first robot "sees" with its hand camera. Your answer should involve **H**.

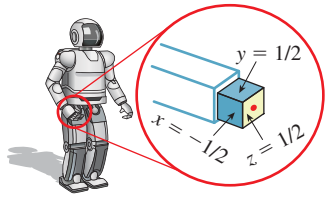


Figure 13.20: The robot hand's sides.

13.8 Orthographic Cameras

While perspective projections are familiar to us from ordinary photographs, many images are created using **parallel projections** or **orthographic projections.** In these projections, we project from the world to the film plane not using a collection of lines that all meet at the eyepoint, but instead using a collection of parallel lines. Imagine a perspective camera with a "film plane" at a fixed location in space, but whose eyepoint moves farther and farther from the film plane, resulting in the lines of projection becoming increasingly parallel (Figure 13.21); we can thus

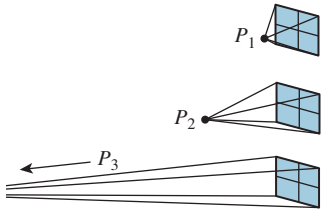


Figure 13.21: Perspective cameras becoming more and more parallel.

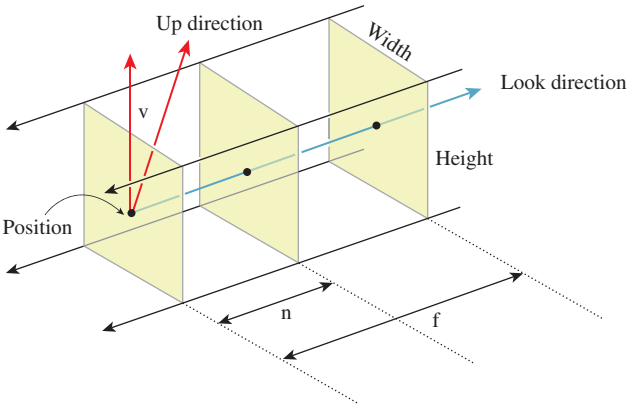


Figure 13.22: The specification of an orthographic camera.

think of a parallel projection as a kind of limit of these perspective projections as the eyepoint moves to infinity. An **orthographic projection** is one in which the parallel lines along which we project are all orthogonal to the film plane. It may seem surprising that you'd ever want the projection lines to *not* be orthogonal to the film plane, but many mechanical drawings are produced this way. This chapter's online materials describe this in some detail. We'll discuss only orthographic cameras here.

An orthographic camera is an abstraction and does not correspond to any physical camera. Figure 13.22 shows how the parts of an orthographic camera are labeled, which corresponds closely to the labeling for a perspective camera. The key distinction is that the "Position" for an orthographic camera does not represent the eyepoint, but rather an arbitrary location in space relative to which we can define the other parts of the camera. The other distinction is that rather than having horizontal and vertical field-of-view angles, we have a width and a height.

For an orthographic camera, we transform directly from the camera's view volume to the standard parallel view volume; the critical step in the construction is

```
1  Transform3 t = Transform3.PointAndVectorsToPointAndVectors(
2     P - n * w, (width/2.0) * u, (height/2.0) * v, (n - f) * w,
3     Point3(0,0,0), Vector3(1,0,0), Vector3(0,1,0), Vector3(0,0,1));
```

where we've used the points-and-vectors form this time, just to show how it can be done. We leave it to you to verify that we have specified the correct transformation.

Inline Exercise 13.8: Rewrite the code for the camera transformation using the points-to-points version.

13.8.1 Aspect Ratio and Field of View

Suppose you want to display a 200×400 image on-screen, a rendering of some virtual world. You need to define a perspective or parallel camera to make that view. Let's work with the somewhat simpler parallel case. It's clear that your parallel camera should have its width set to be twice its height. If you set the width

and height equal, and then display the resultant image on a 200×400 window, horizontal distances will appear stretched compared to vertical ones.

To get a nondistorted on-screen view, assuming that the screen display has square pixels, we need the aspect ratio of the viewport and the image to be the same. Some camera-specification systems let the user specify not the width and height, but instead any two of width, height, and aspect ratio. It's also possible to make the specification of a viewport accept any two of these, making it easier to get cameras and viewports that match. (It's easier to specify width and aspect ratio for both than to specify width and height for both, because in the latter case you'll have to choose the second height to match the aspect ratio established by the first.)

The three parameters—width, height, and aspect ratio—are not independent; if the user specifies all three, it should be treated as an error.

Note that for a perspective camera, the ratio of the vertical and horizontal field-of-view angles is *not* the aspect ratio of the view rectangle (see Exercise 13.1).

13.9 Discussion and Further Reading

The camera model introduced in this chapter is very simple. It's a "camera" suited to the "geometric optics" view of the world, in which light travels along infinitesimally thin rays, etc. Real-world cameras are more complex, the main complexity being that they have *lenses* (or more often, multiple lenses stacked up to make a lens assembly). These lenses serve to focus light on the image plane, and to gather more light than one can get from a pinhole camera, thus allowing them to produce brighter images even in low-light situations. Since we're working with virtual imagery anyhow, brightness isn't a big problem: We can simply scale up all the values stored in an image array. Nonetheless, simulating the effects of real-world lenses can add to the visual realism of a rendered image. For one thing, in real-world cameras, there's often a small range of distances from the camera where objects are in focus; outside this range, things appear blurry. This happens with our eyes as well: When you focus on your computer screen, for instance, the rim of your eyeglasses appears as a blur to you. Photographs made with lenses with a narrow depth of field give the feeling of being like what we see with our own narrow-depth-of-field eyes.

To simulate the effects of cameras with lenses in them, we must, for each pixel we want to render, consider all the ways that light from the scene can arrive at that pixel, that is, consider rays of light passing through each point of the surface of the lens. Since there are infinitely many, this is impractical. On the other hand, by sampling many rays per pixel, we can approximate lens effects surprisingly well. And depending on the detail of the lens model (Does it include chromatic aberration? Does it include nonsphericity?) the simulation can be very realistic. Cook's work on **distribution ray tracing** [CPC84] is the place to start if you want to learn more about this.

There's a rather different approach we can take, based on phenomenology: We can simply take polygons that need to be rendered and blur them somewhat, with the amount of blur varying as a function of distance from the camera. This can achieve a kind of basic depth-of-field effect even in a rasterizing renderer, at very low cost. If, however, the scene contains long, thin polygons with one end close to the camera and the other far away, the blurring will not be effective. Such approaches are better suited for high-speed scenes in video games than for a single, static rendering of a scene.

13.10 Exercises

Exercise 13.1: (a) Suppose that a perspective camera has horizontal and vertical field-of-view angles θ_h and θ_v. What is the aspect ratio (width/height) of the film? (b) Show that if θ_h and θ_v are both small, then the film aspect ratio and the ratio θ_h/θ_v are approximately equal.

Exercise 13.2: Equations 13.2–13.5 show how to determine the **uvw** frame from the look and up directions. Show that the following approach yields the same results:

$$\mathbf{w} = \frac{-\mathbf{look}}{\|\mathbf{look}\|} \tag{13.22}$$

$$\mathbf{t} = \mathbf{w} \times \mathbf{vup} \tag{13.23}$$

$$\mathbf{u} = \frac{\mathbf{t}}{\|\mathbf{t}\|} \tag{13.24}$$

$$\mathbf{v} = \mathbf{u} \times \mathbf{w}, \tag{13.25}$$

Also explain why it's not necessary to normalize **v**.

Exercise 13.3: We noted that as the viewpoint in a perspective view moved farther and farther from the film plane, the view approached a parallel view. Consider the case where the eye is at position $(0, 0, n)$, the near plane is at $z = 0$, and the far plane is at $z = -1$ so that $f = n + 1$. Let $\theta_v = \theta_h = \arctan(\frac{1}{f})$ so that the viewing area on the far plane is $-1 \leq x, y \leq 1$. Write down the product $\mathbf{M}_{pp}\mathbf{M}_{per}$, as a function of n, and see what happens in the limit as $n \to \infty$. Explain the result.

Exercise 13.4: Just as a projective transformation of the plane is determined by its value on four points, a projective transformation of the line is determined by its value on three points. Such a projective transformation always has the form $t \mapsto \frac{at+b}{ct+d}$, where a, b, c, and d are real numbers with $ad - bc \geq 0$.
(a) Suppose you want to send the points $t = 0, 1, \infty$ to 3, 7, and 2, respectively. Find values of a, b, c, and d that make this happen. The value at $t = \infty$ is defined as the limit of values as $t \to \infty$, and turns out to be a/c.
(b) Generalize: If we want $t = 0, 1, \infty$ to be sent to A, B, and C, find the appropriate values of a, b, c, and d.

Exercise 13.5: Create examples to show that a connected n-sided polygon in the plane, when clipped against a square, can produce up to $\lfloor n/2 \rfloor$ disconnected pieces within the square (ignore the parts that are "clipped away"). What is the largest number of pieces that can be produced if the polygon is convex? Explain.

Exercise 13.6: Construct a pinhole camera from a shoebox and a sheet of tissue paper by cutting off one end of the shoebox and replacing it with tissue paper, punching a tiny hole in the other end, and taping the top of the box in place. Stand inside a darkened room that looks out on a bright outdoor scene; look at the tissue paper, pointing the pinhole end of the box toward the window. You should see a faint inverted view of the outdoor scene appear on the tissue paper. Now enlarge the hole somewhat, and again view the scene; notice how much blurrier and brighter the image is. What happens if you make the pinhole a square rather than a circle?

Exercise 13.7: Find a photograph of a person, and estimate the distance from the camera to the subject—let's say it's 3 meters. Have a friend stand at that distance, and determine at what distance you would have to place the photograph

so that the person in the photo occupies about the same visual area as your friend. Is this in fact the distance at which you are likely to view the photo? Try to explain what your brain might be doing when it views such a photo at a distance other than this "ideal."

Exercise 13.8: (a) Fixate on a point on a wall in front of you, and place your arms outstretched to either side. Wiggle your fingers, and move your arms forward until you can *just* detect, in your peripheral vision, the motion on both sides, while remaining fixated on the point in front of you. Have a friend measure the angle subtended by your two arms, at your eyepoint. This gives you some idea of your actual field of view, at least for motion detection.
(b) Have a friend stand behind you, holding his hands out in place of yours, but showing either one, two, or three fingers on each hand. Ask him to move them forward until you can tell how many fingers he's holding up on each hand (while still fixating on the wall). Measure the angle subtended by his hands at your eyes to get a sense of your field of view for nonmoving object comprehension.

Exercise 13.9: We said that the unhinging transform in the zw-plane was uniquely determined by three properties: The plane $z = -n/f$ transforms to $z = 0$; the eye, at $(z, w) = (0, 1)$, transforms to a point with $w = 0$; and the plane $z = -1$ remains fixed. In this exercise, you'll prove this. Restricting to the $x = y = 0$ plane, this last constraint says that the point $(z, w) = (-1, 1)$ transforms to itself. To start with, the matrix we seek is unknown: $\mathbf{M} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$.
(a) Show that the condition on the transformation of the eyepoint implies that $d = 0$.
(b) Now setting $d = 0$, show that the third condition implies that $c = -1$ and $a = b + 1$.
(c) Finally, show that the first condition implies $b = n/(f - n)$, and solve for a as well.

This page intentionally left blank

Chapter 14

Standard Approximations and Representations

14.1 Introduction

The real world contains too much detail for us to simulate it efficiently from first principles of physics and geometry. Mathematical models of the real world and the data structures and algorithms that implement them are always approximations. These approximations make graphics computationally tractable but introduce restrictions and error. The models and approximations are both geometric and algorithmic. For example, a ball is a simple geometric model of an orange. A simple computational model of light interaction might specify that the light passing through glass does not refract or lose energy.

In this chapter, we survey some pervasive approximations and their limitations. This chapter brings together a number of key assumptions about models and data structures for representing them that are implicit in the rest of the book and throughout graphics. It contains some of the engineering conventional wisdom and practical mathematical techniques accumulated over the past 50 years of computer graphics. It is what you need to know to apply your existing mathematics and computer science knowledge to computer graphics as it is practiced today. In order to quickly communicate a breadth of material, we'll stay relatively shallow on details. Where there are deep implications of choosing a particular approximation, a later chapter on each particular topic will explain those implications with more nuance. To keep the text modular (and save you a lot of flipping), there is some duplication of ideas from both prior and succeeding chapters, and we've used some terms and units that have not yet been introduced, like steradians, but whose precise details don't matter in a first reading at this stage.

The code samples in this chapter are based on the freely available OpenGL API (http://opengl.org) and G3D Innovation Engine library (http://g3d.sf.net). We recommend examining the details in the documentation for those or equivalent

alternatives for further study of how these common approximations and representations manifest themselves in programming practice.

14.2 Evaluating Representations

In many cases, there are competing representations that have different properties. Which representation is best suited to *a particular application* depends on the goals of that application. Choosing the right representation for an application is a large part of the art of system design. Some factors to consider when evaluating a representation are

- Physical accuracy
- Perceived accuracy
- Design goals
- Space efficiency
- Time efficiency
- Implementation complexity
- Associated cost of content creation

Physical accuracy is the easiest property to measure objectively. We can use a calibrated camera to measure the energy reflected from a known scene and compare that to a rendering of the scene, for example, as is often done with the Cornell Box (see Figure 14.1).

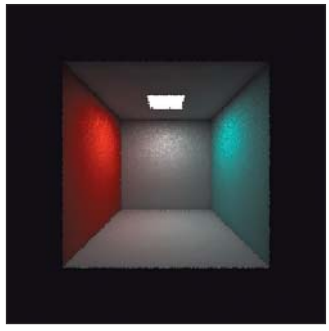


Figure 14.1: The Cornell box, a carefully measured five-sided, painted plywood box with a light source at the top, is used as a standard test model for rendering algorithms. Here it's rendered by photon mapping with 1 million photons.

But physical accuracy is rarely the most important consideration in the creation of images. When the image is to be viewed by a human observer, errors that are imperceptible are less significant than those that are perceptible. So physical accuracy is the wrong metric for image quality. That's also fortunate—regardless of how well we simulate a virtual scene, we are forced to accept huge errors from our displays. Today's displays cannot reproduce the full intensity range of the real world and don't create true 3D light fields into which you can focus your eyes.

Perceived accuracy is a better metric for quality, but it is hard to measure. There are many reasonable models that measure how a human observer will perceive a scene. These are used both for scientific analysis of new algorithms and directly as part of those algorithms—for example, video compression schemes frequently consider the perceptual error introduced by their compression. However, as discussed in Chapter 5, human perception is sensitive to the viewing environment, the task context, the image content, and of course, the particular human involved. So, while we can identify important perceptual trends, it is not possible to precisely quantify the reduction in perceived image quality at the level that we can quantify, say, a reduction in performance.

Even perceptual accuracy is not necessarily a good measure of image quality. A line drawing has little perceptual relationship to the hues and tones in a photograph, yet a good line drawing may be considered a higher-quality depiction of a scene than a poorly composed photograph, as shown in Figure 14.2. The model with best image quality is the one that *best communicates* the virtual scene to the viewer, in the style that the designer desires. This may be, for example, wireframe in a CAD program, painterly in an art piece, cartoony in a video game, or photorealistic for film. Often artists and designers intentionally simplify and deform geometric models, stylize lighting, and remove realism from rendering to better

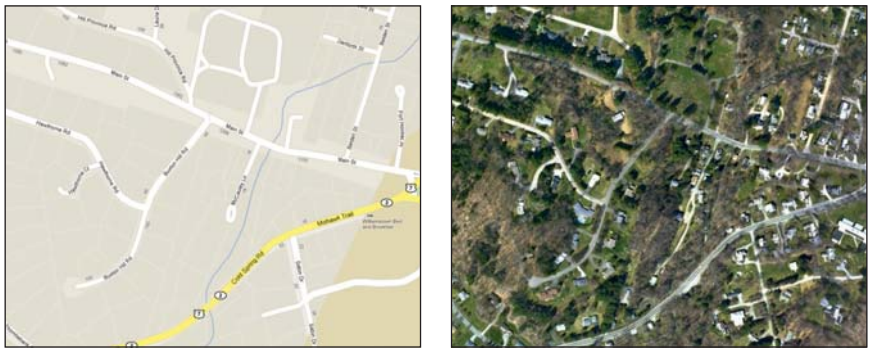


Figure 14.2: A map contains less information and detail than a satellite photograph, but presents its information in a way that better communicates the salient elements to a human viewer. This is evidence that capturing many aspects of reality is not always the most effective way to model a scene. (Credit: © 2012 Google - Map data © 2012 Cnes/Spot Image, DigitalGlobe, GeoEye, MassGIS, Commonwealth of Massachusetts EOEA, New York GIS, USDA Farm Service Agency)

communicate their ideas. This kind of image quality is beyond objective measurement, which is one of the reasons that designing a graphics system is a subjective art as well as an engineering exercise.

Space and time efficiency and implementation complexity go beyond mathematical modeling and into implementation. We seek to actually implement the algorithms that we design and apply them to real problems. For real-time interactive rendering, efficiency is paramount. A low-quality animation that is interactive almost always leads to a better experience in a virtual world than a high-quality one with limited or high-latency interaction. The accessibility and viability of a system in the market is driven by price. The computational and memory requirements, and developer-time costs to build a system, must be balanced against the quality of the images produced.

14.2.1 The Value of Measurement

We can draw some lessons by considering measurements of image quality. Advances in graphics have largely focused on space and time efficiency and physical image quality, even though we claim that perceptual quality, fidelity to the designer's vision, and implementation complexity are also important factors. This is likely because efficiency and physical quality are more amenable to objective measurement. They aren't necessarily easier to optimize for, but the objective measurements allow quantitative optimization. So the first lesson is that if you want something to improve, find an objective way to quantify it. Today's physical image quality is very high, and within some limits we can also achieve very good perceptual image quality. Feature films regularly contain completely computer-generated images that are indistinguishable from photographs, and even low-power mobile devices feature interactive 3D graphics. The second lesson is to make sure that you optimized for what you really wanted. (This is an instance of the Know Your Problem principle from Chapter 1!) Despite the many advances in image quality, the process of modeling, animating, and rendering scenes using either tools or code has not advanced as far as one might

hope. Implementation complexity has skyrocketed over the past 50 years despite (and sometimes because of) graphics middleware libraries and standardization of certain algorithms. Progress has been very slow outside of photorealism, perhaps because the quality of nonphotorealistic renderings is evaluated subjectively. Computer graphics does not today empower the typical user with the expressive and communicative ability of an artist using natural media.

14.2.2 Legacy Models

Beware that in this chapter we describe both the representations that are preferred for current practice and some that are less frequently recommended today. Some of the older techniques make tradeoffs that one might not select intentionally if designing a system from a blank slate today. That can be because they were developed early in the history of computer graphics, before certain aspects were well understood. It can also be because they were developed for systems that lacked the resources to support a more sophisticated model.

We include techniques that we don't recommend using for two reasons. First, this chapter describes what you need to *know,* not what you should *do*. Classic graphics papers contain great key ideas surrounded by modeling artifacts of their publication date. You need to understand the modeling artifacts to separate them from the key ideas. Graphics systems contain models needed to support legacy applications, such as Gouraud interpolation of per-vertex lighting in OpenGL. You will encounter and likely have to help maintain such systems and can't abandon the past in practice.

Second, out-of-fashion ideas have a habit of returning in systems. As we discussed in this section, the best model for an application is rarely the most accurate—there are many factors to be considered. The relative costs of addressing these are highly dynamic. One source of change in cost is due to algorithmic discoveries. For example, the introduction of the fast Fourier transform, the rise of randomized algorithms, and the invention of shading languages changed the efficiency and implementation complexity of major graphics algorithms. Another source of change is hardware. Progress in computer graphics is intimately tied to the "constant factors" prescribed by the computers of the day, such as the ratio of memory size to clock speed or the power draw of a transistor relative to battery capacity. When technological or economic factors change these constants, the preferred models for software change with them. When real-time 3D computer graphics entered the consumer realm, it adopted models that the film industry had abandoned a decade earlier as too primitive. A film industry server farm could bring thousands of times more processing and memory to bear on a single frame than a consumer desktop or game console, so that industry faced a very different quality-to-performance tradeoff. More recently the introduction of 3D graphics in mobile form factors again resurrected some of the lower-quality approximations.

14.3 Real Numbers

An implicit assumption in most computer science is that we can represent real numbers with sufficient accuracy for our application in digital form. In graphics we often find ourselves dangerously close to the limit of available precision, and many errors are attributable to violations of that assumption. So, it is worth

explicitly considering how we approximate real numbers before we build more interesting data structures that use them.

Fixed point, normalized fixed point, and **floating point** are the most pervasive approximations of real numbers employed in computer graphics programs. Each has finite precision, and error tends to increase as more operations are performed. When the precision is too low for a task, surprising errors can arise. These are often hard to debug because the algorithm may be correct—for real numbers—so mathematical tests will yield seemingly inconsistent results. For example, consider a physical simulation in which a ball approaches the ground. The simulator might compute that the ball must fall d meters to exactly contact the ground. It advances the ball $d - 0.0001$ meters, on the assumption that this will represent the state of the system immediately before the contact. However, after that transformation, a subsequent test reveals that the ball is in fact partly *underneath* the ground. This occurs because mathematically true statements, such as $d = d-a+a$ (and especially, $a = (a/b) * b$), may not always hold for a particular approximation of real numbers. This is compounded by optimizing compilers. For example, $a = b + c; e = a + d$ may yield a different result than $e = b + c + d$ due to differing intermediate precision, and even if you write the former, your optimizing compiler may rewrite it as the latter. Perhaps the most commonly observed precision artifact today is self-shadowing "acne" caused by insufficient precision when computing the position of a point in the scene independently relative to the camera and to the light. When these give different results with an error in one direction, the point casts a shadow on itself. This manifests as dark parallel bands and dots across surfaces.

More exotic, and potentially more accurate, representations of real numbers are available than fixed and floating point. For example, **rational numbers** can be accurately encoded as the ratio of two bignums (i.e., dynamic bit-length integers). These rational numbers can be arbitrarily close approximations of real numbers, provided that we're willing to spend the space and time to operate on them. Of course, we are seldom willing to pay that cost.

14.3.1 Fixed Point

Fixed-point representations specify a fixed number of binary digits and the location of a decimal point among those digits. They guarantee equal precision independent of magnitude. Thus, we can always bound the maximum error in the representation of a real number that lies within the representable range. Fixed point leads to fairly simple (i.e., low-cost) hardware implementation because the implementation of fixed-point operations is nearly identical to that of integer operations. The most basic form is exact integer representation, which almost always uses the **two's complement** scheme for efficiently encoding negative values.

Fixed-point representations have four parameters: signed or unsigned, normalized or not, number of integer bits, and number of fractional bits. The latter two are often denoted using a decimal point. For example, "24.8 fixed point format" denotes a fixed-point representation that has 32 bits total, 24 of which are devoted to the integer portion and eight to the fractional portion.

An **unsigned normalized** b-bit fixed-point value corresponding to the integer $0 \leq x \leq 2^b - 1$ is interpreted as the real number $x/(2^b - 1)$, that is, on the range $[0, 1]$. A **signed normalized** fixed-point value has a range of $[-1, 1]$. Since direct mapping of the range $[0, 2^b - 1]$ to $[-1, 1]$ would preclude an exact representation

of 0, it is common to map the two lowest bit patterns to -1, thus sliding the number line slightly and making -1, 0, and 1 all exactly representable.

Normalized values are particularly important in computer graphics because we frequently need to represent unit vectors, dot products of unit vectors, and fractional reflectivities using compact storage.

A terse naming convention is desirable for expressing numeric types in a graphics program because there are frequently so many variations. One common convention for fixed point decorates `int`, or `fix` with prefixes and suffixes. In this convention, the prefix `u` denotes unsigned, the prefix `n` denotes normalized, and the suffix denotes the bit allocations using an underscore instead of a period. For example, `uint8` is an 8-bit unsigned fixed-point number with range [0, 255] and `ufix5_3` is an unsigned fixed-point number with 5 integer bits and 3 fractional bits on the range $[0, 2^5 - 2^{-3}]$ = [0, 31.875]. An even more terse variation of this in OpenGL is the use of `I` to represent non-normalized fixed point and an assumption of unsigned normalized representation. For example, `GL_R8` indicates an 8-bit normalized value (`unint8`) on the range [0, 1] and `GL_RI8` indicates an integer on the range [0, 255].

Some common fixed-point formats currently in use in hardware graphics are unsigned normalized 8-bit for reflectivity, normalized 8-bit for unit vectors, and 24.8 fixed point for 2D positions during rasterization. Fixed-point is infrequently used in modern software rendering. CPUs are not very efficient for most operations on fixed-point formats and software rendering today tends to focus on quality more than performance, so one less frequently seeks minimal data formats if they are inconvenient. The exception is *software rasterization*—24.8 format is used in hardware, not for performance but because fixed-point arithmetic is exact: $a + b - b = a$ (so long as the intermediate results do not overflow), which is not the case for most floating-point a and b.

14.3.2 Floating Point

Floating-point representations allow the location of the decimal point to move—in some cases, far beyond the number of digits. Although the details of the commonly used IEEE 754 floating-point representations are slightly more complicated than scientific notation, the key ideas are similar. A number can be represented as a mantissa and an exponent; for example, $a \times 10^b$ can be encoded by concatenating the bits of a and b, which are themselves integer or fixed-point numbers. In practice, the IEEE 754 representations allow explicit representation of concepts like "not a number" (e.g., $0/0$) and positive and negative infinity. These could be, but rarely are, represented as specific bit patterns in fixed point. Floating point offers increased range or precision over fixed point at the same bit count; the catch is that it rarely offers *both* at the same time. The magnitude of the error in the approximation of a real number depends on the specific number; it tends to be larger for larger-magnitude numbers (see Figures 14.3, 14.4). This makes it complicated to bound the error in algorithms that use this representation. Floating point also tends to require more complicated circuits to implement.

Both 32-bit and 64-bit floating-point numbers (sometimes called single- and double-precision) are common across all application domains. The 32-bit float is often preferred in graphics for space and time efficiency. Graphics also employs other floating-point sizes that are less common in other areas, such as 16-bit "half" precision and some very special-purpose sizes like 10-bit floating

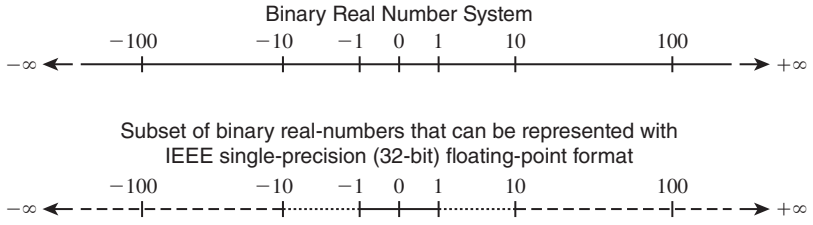


Figure 14.3: Subset of binary real numbers that can be represented with IEEE single-precision (32-bit) floating-point format. (Credit: Courtesy of Intel Corporation)

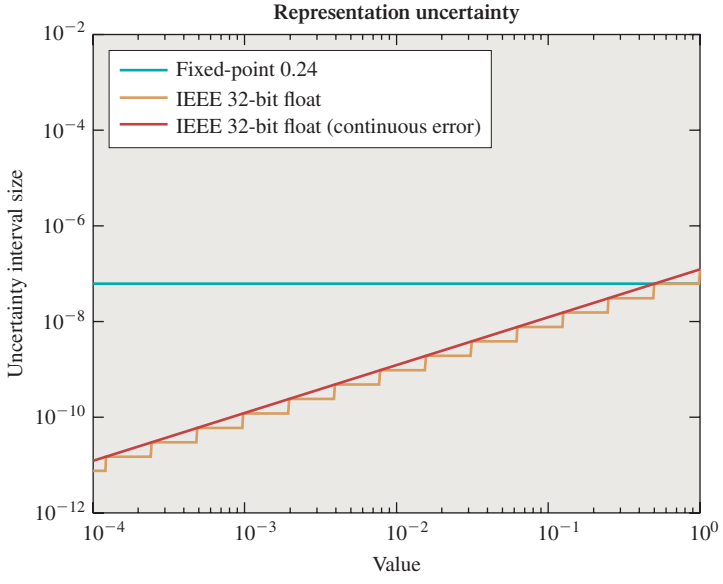


Figure 14.4: Distance between adjacent representable real numbers in 8.24-bit fixed point versus 32-bit floating point [AS06] over the range $[10^{-4}, 1)$. *Floating-point representation accuracy varies with magnitude. ©2006 ACM, Inc. Included here by permission.*

point (a.k.a. 7e3). Ten bits may seem like a strange size given that most architectures prefer power-of-two sizes for data types. In the context of a 3-vector storing XYZ or RGB values, three 10-bit values fit within a 32-bit word (the remaining two bits are then unused). Shared-exponent formats efficiently combine separate mantissas for each vector element with a single exponent [War94]. These are particularly useful for images, in which values may span a large range.

14.3.3 Buffers

The term of art, "buffer," usually refers to a 2D rectangular array of "pixel" values in computer graphics; for example, an image ready for display or a map of the distance from the camera to the object seen at each pixel. Beware that in general computer science, a "buffer" is often a queue (and that sometimes a "2D vector" refers to a 2D array, not a geometric vector!); to avoid confusion, we never use the general computer science terminology in this book.

The **color buffer** holds the image shown on-screen in a graphics system. A reasonable representation might be a 2D array of pixel values, each of which stores three fields: red, green, and blue. Set aside the interpretation of those fields for the moment and consider the implementation details of this representation.

The fields should be small, that is, they should contain few bits. If the color buffer is too large then it might not fit in memory, so it is desirable to make each field as compact as possible without affecting the perceived quality of the final image. Furthermore, ordered access to the color buffer will be substantially faster if the working set fits into the processor's memory cache. The smaller the fields, the more pixels that can fit in cache.

The size of each pixel in bits should be an integer multiple or fraction of the word size of the machine. If each pixel fits into a single word, then the memory system can make aligned read and write operations. Those are usually twice as fast as unaligned memory accesses, which must make two adjacent aligned accesses and synthesize the unaligned result. Aligned memory accesses are also required for hardware vector operations in which a set of adjacent memory locations are read and then processed in parallel. This might give another factor of 32 in performance on a vector architecture that has 32 lanes. If a pixel is larger than a word by an integer multiple, then multiple memory accesses are required to read it; however, vectorization and alignment are still preserved. If a pixel is smaller than a word by an integer multiple, then multiple pixels may be read with each aligned access, giving a kind of super-vectorization.

One common buffer format is shown in Figure 14.5. This figure shows a 3×3 buffer in the `GL_R5G6B5` format. This is a normalized-fixed point format for 16-bit pixels. On a 64-bit computer, four of these pixels can be read or written with a single scalar instruction.

Five bits per channel is not much considering that the human eye can distinguish hundreds of shades of gray. Eight bits per channel enable 256 monochrome shades. But three 8-bit channels consume 24 bits, and most memory systems are built on power-of-two word sizes. One solution is to round up to a 32-bit pixel and simply leave the last eight bits of each pixel unused. This is a common practice beyond graphics—compilers often align the fields of data structures with such unused bits to enable efficient aligned memory access. However, it is also common in graphics to store some other value in the available space. For example, the common `GL_RGBA8` format stores three 8-bit normalized fixed-point color channels and an additional 8-bit normalized fixed-point value called α (or "alpha," represented by an "A") in the remaining space (see Figure 14.6). This value might represent coverage, where $\alpha = 0$ is a pixel that the viewer should be able to see through and $\alpha = 1$ is a completely opaque pixel.

Obviously, on most displays one cannot see through the display itself when a color buffer pixel has $\alpha = 0$; however, the color buffer may not be intended for direct display. Perhaps we are rendering an image that will itself be composited into another image. When writing this book, we prepared horizontal and vertical grid lines of Figure 14.5 as an image in a drawing program and left the pixels that appear "white" on the page as "transparent." The drawing program stored those values with $\alpha = 0$. We then pasted the grid over the text labels "R," "G," etc. Because the color buffer from the grid image indicated that the interior of the grid cells had no coverage, the text labels showed through, rather than being covered with white squares. We return more extensively to coverage and transmission in Section 14.10.2.

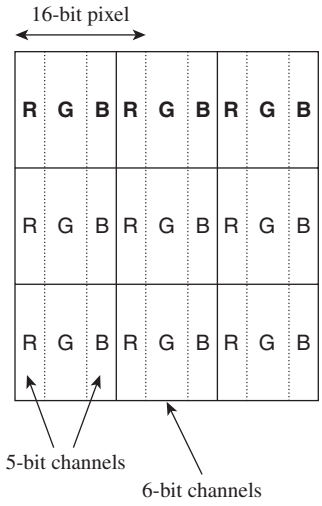


Figure 14.5: The `GL_R5G6B5` buffer format packs three normalized fixed-point values representing red, green, blue, and coverage values, each on [0, 1], into every 16-bit pixel. The red and blue channels each receive five bits. Because 16 is not evenly divisible by three, the "extra" bit is (mostly arbitrarily) assigned to the green channel.

The compositing example is one of many cases where a buffer is intended as input for an algorithm rather than for direct display to a human as an image, and α is only one of many common quantities found in buffers that has no direct visible representation. For example, it is common to store "depth" in a buffer that corresponds 1:1 to the color buffer. A **depth buffer** stores some value that maps monotonically to distance from the center of projection to the surface seen at a pixel (we motivate and show how to implement and use a depth buffer in Chapter 15, and evaluate variations on the method and alternatives extensively in Chapter 36 and Section 36.3 in particular).

Another example is a **stencil buffer,** which stores arbitrary bit codes that are frequently used to mask out parts of an image during processing in the way that a physical stencil (see Figure 14.7) does during painting.

Stencil buffers typically use very few bits, so it is common to pack them into some other buffer. For example, Figure 14.8 shows a 3×3 combined depth-and-stencil buffer in the `GL_DEPTH24STENCIL8` format.

A **framebuffer**[1] is an array of buffers with the same dimensions. For example, a framebuffer might contain a `GL_RGBA8` color buffer and a `GL_DEPTH24STENCIL8` depth-and-stencil buffer. The individual buffers act as parallel arrays of fields at each pixel. A program might have multiple framebuffers with many-to-many relationships to the individual buffers.

Why create the framebuffer level of abstraction at all? In the previous example, instead of two buffers, one storing four channels and one with two, why not simply store a single six-channel buffer? One reason for framebuffers is the many-to-many relationship. Consider a 3D modeling program that shows two views of the same object with a common camera but different rendering styles. The left view is wireframe with hidden lines removed, which allows the artist to see the tessellation of the meshes involved. The right view has full, realistic shading. These images can be rendered with two framebuffers. The framebuffers share a single depth buffer but have different color buffers.

Another reason for framebuffers is that the semantic model of channels of specific-bit widths might not match the true implementation, even though it was motivated by implementation details. For example, depth buffers are highly amenable to lossless spatial compression because of how they are computed from continuous surfaces and the spatial-coherence characteristics of typically rendered scenes. Thus, a compressed representation of the depth buffer might take significantly less space (and correspondingly take less time to access because doing so consumes less memory bandwidth) than a naive representation. Yet the compressed representation in this case still maintains the full precision required by the semantic buffer format requested through an API. Unsurprisingly given these observations, it is common practice to store depth buffers in compressed form but present them with the semantics of uncompressed buffers [HAM06]. Taking advantage of this compressibility, especially using dedicated circuitry in a hardware renderer, requires storing the depth values separately from the

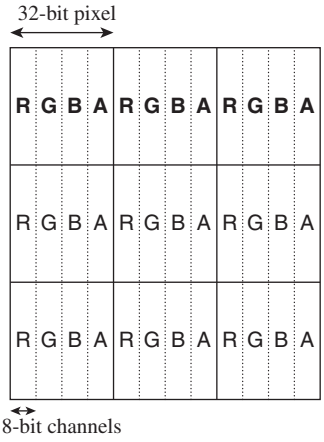


Figure 14.6: The `GL_RGBA8` buffer format packs three 8-bit normalized fixed-point values representing red, green, blue, and coverage values, each on [0, 1], into every 32-bit pixel. This format allows efficient, word-aligned access to an entire pixel for a memory system with 32-bit words. A 64-bit system might fetch two pixels at once and mask off the unneeded bits—although if processing multiple pixels of an image in parallel, both pixels likely need to be read anyway.

Figure 14.7: A real "stencil" is a piece of paper with a shape cut out of it. The stencil is placed against a surface and then painted over. When the stencil is removed, the surface is only painted where the holes were. A computer graphics stencil is a buffer of data that provides similar functionality.

1. The framebuffer is an abstraction of an older idea called the "frame buffer," which was a buffer that held the pixels of the frame. The modern parallel-rendering term is "framebuffer" as a nod to history, but note that it is no longer an actual buffer. It stores the other buffers (depth, color, stencil, etc.). Old "frame buffers" stored multiple "planes" or kinds of values at each pixel, but they often stored these values in the pixel, using an array-of-structs model. Parallel processors don't work as well with an array of structs, so a struct of arrays became preferred for the modern "framebuffer."

other channels. Thus, the framebuffer/color buffer distinction steers the high-level system toward an efficient low-level implementation while abstracting the details of that implementation.

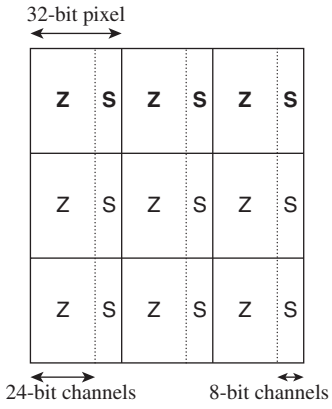


Figure 14.8: The `GL_DEPTH24_STENCIL8` *buffer format encodes a 24-bit normalized fixed point "depth" value with eight stencil bits used for arbitrary masking operations.*

14.4 Building Blocks of Ray Optics

In the real world, light sources emit photons. These scatter through the world and interact with matter. Some scatter from matter, through an aperture, and then onto a sensor. The aperture may be the iris of a human observer and the sensor that person's retina. Alternatively, the aperture may be at the lens of a camera and the sensor the film or CCD that captures the image. Photorealistic rendering models these systems, from emitter to sensor. It depends on five other categories of models:

1. Light
2. Light emitters
3. Light transport
4. Matter
5. Sensors and their imaging apertures and optics (e.g., cameras and eyes)

We now explore the concepts of each category and some high-level aspects that can be abstracted to conserve space, time, and implementation complexity. Later in the chapter we return to specific common models within each category. We must defer that until later because the models interact, so it is important to understand all before refining any.

Although the first few sections of this chapter have covered a great many details, there is a high-level message as well, one that we summarize in a principle we apply throughout the remainder of the chapter:

> ✓ **THE HIGH-LEVEL DESIGN PRINCIPLE:** Start from the broadest possible view. Elements of a graphics system don't separate as cleanly as we might like; you can't design the ideal representation for an emitter without considering its impact on light transport. Investing time at the high level lets us avoid the drawbacks of committing, even if it defers gratification.

14.4.1 Light

14.4.1.1 The Visible Spectrum

The energy of real light is transported by photons. Each photon is a quantized amount of energy, so a powerful beam of light contains more photons than a weak beam with the same spectrum, not more powerful photons. The exact amount of energy per photon determines the frequency of the corresponding electromagnetic wave; we perceive it as color. Low-frequency photons appear red to us and high-frequency ones appear blue, with the entire rainbow spectrum in between (see Figure 14.9). "Low" and "high" here are used relative to the visible spectrum. There are photons whose frequencies are outside the visible spectrum, but those can't directly affect rendering, so they are almost always ignored.

The human visual system perceives light containing a mixture of photons of different frequencies as a color somewhere between those created by the

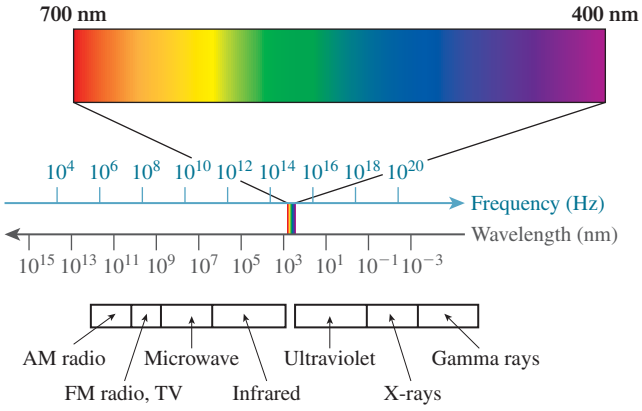


Figure 14.9: The visible spectrum is part of the full electromagnetic spectrum. The color of light that we perceive from an electromagnetic wave is determined by its frequency. The relationship between frequency and wavelength is determined by the medium through which the wave is propagating. (Courtesy of Leonard McMillan)

individual photons. For example, a mixture of "red" and "green" photons appears yellow, and is mostly indistinguishable from pure "yellow" photons. This **aliasing** (i.e., the substitutability of one item for another) is fortunate. It allows displays to create the appearance of many colors using only three relatively narrow frequency bands. Digital cameras also rely on this principle—because the image will be displayed using three frequencies, they only need to measure three.[2] Most significantly for our purposes, almost all 3D rendering treats photons as belonging to three distinct frequencies (or bands of frequencies), corresponding to red, green, and blue. This includes film and games; some niche predictive rendering does simulate more spectral samples. We'll informally refer to rendering with three "frequencies," when what we really mean is "rendering with three frequency bands," Using only three frequencies in simulation minimizes both the space and time cost of rendering algorithms. It creates two limitations. The first is that certain phenomena are impossible to simulate with only three frequencies. For example, the colors of clothing often appear different under fluorescent light and sunlight, even though these light sources may themselves appear fairly similar. This is partly because fluorescent bulbs produce white light by mixing a set of narrow frequency bands, while photons from the sun span the entire visible spectrum. The second limitation of using only three frequencies is that renderers, cameras, and displays rarely use the *same* three frequencies. Each system is able to create the perception of a slightly different space of colors, called a **gamut.** Some colors may simply be outside the gamut of a particular device and lost during capture or display. This also means that the input and output image data for a renderer must be adjusted based on the color profile of the device. Today most devices automatically convert to and from a standard color profile, called sRGB, so color shifts are minimized on such devices but gamut remains a problem.

2. This is not strictly true; Chapter 28 explains why.

The different appearance of cloth under fluorescent light and sunlight is the first example of the Noncommutativity principle—the idea that order matters in some operations we perform in graphics, but that this is often ignored for the sake of speed or simplicity. In this case, the computation of the spectrum of reflected light *should* be carried out with a full representation of the spectrum, and the light should be represented by three samples only when it comes time to store an image preparatory to it being shown on a three-color display. Instead, we've sampled both the spectrum of the emitted light from the source and the reflectance characteristics of the cloth, and multiplied samples rather than spectra. This often produces good-enough results, but can lead to errors.

✓ **THE NONCOMMUTATIVITY PRINCIPLE:** The order of operations often matters in graphics. Swapping the order of operations can introduce both efficiencies in computations and errors in results. You should be sure that you know when you're doing so.

14.4.1.2 Propagation

The speed of propagation of a photon is determined by a material. In a vacuum, it is about $c = 3 \times 10^8$ m/s, which is therefore called the speed of light. The **index of refraction** of a material is the ratio of the speed of light in a vacuum to the rate s of propagation in that material:

$$\eta = \frac{c}{s}. \tag{14.1}$$

For everyday materials, $s < c$, so $\eta \geq 1$ (e.g., household glass has $\eta \approx 1.5$). The exact propagation speed and index of refraction depend on the wavelength of the photon, but the variation is small within the visible spectrum, so it is common to use a single constant for all wavelengths. The primary limitation of this approximation is that the angle of refraction at the interface to a transmissive material is constant for all wavelengths, when it should in fact vary slightly. Effects like rainbows and the spectrum seen from a prism cannot be rendered under this approximation—but when simulating only three wavelengths, rainbows would have only three colors anyway.

Beware that it is common in graphics to refer to the **wavelength** λ of a photon, which is related to temporal frequency[3] f by

$$\lambda = \frac{s}{f}. \tag{14.2}$$

Because the speed of propagation changes when a stream of photons enters a different medium, the wavelength also changes. Yet in graphics we assume that each of our spectral samples is fixed independent of the speed of propagation, so frequency is really what is meant in most cases.

3. Waves have a *temporal* frequency measured in $1/\text{s}$ (i.e., Hz) and a spatial frequency measured in $1/\text{m}$. The *spatial* frequency of a photon is necessarily $1/\lambda$ and is rarely used in graphics because it varies with the speed of propagation.

Photons propagate along rays within a volume that has a uniform index of refraction, even if the material in that volume is chemically or structurally inhomogeneous. Photons are also selectively absorbed, which is why the world looks darker when seen through a thick pane of glass. At the boundary between volumes with different indices of refraction, light **scatters** by reflecting and refracting in complex ways determined by the microscopic geometry and chemistry of the material. Chapter 26 describes the physics and measurement of light in detail, and Chapter 27 discusses scattering.

14.4.1.3 Units

Photons transport **energy,** which is measured in joules. They move immensely fast compared to a human timescale, so renderers simulate the steady state observed under continuous streams of photons. The **power** of a stream of photons is the rate of energy delivery per unit time, measured in watts. You are familiar with appliance labels that measure the *consumption* in watts and kilowatts. Common household lighting solutions today convert 4% to 10% of the power they consume into visible light, so a typical "100 W" incandescent lightbulb emits at best 10 W of visible light, with 4 W being a more typical value.

In addition to measuring power in watts, there are two other measurements of light that appear frequently in rendering. The first is the power per unit area entering or leaving a surface, in units of W/m^2. This is called **irradiance** or **radiosity** and is especially useful for measuring the light transported between matte surfaces like painted walls. The second is the power per unit area per unit solid angle, measured[4] in $\mathrm{W}/(\mathrm{m}^2\ \mathrm{sr})$, which is called **radiance.** It is conserved along a ray in a homogeneous medium. It is the quantity transported between two points on different surfaces, and from a point on a surface to a sample location on the image plane.

14.4.1.4 Implementation

It is common practice to represent all of these quantities using a generic 3-vector class (e.g., as done in the GLSL and HLSL APIs), although in general-purpose languages it is frequently considered better practice to at least name the fields based on their frequency, as shown in Listing 14.1.

Listing 14.1: A general class for recording quantities sampled at three visible frequencies.

```
1  class Color3 {
2  public:
3      /** Magnitude near 650 THz ("red"), either at a single
4          frequency or representing a broad range centered at
5          650 THz, depending on the usage context. 650 THz
6          photons have a wavelength of about 450 nm in air.*/
7      float r;
8
9      /** Near 550 THz ("green"); about 500 nm in air. */
10     float g;
11
12     /** Near 450 THz ("blue"); about 650 nm in air. */
13     float b;
```

4. The unit "sr" is "steradians," a measure of the size of a region on the unit sphere, described in more detail in Section 14.11.1.

```
14
15      Color3() : r(0), g(0), b(0) {}
16      Color3(float r, float g, float b) : r(r), g(g), b(b);
17      Color3 operator*(float s) const {
18        return Color3(s * r, s * g, s * b);
19      }
20      ...
21  };
```

One could use the type system to help track units by creating distinct classes for power, radiance, etc. However, it is often convenient to reduce the complexity of the types in a program by simply aliasing these to the common "color"[5] class, as shown, for example, in Listing 14.2.

Listing 14.2: Aliases of `Color3` with unit semantics.

```
1  typedef Color3 Power3;
2  typedef Color3 Radiosity3;
3  typedef Color3 Radiance3;
4  typedef Color3 Biradiance3;
```

Because bandwidth and total storage space are often limited resources, it is common to employ the fewest bits practical for your needs for each frequency-varying quantity. One implementation strategy is to parameterize the class, as shown in Listing 14.3.

Listing 14.3: A templated `Color` class and instantiations.

```
1  template<class T>
2  class Color3 {
3  public:
4      T r, g, b;
5
6      Color3() : r(0), g(0), b(0) {}
7      ...
8  };
9
10 /** Matches GL_RGB8 format */
11 typedef Color3<unint8> Color3un8;
12
13 /** Matches GL_RGB32F format */
14 typedef Color3<float> Color3f32;
15
16 /** Matches GL_RGB16I format */
17 typedef Color3<unsigned short> Color3ui16;
```

14.4.2 Emitters

Emitters are fairly straightforward to model accurately. They create and cast photons into the scene. The photons have locations, propagation directions, and frequencies (i.e., "colors"), and are emitted at some rate. Given probability distributions for those parameters, we can generate many representative photons and

5. We discuss why color is not a quantifiable phenomenon in Chapter 28; here we use the term in a nontechnical fashion that is casual jargon in the field.

trace them through the scene. We say "representative" because real images are formed by trillions of photons, yet graphics applications can typically estimate the image very well from only a few million photons, so each graphics photon represents many real ones. Today's computers and rendering algorithms can execute a simulation in this model for rendering images in a few minutes. The emission itself isn't particularly expensive. Instead, the later steps of the tracing consume most of the processing time because each representative photon must be handled individually, and the interaction of millions of photons with millions or billions of polygons can be complicated.

To render even faster, we can simplify the emission model so that an aggregate of photons along a light ray can be considered by the later light transport steps. This is a common approximation for real-time rendering. The simplified models tend to fix the origin for all photons from an emitter at a single point. Doing so allows algorithms to amortize the cost of processing light rays from an emitter over the large number of light rays that share a single origin. As we said earlier, it is common practice to consider a small number of frequencies, to simplify the spectral representation, and to treat photons in the aggregate by measuring the average rate of energy emitted at each of those frequencies. Three frequencies loosely corresponding to "red," "green," and "blue" are almost always chosen to represent the visible spectrum, where each represents a weighted sum of the spectral values over an interval of the true spectrum, but is treated during simulation as a point sample, say, at the center of the interval. For an example of a more refined model, Pharr and Humphreys [PH10] describe a renderer with a nice abstraction of spectral curves.

14.4.3 Light Transport

In computer graphics, light transport is almost always modeled by (steady-state) ray optics on uncollimated, unpolarized light. This substantially simplifies the simulation by neglecting phase and polarization. In this model, photons propagate along straight lines through empty space. They do not interfere with one another, and their energy contribution simply sums. Under this simplification and with a discrete set of frequency samples, a geometric ray and a radiance vector (indicating radiance in the red, green, and blue portions of the spectrum) are sufficient to represent a stream of photons.

> In more sophisticated models of light, some physicists model the phase of photons. Such photons can interfere with one another in certain conditions, giving rise to phenomena such as Newton rings. But Newton rings and other small-scale diffraction events rarely occur at noticeable levels in common experience, so we generally ignore them in graphics.

We'll see in Chapter 27 that ignoring photon interference and polarization to simplify the representation of light energy is what forces us to complicate our representation of matter. For example, glossy and perfect reflection arises from the interference of nearly parallel streams of photons. This interference does not arise under ray optics, so we must introduce specific terms (such as Fresnel terms) to materials to model the same phenomena. One could use a richer model of light and a simpler model of a surface to produce the same image. However, a simple

model of matter is not necessarily one that is easy to describe in terms of macroscopic phenomena, either for specification or for digital representation. Representing and modeling a brick as a rough, reddish slab of dried clay is both intuitive and compact. Representing it as a collection of 10^{26} or so molecules of varying composition is unwieldy at best.

14.4.4 Matter

There are many models of matter in graphics. The simplest is that matter is geometry that scatters light, and further, that this light scattering takes place only at the surfaces of opaque objects, ignoring the very small interactions of photons with air over short distances and any subsurface interaction effects. The surface scattering model builds on these assumptions by modeling only the surfaces of opaque objects. This reduces the complexity of a scene substantially. For example, a computer graphics car might have no engine, and a computer graphics house might be only a façade. Only the parts of objects that can interact with light need to be modeled. Of course, this approach poorly represents matter with deep interaction, such as skin and fog, and is only sufficient for rendering. To animate objects, for instance, we need to know properties such as joint locations and masses.

A consequence of computer graphics relying on complex models of matter is that different models are often employed for surface detail at different scales. Supporting different models and ways of combining them at intermediate scales complicates a graphics system. However, it also yields great efficiencies and matches our everyday perception. For example, from 100 meters, you might observe that a fir tree is similar to a green cone. From ten meters, individual branches are visible. From one meter, you can see separate needles. At one centimeter, small bumps and details on the needles and branches emerge. With a light microscope you can see individual cells, and with an electron microscope you can see molecule-scale detail. For this chapter, we consider details to be large-scale if their impact on the silhouette can be observed from about one meter, medium scale if they are smaller than that but observable by the naked eye at some scale, and small-scale if they are not observable by the naked eye.

Figure 14.10: The darkening of this photograph near the edges is called vignetting. (Credit: Swanson Tennis Center at Gustavus Adolphus College by Joe Lencioni, shiftingpixel.com)

14.4.5 Cameras

Lenses and sensors (the components of eyes and cameras) are complicated. This is true whether they have biological or mechanical origins. From a photographer's perspective, the ideal lens would focus all light from a point that is "in focus" onto a single point on the imager (the sensing surface of the sensor) regardless of the frequency of light or the location on the imager. Real lenses have imperfect geometry that distorts the image slightly over the image plane and causes darkening near the edges, an effect known as **vignetting** (see Figure 14.10). They also necessarily focus different frequencies differently, creating an artifact called **chromatic aberration** (see Figure 14.11; see also Chapter 26). Camera manufacturers compensate for these limitations by combining multiple lenses. Unfortunately, these compound lenses absorb more light, create internal reflections, and can diffuse the focus. We perceive the reflections as **lens flare**—a series of iris shapes in line with the light source overlaid on the image, as seen in Figure 14.12. We perceive slightly diffused focus as **bloom,** where very bright objects appear defocused. Real film has a complex nonlinear response to light, and has grain that

Figure 14.11: The rainbowlike edges on the objects in this photograph are caused by chromatic aberration in the camera's lens. Different frequencies of light refract at different angles, so the resultant colors shift in the image plane. High-quality cameras use multiple lenses to compensate for this effect. (Credit: Corepics VOF/Shutterstock)

arises from the manufacturing process. Digital imagers are sensitive to thermal noise and have small divisions between pixels.

Since the simple model of a lens as an ideal focusing device and a sensor as an ideal photon measurement device yields higher image quality than a realistic camera model, there is little reason to use a more realistic model. Because lens flare, film grain, bloom, and vignetting are recognized as elements of realism from films, those are sometimes modeled using a post-processing pass. There is no need to model the true camera optics to produce these effects, since they are being added for aesthetics and not realism. Note that this arises purely from camera culture—except for bloom, none of these effects are observed by the naked eye.

Figure 14.12: The streaks from the sun and apparently translucent-colored polygons and circles along a line through the sun in this photograph are a lens flare created by the intense light reflecting within the multiple lenses of the camera objective. Light from all parts of the scene makes these reflections, but most are so dim compared to the sun that their impact on the image is immeasurable. (Credit: Spiber/Shutterstock)

14.5 Large-Scale Object Geometry

This section describes common models of object surfaces. Many rendering algorithms interact only with those surfaces. Some interact with the interior of objects, whose boundaries can still be represented by these methods. Section 14.7 briefly describes some representations for objects with substantial internal detail.

Some objects are modeled as thin, two-sided surfaces. A butterfly's wing and a thin sheet of cloth might be modeled this way. These models have zero volume—there is no "inside" to the model. More commonly, objects have volume, but the details inside are irrelevant. For an opaque object with volume, the surface typically represents the side seen from the outside of the object. There is no need to model the inner surface or interior details, because they are never seen (see Chapter 36). To eliminate the inner side of the skin of an object, polygons have an orientation. The **front face** of a polygon is the side indicated to face outward and the **back face** is the side that faces inward. A process called **backface culling** eliminates the inward-facing side of each polygon early in the rendering process. Of course, this model is revealed as a single-sided, hollow skin should the viewer ever enter the model and attempt to observe the inside, as you saw in Chapter 6. This happens occasionally in games due to programming errors. Because there is no detail inside such an object and the back faces of the outer skin are not visible, in this case the entire model seems to disappear from view once the viewpoint passes through its surface.

Translucent objects naturally reveal their interior and back faces, so they require special consideration. They are often modeled either as a translucent, two-sided shell, or as two surfaces: an outside-to-inside interface and an inside-to-outside interface. The latter model is necessary for simulating refraction, which is sensitive to whether light rays are entering or leaving the object.

Surface and object geometry is useful for more than rendering. Intersections of geometry are used for modeling and simulation. For example, we can model an ice-cream cone with a bite taken out as a cone topped by a hemisphere . . . with some smaller balls subtracted from the hemisphere. Simulation systems often use **collision proxy geometry** that is substantially simpler than the geometry that is rendered. A character modeled as a mesh of 1 million polygons might be simulated as a collection of 20 ellipsoids. Detecting the intersection of a small number of ellipsoids is more computationally efficient than detecting the intersection of a large number of polygons, yet the resultant perceived inaccuracy of simulation may be small.

14.5.1 Meshes

14.5.1.1 Indexed Triangle Meshes

Indexed triangle meshes (see Chapter 8) are a pervasive surface representation in graphics. The minimal representation is an array of vertices and a list of indices expressing connectivity. There are three common base schemes for the index list. These are called **triangle list** (or sometimes **soup**), **triangle strip,** and **triangle fan** representations. Figures 14.13 through 14.15 describe each in the context of counterclockwise triangle winding and 0-based indexing for a list describing $n > 0$ triangles.

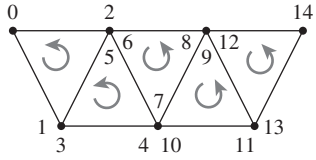


*Figure 14.13: A **triangle list,** also known as a **triangle soup,** contains $3n$ indices. List elements $3t, 3t + 1$, and $3t + 2$ are the ordered indices of triangle t.*

14.5.1.2 Alternative Mesh Structures

For each representation there is a corresponding nonindexed representation as a list whose element *j* is `vertex[index[j]]` from the indexed representation. Such nonindexed representations are occasionally useful for streaming large meshes that would not fit in core memory through a system. Because they duplicate storage of vertices (which are frequently much larger than indices), these representations are out of favor for moderate-sized models.

One can also construct quadrilateral or higher-order polygon meshes following comparable schemes. However, triangles have several advantages because they are the 2D simplex: Triangles are always planar, define an unambiguous barycentric interpolation scheme, never self-intersect, and are irreducible to simpler polygons. These properties also make them slightly easier to rasterize, ray trace, and sample than higher-order polygons. Of course, for data such as city architecture that is naturally modeled by quadrilaterals, a triangle mesh representation increases storage space without increasing model resolution.

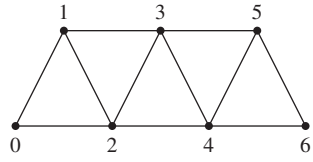


*Figure 14.14: A **triangle strip** contains $n + 2$ indices. The ordered indices of triangle t are given as follows. For even t, use list elements t, $t + 2$, $t + 1$. For odd t, use list elements t, $t + 1$, $t + 2$.*

14.5.1.3 Adjacency Information

Some algorithms require efficient computation of adjacency information between faces, edges, and vertices on a mesh. For example, consider the problem of rendering the contour of a convex mesh for a line-art program. Each edge is drawn only if it is on the contour. An edge lies between two faces. It is on the contour if exactly one face is oriented toward the viewer. We can make this determination more quickly if we augment the minimal indexed mesh with additional information describing faces, edges, and adjacency. Under that representation, we might directly iterate over edges (instead of triangles) and expect constant-time access to the two faces adjacent to each edge.

Adjacency information depends only on topology, so it may be precomputed for an animated mesh so long as the mesh does not "tear apart" under animation. Listing 14.4 gives a possible representation for a mesh with full adjacency information. In the listing, all the integers in the `Vertex`, `Edge`, and `Face` classes are indices into the arrays at the bottom of the class definition. Because faces are oriented, the order of elements matters in their `vertices` index arrays. This is the modern array-based equivalent of a classic mesh data structure called the **winged edge polyhedral representation** [Bau72] (see Chapter 8).

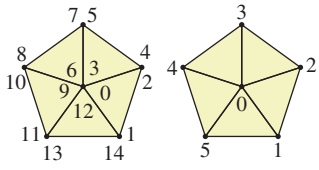


*Figure 14.15: A triangle soup pentagon on the left, and the more efficient triangle fan model on the right. A **triangle fan** contains $n + 2$ indices. List elements 0, $t + 1$, $t + 2$ are the ordered indices of triangle t (with indices taken mod $n + 2$).*

There are several ways to encode the edge information within these data structures. One is to consider the directed **half-edges** that each exist in one face. A true edge that is not on the boundary of the edge would then be a pair of half-edges. The half-edge representation offers the advantage of retaining orientation information when it is reached by following an index from a face. It has the disadvantage of

storing redundant information for all the edges with two adjacent faces. A common trick obfuscates the code a bit by eliminating this storage overhead. The trick is to store only one half-edge for each mesh edge, and to index from a face using two's complement when the half-edge is oriented opposite the direction that it should appear in that face. The two's complement of a non-negative index `e` (written `~e` in C-like languages) is guaranteed to be a negative number, so indices of oppositely directed edges are easy to identify. The two's complement operator is efficient on most architectures, so it incurs little overhead. Each edge then uses the same trick to encode the indices of the adjacent faces, indicating whether that half-edge or its oppositely directed mate actually appears in the face.

Listing 14.4: Sample mesh representation with full adjacency information.

```
struct Mesh {
    enum NO_FACE = MAX_INT;

    struct Vertex {
        Point3            location;
        std::vector<int>  edges;
        std::vector<int>  faces;
    };

    struct Edge {
        int               vertices[2];
        /* May be NO_FACE if this edge is on a boundary. */
        int               faces[2];
    };

    struct Face {
        int               vertices[3];
        int               edges[3];
    };

    std::vector<int>      index;
    std::vector<Vertex>   vertex;
    std::vector<Edge>     edge;
    std::vector<Face>     face;
};
```

14.5.1.4 Per-Vertex Properties

It is common to label the vertices of a mesh with additional information. Common rendering properties include **shading normals, texture coordinates,** and **tangent-space bases.**

A polygonal approximation of a curved surface appears faceted. The perception of faceting can be greatly reduced by shading the surface as if it were curved, that is, by shading the *points* indicated by the surface geometry, but altering the orientation of their tangent plane during illumination computations, as you saw in Chapter 6. It is common to model the orientation by specifying the desired surface normal at each vertex and interpolating between those normals within the surface of each polygon.

Texture coordinates are the additional points or vectors specified at each vertex to create a mapping from the surface of the model to a texture space that defines material properties, such as reflectance spectrum ("color"). Mapping from the surface to a 2D square using 2D points is perhaps the most common, but mappings to 1D spaces, 3D volumetric spaces, and the 2D surface of a 3D sphere are also

common; Chapter 20 discusses this in detail. The last is sometimes called **cube mapping, sphere mapping,** or **environment mapping** depending on the specific parameterization and application.

A tangent space is just a plane that is tangent to a surface at a point. A mesh's tangent space is undefined at edges and vertices. However, when the mesh has vertex normals there is an implied tangent space (the plane perpendicular to the vertex normal) at each vertex. The interpolated normals across faces (and edges) similarly imply tangent spaces at every point on the mesh. Many rendering algorithms depend on the orientation of a surface within its tangent plane. For example, a hair-rendering algorithm that models the hair as a solid "helmet" needs to know the orientation of the hair (i.e., which way it was combed) at every point on the surface. A **tangent-space basis** is one way to specify the orientation; it is simply a pair of linearly independent (and usually orthogonal and unit-length) vectors in the tangent plane. These can be interpolated across the surface of the mesh in the same way that shading normals are; of course, they may cease to be orthogonal and change length as they are interpolated, so it may be necessary to renormalize or even change their direction after interpolation to achieve the goals of a particular algorithm. Finding such a pair of vectors at every point of a closed surface is not always possible, as described in Chapter 25.

14.5.1.5 Cached and Precomputed Information on the Mesh

The preceding section described properties that extend the mesh representation with additional per-vertex information. It is also common to precompute properties of the mesh and store them at vertices to speed later computation, such as curvature information (and the adjacency information that we have already seen). One can even evaluate arbitrary, expensive functions and then approximate their value at points within the mesh (or even within the volume contained by the mesh) by barycentric interpolation.

Gouraud shading is an example. We compute and store direct illumination at vertices during the rendering of a frame, and interpolate these stored values across the interior of each face. This was once common practice for all rasterization renderers. Today it is primarily used only on renderers for which the triangles are small compared to pixels so that there is no loss of shading resolution from the interpolation. The **micropolygon** renderers popular in the film industry use this method, but they ensure that vertices are sufficiently dense in screen space by subdividing large polygons during rendering until each is smaller than a pixel [CCC87]. Per-pixel direct illumination is now considered sufficiently inexpensive because processor performance has grown faster than screen resolutions. However, it has not grown faster than scene complexity, so some algorithms still compute **global** illumination terms such as ambient occlusion (an estimated reduction in brightness due to nearby geometry) or diffuse interreflection at vertices [Bun05].

The vertices of a mesh form a natural data structure for recording values that describe a piecewise linear approximation of an arbitrary function as described in Chapter 9. The drawback of this approach is that other constraints on the modeling process may lead to a tessellation that is not ideal for representing the arbitrary function. For example, many meshes are created by artists with the goal of using the fewest triangles possible to reasonably approximate the silhouette of an object. Large, flat areas of the mesh will therefore contain few triangles. If we were to compute global illumination only at the vertices, we would find that the

illumination computation became extremely blurry in these areas simply because the model had too few vertices.

There are two common solutions to this problem, other than simply increasing the tessellation everywhere. The first is to subdivide triangles of the mesh during computation of the function that is to be stored at vertices [Hec90], until the approximation error across each triangle is small enough. The second is to define an invertible and approximately conformal mapping from the surface of the mesh into texture space, and encode the function values in a texture map. The latter is more efficient for functions with high variance where it is hard to predict the locations where changes occur a priori. Today this approach is more popular than the per-vertex computation. For example, many games rely on **light maps,** which store precomputed global illumination for static scenes in textures. Combined with real-time direct illumination, these provide a reasonable approximation of true global illumination if few objects move within the scene. Traditional light maps encoded only the magnitude, but not direction, of incident light at a surface. This has since been extended to encode directionality in various bases [PRT, AtiHL2]. **Texture-space diffusion,** as seen in d'Eon et al.'s subsurface scattering work [dLE07], is an example of dynamic data encoded in texture space.

14.5.2 Implicit Surfaces

Some geometric primitives are conveniently described by simple equations and correspond closely to shapes we encounter in the world around us. In 2D, these include lines, line segments, arcs of ellipses (including full circles), rectangles, trigonometric expressions such as sine waves, and low-order polynomial curves. In 3D, these include spheres, cylinders, boxes, planes, trigonometric expressions, quadrics, and other low-order polynomial surfaces.

Simple primitives can be represented via implicit equations or explicit parametric equations, as described in Chapter 7. We'll recall some of those ideas briefly here.

An implicit equation is a test function $f : \mathbf{R^3} \rightarrow \mathbf{R}$ that can applied to a point. The function classifies points in space: For any point P, either $f(P) > 0$, $f(P) < 0$, or $f(P) = 0$. Those with $f(P) = 0$ are said to constitute the **implicit surface** defined by f; by convention, those with $f(P) < 0$ are said to be inside the surface, and the remainder are outside. Such a surface is an instance of a **level set** (for level 0) and an **isocontour** (for value 0) of the function.

As an example, consider a surface defined by the plane through point Q with normal $\mathbf{n}$. A suitable test function is

$$f : \mathbf{R^3} \rightarrow \mathbf{R} : P \rightarrow (P - Q) \cdot \mathbf{n}. \tag{14.3}$$

For every point P in the plane, $f(P) = 0$. For points on the side containing $Q + \mathbf{n}$, $f(P) > 0$; for points on the other side, $f(P) < 0$.

An **explicit equation** or **parametric equation** defines a generator function for points in the plane in terms of scalar parameters. We can use such a function to synthesize points on the surface. The explicit form for a plane is

$$g : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R^3} : (u, v) \rightarrow u\mathbf{h} + v\mathbf{k} + Q, \tag{14.4}$$

where $\mathbf{h}$ and $\mathbf{k}$ are two vectors in the plane that are linearly independent. For any particular pair of numbers u and v, the point $g(u, v)$ lies on the plane. Chapter 7

gives both implicit and parametric descriptions for spheres and ellipsoids, and parametric descriptions of several other common shapes like cylinders, cones, and toruses. These, and more general implicit surfaces, are discussed in Chapter 24.

14.5.2.1 Ray-Tracing Implicit Surfaces

Implicit surface models are useful for ray casting and other intersection-based operations. For ray tracing, we take the parametric form of the ray with origin A and direction ω,

$$g(t) = A + t\omega, \tag{14.5}$$

and solve for the point at which it intersects the plane by substituting into the plane's implicit form and finding the roots of the resultant expression. We want to find a value t for which $f(g(t)) = 0$. That means

$$(g(t) - Q) \cdot \mathbf{n} = 0, \text{i.e.,} \tag{14.6}$$

$$(A + t\omega - Q) \cdot \mathbf{n} = 0, \text{so} \tag{14.7}$$

$$t = \frac{(Q - A) \cdot \mathbf{n}}{\omega \cdot \mathbf{n}}. \tag{14.8}$$

We can follow the same process for any surface whose equation admits an efficient closed-form solution after substituting the ray's parametric form.

For a sphere of radius r about the point Q, we can use the implicit form $f(P) = \|Q - P\|^2 - r^2$. Substituting the parametric form for the ray, and setting to zero, we get

$$0 = \|(A + \omega t) - Q\|^2 - r^2 \tag{14.9}$$

$$r^2 = \|(A - Q) + \omega t\|^2 \tag{14.10}$$

$$r^2 = \|(A - Q\|^2 + 2t(A - Q) \cdot \omega + \|\omega\|^2 t^2 \tag{14.11}$$

$$0 = (\|(A - Q\|^2 - r^2) + 2t(A - Q) \cdot \omega + \|\omega\|^2 t^2. \tag{14.12}$$

This is a quadratic equation in t, $at^2 + bt + c = 0$, where $a = \|\omega\|^2$, $b = (A - Q) \cdot \omega$, and $c = \|(A - Q\|^2 - r^2$. It can be solved with the quadratic formula to find all intersections of the ray with the sphere.

Inline Exercise 14.1:
(a) Write out the solutions using the quadratic formula, and simplify.
(b) What does it mean if one of the roots of the quadratic equation is at a value $t < 0$? What about $t = 0$?
(c) In general, if $b^2 - 4ac = 0$ in a quadratic equation, there's only a single root. What does this correspond to geometrically in the ray-sphere intersection problem?

More general quadratics can be used to determine intersections with ellipsoids or hyperboloids, while higher-order polynomials arise in determining the intersection of a ray with a torus, for example, and for more general shapes, the equation we must solve can be very complicated. Multiple roots of the equation that results from substituting the parametric line form into the function defining the implicit surface indicate multiple potential intersections. See Chapter 15 for further discussion of ray casting and interpreting its results.

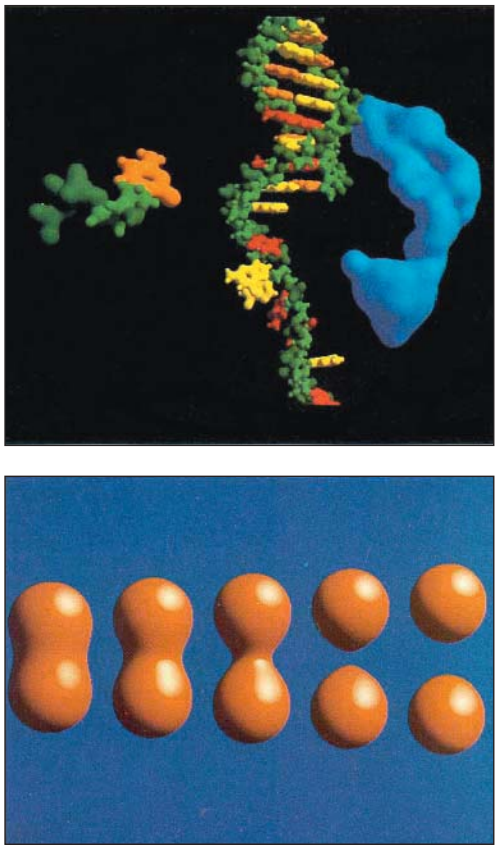


Figure 14.16: Blobby models, each defined by the isocontour of a sum of 3D Gaussian density functions [Bli82a]. (Credit: Courtesy of James Blinn © 1982 ACM, Inc. Reprinted by permission.)

What about implicit surfaces that do not admit efficient closed-form solutions? If the implicit surface function is continuous and maps points inside the object to negative values and points outside the object to positive values, then any root-finding method such as Newton-Raphson [Pre95] will find zero points, that is, it will find the surface. The term "implicit surface" usually refers to this kind of model and intersection algorithm.

Implicit surfaces that are defined by the sum of some simple basis functions with different origins are favored for modeling organic, "blobby" shapes (see Figure 14.16). This is called **blobby modeling** and **metaball modeling** [Bli82a].

14.5.3 Spline Patches and Subdivision Surfaces

We've seen that smooth shapes can be modeled by arbitrary expressions defining their surface curves through three dimensions and by the implicit surface defined by a parametric sum of fixed functions. Spline curves and patches and subdivision curves and surfaces are alternative representations that fall between these extremes. A **spline** is simply a piecewise-polynomial curve, typically represented on each interval as a linear combination of four predefined basis functions, where the coefficients are *points*. Thus, the curve can be represented by just storing

the coefficients, which is very compact. A **spline patch** is a surface constructed analogously: a linear combination of several basis functions (each a function of two variables), where the coefficients are again points. This fixed mathematical form allows compact storage. The fact that the basis functions are carefully constructed, low-degree polynomials makes computations like ray-path intersection, sampling, and determining tangent and normal vectors efficient and fast. By gluing together multiple patches, we can model arbitrarily complex surfaces. (Indeed, spline patches are at the core of most CAD modeling packages.) There are many kinds of splines, each determined by a choice of the so-called basis polynomials. Graphics commonly uses third-order polynomial patches, which let us model surfaces with continuously varying normal vectors and no sharp corners. More general spline types, such as Nonuniform Rational B-Splines (NURBS), have historically been very popular modeling primitives. Splines may be rendered either by discretizing them to polygons by sampling, or by directly intersecting the spline surface, often using a root-finding method such as Newton-Raphson.

Subdivision surfaces are smooth shapes defined by recursive subdivision (using carefully designed rules) and smoothing of an initial mesh cage (see Figure 14.17). Because many modeling tools and many algorithms operate on meshes, subdivision surfaces are a practical method for adapting those tools to curved surfaces. They are especially convenient for polygon-based rendering because the mesh need only be subdivided down to the screen-space sampling density at each location. They have been favored for implementation in graphics hardware over other smooth surface representations because of this. For example, so-called tessellation, hull, and geometry shaders each map meshes to meshes inside the graphics hardware pipeline using subdivision schemes. As with all curve and surface representations, a major challenge is mixing sharp creases and other boundary conditions with smooth interiors. Representations that admit this efficiently and conveniently are an active area of research. At the time of this writing, that research has advanced sufficiently that the techniques are now being used in real-time rendering [CC98, HDD+94, VPBM01, BS05, LS08, KMDZ09].

14.5.4 Heightfields

A **heightfield** is a surface defined by some function of the form $z = f(x, y)$; it necessarily has the property that there is a single "height" z at each (x, y) position. This is a natural representation for large surfaces that are globally roughly planar, but have significant local detail, such as terrain and ocean waves (see Figure 14.18). The single-height property of course means that these models cannot represent overhangs, land bridges, caves, or breaking waves. By the Wise Modeling principle, you should only use heightfields when you're certain that these things are not important to you. At a smaller scale, heightfields can be wrapped around meshes or other surface representations to represent displacements from the surface. For example, we can model a tile floor as a plane with a heightfield representing the grout lines. Heightfields used in this manner are often called **displacement maps** or **bump maps** [Bli78]. "Height" is of course relative to our orientation—it simply denotes distance from the base plane or surface along its normal, so we can use a heightmap to represent the wall of a log cabin simply by rotating our reference frame.

The height function can be implemented by a continuous representation, such as a sum of cosine waves, or by the interpolation of control points. The latter

Figure 14.17: Top: A video-game character from Team Fortress 2 rendered in real time using Approximate Catmull Clark subdivision surfaces. Bottom: The edges of the subdivision cage (projected onto the limit surface) in black, with special crease edges highlighted in bright green. (Credit: top: © Valve, all rights reserved, bottom: Courtesy of Denis Kovacs; © 2010 ACM, Inc. Reprinted by permission.)

representation is particularly good for simulation, modeling, and measured data. The control points may be irregularly spaced so as to efficiently discretize the desired shape (e.g., a Triangulated Irregular Network (TIN), or the ROAM algorithm [DWS+97]), or they can be placed regularly to simplify the algorithms that operate on them. Because of their inability to model overhangs, heightfields are often used as a modeling primitive and then converted to generic meshes. Those meshes may be further edited without the heightfield constraint.

14.5.5 Point Sets

Heightfields, splines, implicit surfaces, and other representations based on control points all define ways to interpolate data from a fixed set of points to define a surface. As we increase the point density, the choice of interpolation scheme has less impact on the shape because the interpolation distances shrink. A natural approach to modeling complex arbitrary shapes is therefore to store dense point

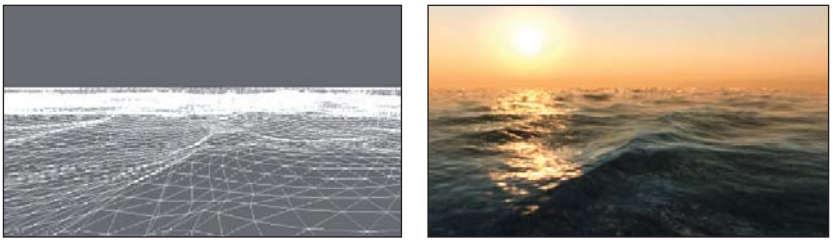


Figure 14.18: Left: The water surface heightfield in CryEngine2. Right: Real-time rendering of the dynamic heightfield [Mit07]. (Credit: Courtesy of Tiago Sousa, ©Crytek)

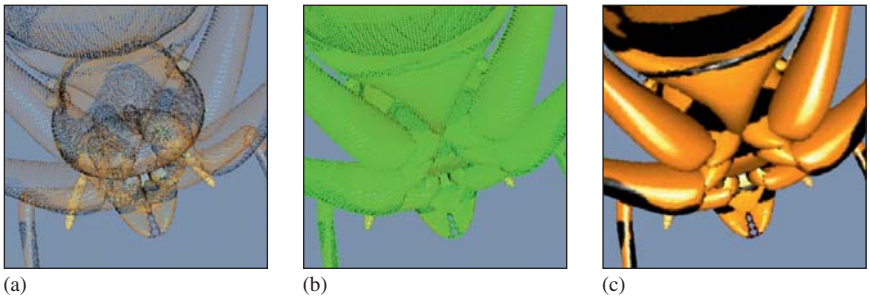
sets and use the most efficient interpolation scheme available. This is a particularly good approach for measured shapes, where the dense point sets naturally arise from the measurement process.

Point-based modeling often stores points at densities so high that the expected viewpoint and resolution will yield about one point per pixel, as shown in Figure 14.19. The interpolation thus need only cover gaps on the order of a pixel. **Splatting** is an efficient interpolation scheme under these conditions: Each point is rasterized as a small sphere (or disk facing the viewer) so that the space between points is covered but the overall shape conforms tightly to the point set. This is simply a form of convolution, and it is also equivalent to an implicit surface defined by a radial function that rapidly falls to zero (and is therefore trivial to evaluate). One can thus also directly ray-trace a point set, using the associated implicit surface.

Because they are a natural fit for measured data but present some efficiency challenges for animation, modeling, and storage, point representations are currently more popular in scientific and medical communities than entertainment and engineering ones.

14.6 Distant Objects

Objects that have a small screen-space footprint or that are sufficiently distant that parallax effects are negligible present an opportunity to improve rendering perfor-



(a) (b) (c)

Figure 14.19: (a) A point set, with attached surface properties. (b) The gaps between points when rendered at this resolution. (c) The surface defined by splatting interpolation of the original points [PZvBG00]. (Credit: courtesy of Hanspeter Pfister, An Wang Professor of Computer Science, © 2000 ACM, Inc. Reprinted by permission.)

mance. Under perspective projection, most of the viewable frustum is "far" from the viewer, and small-scale detail is necessarily less visible there. By simplifying the representation of distant or small objects, we can improve rendering performance with minimal impact on image quality. In fact, a simplified representation may even *improve* image quality because excluding small details prevents them from aliasing, especially under animation (see Section 25.4 for a further discussion of this).

14.6.1 Level of Detail

It is common to create geometric representations of a single object with varying detail and select among them based on the screen-space footprint of the object. This is called a **level of detail (LOD)** scheme [HG97, Lue01]. Discrete LOD schemes contain distinct models. To conceal the transitions, they may blend rendered images of the lower- and higher-detail models when switching levels, or attempt to morph the geometry. Continuous LOD schemes parameterize the model in such a way that these morphing transitions are continuous and inherent in the structure of the model.

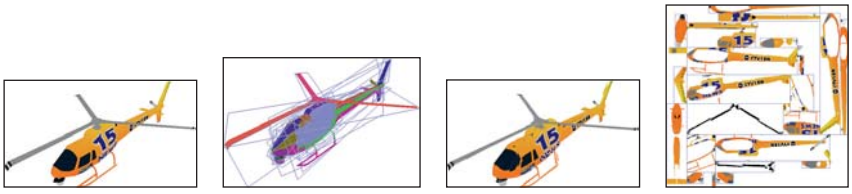
To minimize the loss of perceived detail as actual geometric detail is reduced, structure that is removed from geometry is often approximated in texture maps. For example, the highest-detail variation of a model may contain only geometry, whereas a mid-resolution variation approximates some of the geometry in a normal or displacement map, and the lowest-resolution version alters the shading algorithm to approximate the impact of the implicit subpixel geometry.

Heightfields are a special case that offers a simple LOD strategy. Because the heightfield data is effectively a 2D elevation "image," image filtering operations normally applied to rescaling (Chapter 19) can be applied to compute lower-resolution versions of the heightfield.

14.6.2 Billboards and Impostors

While the location within the viewport of a large, distant, static object changes with the camera's orientation, the projection of that object is largely unchanged under small translations or rotations. Thus, a complex three-dimensional shape in the distance can be approximated by a flat, so-called **billboard** that bears a picture of the object rendered from approximately the current viewpoint. Such billboards are inexpensive to render because they are simply quadrilaterals with images mapped over them. Billboards may be used as the lowest level of detail in an LOD scheme, or as the only level of detail if it is known that the viewer will never approach the object. In some cases, billboards are also used for objects that are naturally flat, exhibit rotational symmetry, or for which orientation errors are difficult to notice. For example, a cluster of many leaves on a tree may be modeled as a single billboard, and likewise for a clump of many blades of grass. It is common to automatically rotate billboards toward the viewer during rendering to conceal their flat nature, although this is not appropriate in all cases. For example, distant tree billboards should rotate around their vertical axis to face the viewer, but should not rotate to face a viewer flying above the forest because doing so would make it appear that the trees had fallen over.

To increase realism, billboards can be augmented with surface normals or displacement maps [Sch97] that allow dynamic relighting.



(a) (b) (c) (d)

Figure 14.20: Example of a billboard cloud: (a) original model (5138 polygons), (b) false-color rendering using one color per billboard to show the faces that were grouped on each billboard, (c) view of the (automatically generated) 32 textured billboards, and (d) the billboards side-by-side. [DDSD03]. (Credit: Courtesy of Xavier Décoret, © 2003 ACM, Inc. Reprinted by permission.)

Décoret et al. [DDSD03] proposed **billboard clouds** to automate for any model a process often employed by artists for foliage. The billboard cloud represents a single object with a collection of billboards oriented to incrementally minimize visual error in the rendered object (see Figure 14.20).

A limitation of individual billboards is that they cannot represent dynamic objects or views of objects as the observer approaches and parallax becomes non-negligible. To address parallax, one could precompute *many* billboards, as was common in early 3D games such as Doom and Wing Commander, or develop warping strategies [POC05]. For dynamic billboards of specific objects, one could rig animation controls within the billboard itself [DHOO05, YD08]. However, a general solution is to simply rerender the billboards at runtime whenever the approximation error grows too large. These dynamic billboards are known as **imposters** [MS95], and they have seen widespread application for a variety of models, from terrain [CSKK99] to characters to clouds [HL01].

14.6.3 Skyboxes

It is often convenient to model parts of a scene as effectively "infinitely" distant. These may be rendered using finite distance for projection, but those distances are held constant regardless of the viewer's translation. A frequent application is the sky, including clouds. For a character on the ground, the distance to objects in the sky is effectively constant and large, so there is no parallax or change in perspective with viewpoint movement. This is the ideal case for a billboard, except that planar geometry is inappropriate for wrapping around the horizon. A **skybox** or **sky sphere** is a geometric proxy for all distant objects. It wraps around the scene and translates so that the viewer is always at the center. The geometry for this proxy is arbitrary, so long as it surrounds the viewer. For example, it could be an icosahedron, tetrahedron, . . . or teapot, and the shape will be indistinguishable from a sphere once the interior is painted with an appropriately projected image of the (virtual) distant scene geometry that it simulates. The choice of proxy geometry is therefore driven by the convenience and efficiency of generating that image under the given projection. Cubes and spheres both lend themselves to natural projections, and are therefore the most common models.

The term "skybox" is also used occasionally to refer to objects at finite distances such as building façades that provide a small amount of parallax but are in areas of the scene that the viewer will never enter. This is common, for example,

in video games, where the player character's movement is often constrained by natural obstacles but the designer wishes to efficiently represent a larger world than the navigable portion of the scene.

14.7 Volumetric Models

Most of the descriptions of matter that we've surveyed are surface representations. These are extremely efficient because they are mostly "empty"; they need not explicitly represent the space inside objects.

Volumetric modeling methods represent solid shapes rather than surfaces. Doing so enables richer simulation, both for dynamics and for illumination in the presence of translucency.

14.7.1 Finite Element Models

Finite element models are general divisions of solid objects into polyhedral chunks. These are very popular for detailed engineering simulation to model the internal forces within objects, heat and pressure propagation, and fluid flow. They are less popular for pure-rendering applications because they offer few advantages in that context over surface meshes.

A regular finite element subdivision into tetrahedrons or cubes offers additional advantages for modeling and simulation applications. Regularizing shapes allows constant-time random spatial access and stabilizes propagation. The tetrahedral division is good for simulation because the tetrahedron is the three-dimensional simplex—it is the simplest polyhedron, and is therefore a good primitive to model effects like fracture. The cube division naturally lends itself to a regular grid, making for straightforward representations, and is also easy to build hierarchies from. This representation is known as a **voxel** model. It is very common for fluid flow simulation and medical or geoscientific imaging, where the underlying source data are often captured on a regular grid.

14.7.2 Voxels

Voxels have gone in and out of favor for rendering, especially in entertainment. Figure 14.21 shows a contemporary game, Minecraft, which models the world with large voxels to intentionally inspire a building-block aesthetic. The game takes advantage of the efficiency of local graphlike operations on voxels to model all illumination and physical dynamics as cellular finite automata. Because the voxel representation requires only storage of the type of material in each cell (the position is implicit in the 3D array), the game is able to efficiently represent huge worlds with a single byte per cubic meter of storage—and large homogeneous regions are amenable to further compression. In comparison, modeling the same world even as a triangle list of cubes would require 12 triangles $\times$ 3 vertices/triangle $\times$ 3 floats/vertex $\times$ 4 bytes/float = 432 bytes per cubic meter, and would be less amenable to compression.

Note that the scene in Figure 14.21 appears to have detail at finer resolution than the 1m^3-voxel grid. For example, fences and reeds are represented by thin objects within a single grid cell. This is because that rendering system uses the voxels for simulation, but for rendering it replaces each with a proxy object that may be more detailed than a simple cube. This is an extreme form of **geometry**

Figure 14.21: The game Minecraft models the entire world with $1\,m^3$ voxels, enabling efficient, real-time illumination, simulation, and rendering for fully dynamic Earth-scale worlds.

instancing. Geometry instancing is often used in less rigid scene representations to efficiently represent many similar elements. For example, a forest can be modeled with only a handful of individual tree models and a large number of tree locations and reference frames. One tree in the forest model then only requires storage for a *pointer* to a tree model and a coordinate frame, rather than for a unique tree geometry. Voxel scenes with relatively large-scale voxels can use a similar scheme to present higher apparent resolution than the voxel grid without the cost of modeling explicit fine-scale geometry for every voxel.

Figure 14.22 demonstrates a more refined use of voxels for efficient rendering of a high-resolution, static scene. Ray tracing through voxel grids is extremely efficient because the ray-surface intersections are trivial and the grid provides good spatial locality in the memory system. A tree data structure allows efficient encoding of large empty regions. Note that even when viewed up close, the voxels do not appear blocky. This image was rendered with a technique by Laine and Karras [LK10] that stores a surface plane along with shading information at each voxel, allowing surface reconstruction at apparently higher resolution along planes other than the grid itself. Several such techniques exist; one commonly used for fluid simulation is marching cubes [LC87] (and marching tetrahedrons [CP98]), which, given only density information stored in voxels, reconstructs some simple geometry in each voxel to produce a relatively smooth mesh (see Section 24.6).

Figure 14.22: Voxel data created by high-resolution surface displacement, with local shadowing precomputed and stored in the voxel grid. The resolution is approximately 5 mm throughout the entire building, including outer walls that are not visible from the inside. The total size of the data in GPU memory is 2.7 GB. Laine and Karras's ray caster was able to cast about 61 million rays per second when rendering this scene; in other words, to render 1M pixels at 60 fps by ray tracing in 2010 [LK10]. (Credit: Courtesy of Samuli Laine and Tero Karras, © 2010 ACM, Inc. Reprinted by permission.)

14.7.3 Particle Systems

Liquid or gaseous objects such as smoke, clouds, fire, and water are often modeled as **particle systems** [Ree83]. A particle system contains a set of individual particles, each of which can be efficiently simulated as a single point mass. There may be a large number of particles—say, thousands or millions—that play a role similar to individual molecules of gas or liquid. However, the simulation typically contains orders of magnitude fewer particles than a real-world scene would contain molecules. During rendering, the relatively low particle count is

concealed by rendering a small billboard for each particle. This is similar to the splatting operation in point-based rendering. One usually calls a dynamic object with translucent billboards a particle system and a rigid object with opaque splats a point set. Section 14.10 describes methods for simulating translucency for both meshes and particles.

The billboard nature of particle systems can be revealed when the billboards intersect other geometry in the scene. **Soft particles** [Lor07] are a technique for concealing this intersection (see Figure 14.23). Soft particles become more translucent as they approach other geometry. Proximity is determined by reading a depth buffer during shading. The effect is particularly convincing for billboards that have high density and little visible structure, such as smoke.

Before:

After:

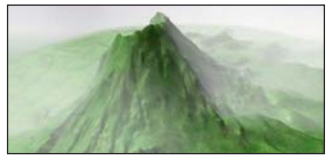


Figure 14.23: Top: The flat, discrete nature of this cloud particle system's rendering billboards is revealed where it intersects the terrain mesh. Bottom: Adjusting the pixel shader to use the "soft particle" technique that fades out the billboard's contribution with proximity to scene geometry conceals this artifact. (Credit: Courtesy of Tristan Lorach, NVIDIA)

14.7.4 Fog

Particles and voxels are discrete representations of amorphous volumetric shapes. Homogeneous, translucent volumes are amenable to continuous analytic representation. The classic application is atmospheric perspective, the relatively small-scale scattering of light by the atmosphere that desaturates distant objects in landscapes. A more extreme variation of the same principle is dense fog, which may be homogeneous over all space or vary in density with elevation.

True atmospheric perspective necessarily involves exponential absorption with distance, but it is often artistically desirable to present arbitrary control over the absorption rate. Homogeneous fog is implemented either during shading by blending the computed shade of each pixel toward the fog color based on distance from the viewer (e.g., in a pixel shader or employing the fixed-function `glFog` command in OpenGL), or by a 2D image post-process that performs the equivalent blending based on depth buffer values. An example of this blending to compute final color c' from distance d, original color c, fog color f, and fog density parameter κ is (following the `glFogf` documentation)

$$c' = c + (f - c) \cdot e^{-d\kappa}. \tag{14.13}$$

The same approach can be applied to the scattering/attenuation of light when the camera is underwater. Much more sophisticated models of atmospheric scattering have been developed (e.g., [NMN87, Wat90, NN94, NDN96, DYN02, HP03]); this common exponential approximation is only the beginning.

Localized fog volumes (see Figure 14.24) can follow the same attenuation schemes as global ones, but the distance on which they are parameterized must measure only the extent traveled through the fog along the view ray, not the total distance from the observed surface to the viewer. This distance is computed by ray intersection with the bounding volume of the fog. Doing so within the shading algorithm for each pixel is reasonable provided that the bounding volume is geometrically simple. Half-plane, rectangular slab, and sphere volumes are common.

14.8 Scene Graphs

It is rare for large graphics systems to treat a scene as a single object. Instead, the scene is typically decomposed into a set of individual objects. This allows different model representations for different parts of a scene. It also reduces the memory size of objects that must be processed to both better accommodate computational

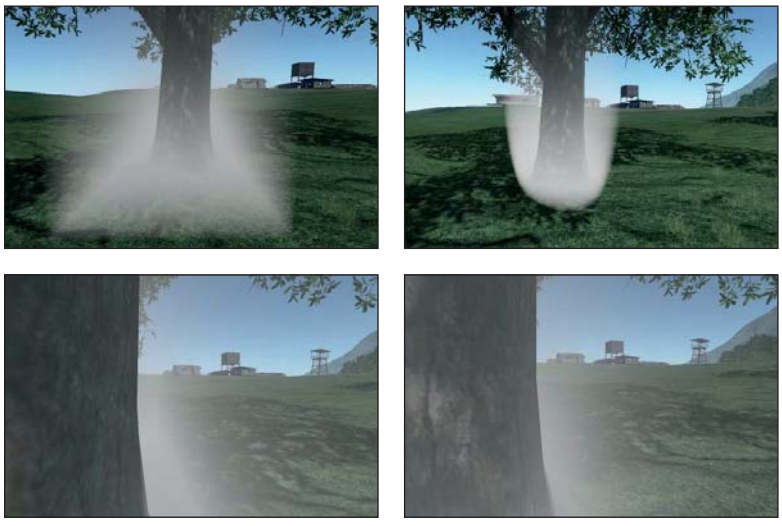


Figure 14.24: Box and ellipsoid fog volumes rendered by intersecting the view ray with an analytic volume inside a pixel shader. (Credit: Courtesy of Carsten Wenzel, © Crytek)

limitations and provide manageable data sizes for the comprehension of human modelers and programmers. That is, this decomposition simply follows classic computer science and software engineering abstraction principles.

The data structure for maintaining the collection of objects in a scene is called a **scene graph,** where "graph" refers to pointers that express relationships between objects; you've already encountered a basic scene graph in the modeling hierarchy of Chapter 6, and the discussions of its traversal in Chapters 10 and 11. There are many scene-graph data structures. Deep trees are well suited to modeling and user-interface elements, where lots of fine-grained abstractions and a low branching factor match human design instincts. Relatively broad and shallow trees are often well suited to rendering on hardware with many parallel processing units and efficient object-level culling. Physical simulation often requires full graphs to express cyclic relationships in the simulation.

More-or-less aligned with the three goals of modeling and interaction, rendering, and simulation, there are three broad strategies for dividing the scene into elements. Classic scene graphs and shading trees divide a scene into **semantic** elements. For example, a character model might contain a "hair" node that is a child of a "head" node to enable easy coloring or replacement of hair. One might also attach a "skin color" property to a root node at the character's torso that propagates that color property throughout the model. Semantic nodes are very similar to the cascading property schemes employed by text markup languages like HTML. This is not surprising; markup effectively describes a scene graph for text layout and rendering. Semantic scene graphics are nearly always directed acyclic graphs. A child node typically inherits shading and simulation properties from its parent in addition to a coordinate reference frame.

Physics scene graphs typically express constraint relationships (edges) between objects (nodes). These constraints are often joints. For example, a character's wrist is a constraint that defines the coordinate transformation between the forearm and the hand. The constraints may be ephemeral; for example, a bouncing ball temporarily is constrained to not penetrate the ground (and perhaps experience limited lateral slip) on contact. Most dynamics systems include both prerigged character and machine articulation graphs and context-dependent

constraint graphs for forces and contacts. See Chapter 35 for a discussion of dynamics and articulation data structures and algorithms.

Spatial data structures/scene graphs are the close analogue of general computer science data structures such as lists, trees, and arrays. They divide the scene into grids or trees that allow efficient spatial queries, such as "Which objects are within 4 m of my avatar?" Child nodes are typically contained within the bounding volumes of their parents. Spatial data structures are employed extensively in simulation and rendering. They are usually computed automatically. Efficient algorithms for building these data structures per-frame have recently emerged and this is an active area of research. Chapters 36 and 37 discuss modeling and interaction data structures and algorithms.

14.9 Material Models

As we said earlier, we conventionally think of objects as defined by their surfaces, which are the boundaries between them and other objects or the surrounding medium. But the solid nature of objects also has an effect on their interaction with light. Fortunately, we can limit our consideration to the interface between two media through which light propagates differently. That there are *two* media involved is essential. The appearance of a surface depends on both, although we commonly observe most objects in air, so this is not always apparent. For the moment, let us assume that objects are rendered in air so that we can define appearance using a single material parameter. We will return to the two-material case in Section 14.10.

The interaction of light and matter is quite simple. To a first approximation, each photon that strikes the surface has one of three fates: It is absorbed and converted into heat, it passes through the surface into the medium, or it reflects. The probability of each of these outcomes and the direction that scattered photons take after the interaction is governed by the materials involved and the microscopic angle of the plane of the surface near the location hit. A few simple laws from physics can describe the entire model.

However, we use high-level models that intentionally introduce more complexity than is present in these simple laws. Doing so lets us work with large numbers of photons in the aggregate and large (or at least, macroscopic) patches of surface. So, in exchange for complicating the material model, we can use simpler geometric surface models and light-sampling strategies. A more complex material model also allows aesthetic controls instead of physical ones, enabling artists to achieve their visions using intuition instead of measurement.

It is common practice to distinguish at least the following five artistically and perceptually significant phenomena.

1. Sharp **specular** (mirror) reflections, as seen on glass.
2. **Glossy** highlights and reflections, like the highlights on a waxed apple.
3. Shallow subsurface scattering, which produces matte **Lambertian** shading that is independent of the viewer's orientation, such as observed with "flat" wall paint.
4. Deep **subsurface scattering** where light diffuses beneath the surface. This is what makes skin and marble appear soft.
5. **Transmission,** where light passes through a mostly translucent material such as water or fog, perhaps being slightly diffused along the way and refracted when it enters this medium.

Since these all just describe scattering (and lack of scattering, due to absorption), they are typically described by a **scattering function.** There are several variations, among them the **bidirectional scattering distribution function** (BSDF) for surface scattering, the reflectance-only variant (BRDF) for purely opaque surfaces, the BTDF for purely transmissive surfaces, and the BSSDF for describing both surface and shallow subsurface effects. BSDFs alone require fairly in-depth discussion of a specific rendering algorithm and surface physics to describe properly. Fortunately, one can also get by with a fairly simple model and application of it. A substantial portion of the pixels rendered in the past 30 years all used variations on the same simplified model, and it will likely be with us for some time.

In the following subsections we sketch the basic idea of a BSDF interface and one of the simple phenomenologically based models in common use today for opaque surfaces. We then return to some common approximations of transmission using compositing instead of BSDFs.

14.9.1 Scattering Functions (BSDFs)

Scattering can be described by a function $(P, \omega_i, \omega_o) \mapsto f_s(P, \omega_i, \omega_o)$ that represents the probability density of light propagating in direction $-\omega_i$ scattering to direction ω_o when it strikes the surface at point P (see Figure 14.25). In general, a "brighter" or more reflective diffuse surface will have higher values of $f_s()$. (The precise definition of f_s is given in Chapter 26.)

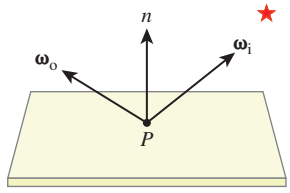


Figure 14.25: The vector ω_i points toward *the light source (represented by the star), so light propagates in direction $-\omega_i$. The light scatters at P and leaves in various directions ω_o. The value $f_s(P, \omega_i, \omega_o)$ measures the scattering.*

In writing $f_s(P, \omega_i, \omega_o)$, we are introducing notation that we'll use throughout the discussion of rendering in the remainder of the book. The function f_s will always represent scattering. P will often represent a point of some surface at which we're computing scattering, ω_i is the direction from P to the source of light arriving at P (thus, the light travels in direction $-\omega_i$), and ω_o will be the direction in which light leaves P.

The use of f_s is a mathematical convenience. In our programs, f_s is typically defined in terms of some "basic" scattering function f defined by how it scatters light from a surface in the xz-plane whose outward normal vector is in the positive-y direction. As an example, a surface that preferentially scatters light in the normal direction could be modeled by

$$f(k, \omega_i, \omega_o) = \begin{cases} 0 & \text{if } \omega_i \text{ or } \omega_o \text{ points in the } -y \text{ halfplane} \\ k \cdot \left(\omega_o \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \right)^2 & \text{otherwise} \end{cases} \tag{14.14}$$

$$= k \cdot \max(\omega_o \cdot \vec{y}, 0)^2$$

where k is a number between 0 and 1 describing how reflective the surface is (for this simple case, it reflects all wavelengths equally well). The function f is large when ω_o is near the y-direction and small when it's near the xz-plane. When we want to use f to represent the scattering from a surface that's not oriented with its normal vector in the positive-y direction, we write f_s so that it first transforms ω_i

and ω_o into a new coordinate frame in which the surface normal at P is the second basis vector, and then apply f to these transformed vectors. Similarly, we might want to use f to represent a surface that's "blotchy"—it's more reflective in some places than others. We can do so by varying the value k as we move across the surface. The code for f_s then has the structure shown in Listing 14.5.

Listing 14.5: Evaluating f_s via a basic function f.

```
1 fs(P, wi, wo)
2   k = getReflectivity(P)
3   b1,b2,b3 = getBasis(P)
4   wiLocal = wi written in the b basis
5   woLocal = wo written in the b basis
6   return f(k, wiLocal, woLocal)
```

Most objects have varying appearance over their surface, like our blotchy sphere does. That is to say, f_s varies over the surface of objects, not only because of the orientation of the surface, but because of varying surface properties. But the variation usually happens in the form just described: Some tool like texture mapping is used to determine the variation in the parameters that we want to pass to the basic scattering function f.

It would be awkward to work with a program in which there was actually a single BSDF for the entire scene that took a scene point as an argument and chose parameters based on it. For modularity, we want to have different BSDFs and attach them to surfaces freely. That is, "BSDF" should be a programmatic interface (i.e., type), and specific BSDFs such as those for glass and wood can be implementations of that interface. The spatial variation within a single logical material still presents a problem. That variation is typically parameterized in the surface's own reference frame since the variation should transform with the object, appearing "painted on" the surface rather than projected through space onto it.

Two natural choices present themselves. One choice is to represent the BSDF for a single, small patch that is itself homogeneous. This pushes the problem of finding the local parameter variation back into the part of the program that sampled the surface location; for example, the ray-casting engine. In this case, we have `fs(wi, wo)` as the BSDF evaluation function.

The other choice is to represent the BSDF for an entire material with spatial variation and explicitly specify the point to be sampled *in the material's own space*; for example, using texture coordinates. In this case, we have `fs(u, v, wi, wo)` as the BSDF evaluation function (although perhaps, since it's a different function, we should use a name other than `fs` for it).

Neither choice is obviously superior; which to use depends on the constraints and design of the surface-sampling machinery. A similar choice can be made for the space in which to express the direction vectors. Thus far, we've followed the mathematical convention of assuming that ω_i and ω_o are in world space. However, the BSDF model is usually derived in the surface's tangent space. Expressing the arguments in world space thus forces the BSDF to transform the arguments into the tangent space. That transformation may be explicit, or it may be implicit by developing all terms as dot products with the tangent and normal vectors. This also forces our "BSDF" representation to be aware of the local orientation of the surface—to be instantiated anew every time a point is sampled from the scene.

In this chapter, we choose to represent a sample of a surface rather than a BSDF. This means that a surface element encodes a position, reference frame, and any spatially varying parameters of the BSDF, as well as the BSDF itself. We favor this representation because it allows separating the ray-surface sampling and scattering portions of a renderer. That separation has pedagogical benefits because it allows us to consider the pieces of a renderer separately. It also has design benefits because the pieces become modular and we can mix different surface and scattering sampling methods. We do not consider the efficiency implications of this decision here, but note that it is used in several rendering libraries, such as PBRT (http://pbrt.org) and The G3D Innovation Engine (http://g3d.sf.net).

In practice, there are two different operations that f_s must support. The first is direct evaluation: Given two directions, we wish to evaluate the function. This is used for direct illumination, where we have already chosen a light-transport path and wish to know the magnitude of the transport along it. The second is sampling. In this case, we are given either the incoming or the outgoing light direction and wish to choose the other direction with probability density proportional to f_s, possibly weighted by projected area along one of the vectors.

For both direct evaluation and sampling, scattering such as by a mirror or lens that does not diffuse light and reflects or transmits a perfect image must be handled separately. The function f_s "takes on infinite values" at the directions corresponding to reflection or transmission, which we call **impulses.** So we divide most operations into separate methods for the finite and impulse aspects.

✓ **THE API PRINCIPLE:** Design APIs from the perspective of the programmer who will use them, not of the programmer who will implement them or the mathematical notation used in their derivation. For example, a single BSDF $f(\omega_i, \omega_o)$ mapped to a function API `Color3 bsdf(Vector3 wi, Vector3 wo)` is easy to implement but hard to use in a real renderer.

Listing 14.6 gives an interface for evaluating the finite part of f. This method abstracts the algorithm typically employed for direct illumination in a pixel shader or ray tracer. It is relatively straightforward to implement.

Listing 14.6: An interface for a scattering function's direct evaluation (similar to `G3D::Surfel`).

```
1  class BSDF {
2  protected:
3      CFrame cframe; // coordinate frame in which BSDF is expressed
4
5      ...
6
7  public:
8
9      class Impulse {
10     public:
11         Vector3     direction;
12         Color3      magnitude;
13     };
14
```

```
    typedef std::vector<Impulse> ImpulseArray;

    virtual ~BSDF() {}

    /** Evaluates the finite portion of f(wi, wo) at a surface
      whose normal is n. */
    virtual Color3 evaluateFiniteScatteringDensity
    (const Vector3&    wi,
     const Vector3&    wo) const = 0;

    ...
```

Listing 14.7 is an interface for the remaining methods needed for algorithms like photon mapping, recursive (Whitted) ray tracing, and path tracing. These are the methods for which the implementation and underlying mathematics are somewhat more complicated. We will not discuss them further here, except to note that the scattering methods are still straightforward to implement, given both the finite scattering density and the impulses, if we are willing to use rather inefficient implementations. There is nothing sacred about the particular methods we've included in this interface. In some implementations of path tracing, for instance, we want to sample with respect to a distribution proportional to the BSDF, without the extra weighting factor of $\omega_i \cdot \mathbf{n}$, and we might include a method for that in our interface.

Listing 14.7: An interface for a scattering function's scattering and impulse methods.

```
class BSDF {
    ...

    /** Given wi, returns all wo directions that yield impulses in
        f(wi, wo). Overwrites the impulseArray. */
    virtual void getOutgoingImpulses
    (const Vector3&    wi,
     ImpulseArray&     impulseArray) const = 0;


    /** Given wi, samples wo from the normalized PDF of
        wo -> g(wi, wo) * |wi . n|,
        where the shape of g is ideally close to that of f. */
    virtual Vector3 scatterOut
    (const Vector3&   wi,
     Color3&          weight) const = 0;


    /** Given wi, returns the probability of scattering
        (vs. absorption). By default, this is computed by sampling
        since analytic forms do not exist for many scattering models. */
    virtual Color3 probabilityOfScatteringOut(
                      const Vector3& wi) const;

    /** Given wo, returns all impulses for wi. */
    virtual void getIncomingImpulses
    (const Vector3&    wo,
     ImpulseArray&     impulseArray) const = 0;


```

```
31      /** Given wo, samples wi from the normalized PDF of wi -> g(wi, wo) * |wi . n|. */
32      virtual Vector3 scatterIn
33      (const Vector3&     wo,
34       Color3&            weight) const = 0;
35
36
37      /** Given wo, returns the a priori probability of scattering (vs. absorption) */
38      virtual Color3 probabilityOfScatteringIn(const Vector3& wo) const = 0;
39
40  };
```

There are two sources for BSDF implementations. **Measured BSDFs** are constructed from thousands or millions of controlled measurements of a real surface. Measurement is expensive (or tricky to perform oneself), but it provides great physical realism. The data describing the BSDF is typically large but generally smooth, and thus amenable to compression.

Analytic BSDFs describe the surface appearance in terms of physically or aesthetically meaningful parameters. They are usually expressed as sums and products of simple functions that are zero for most arguments and rise in a smooth lobe over a narrow region of the parameter space. Those analytic BSDFs that model the underlying physics can be used predictively. We now describe some simple yet popular analytic BSDFs.

14.9.2 Lambertian

Lambert observed that most flat, rough surfaces reflect light energy proportional to the cosine of the angle between their surface normal and the direction of the incoming light. This is known as **Lambert's Law.** It follows from geometry for surfaces with a constant BSDF because the projected area of the surface is proportional to the cosine of the incoming-light angle. A constant BSDF is named Lambertian because it follows this law.

Although few surfaces exhibit truly Lambertian reflectance, most insulators can be recognizably approximated by a Lambertian BSDF. The residual error is then addressed by adding other terms, as described in the following subsection.

Examples of nearly Lambertian surfaces are a wall painted with flat (i.e., matte) paint, dry dirt, and skin and cloth observed from several meters away. The primary error in approximating these as Lambertian is that they tend to appear shinier than predicted by a constant BSDF when observed at a glancing angle.

In practice, the approximately Lambertian appearance usually arises because the surface is somewhat permeable to light at a very shallow level and all directionality is lost by the time light emerges. Glossy highlights are caused by light preferentially reflecting close to the mirror-reflection direction. When that does not happen, the surface appears matte.

Listing 14.8 implements a Lambertian BSDF's evaluate method. We specify a single "Lambertian constant" k_L for each frequency band, that is, a `Color3`. The components of k_L must each be in the range [0, 1]. They represent the reflectivity of the surface to each "color" of light. Larger values are brighter, so (1, 0, 0) appears bright red and (0.2, 0.4, 0.0) is a dark brown. Of course, few real surfaces truly have perfect absorption or perfect reflectance along any color channel. Many physically based rendering systems also tend to risk dividing by zero if any color channel is at either limit, so it is a good idea to select constants on the open interval (0, 1) in practice.

Listing 14.8: The finite direct evaluation portion of a simple Lambertian BSDF for surfaces such as walls covered in matte paint.

```
class LambertianBSDF : public BSDF {
private:
    // Each element on [0, 1]
    Color3 k_L;

public:
    virtual Color3 evaluateFiniteScatteringDensity
    (const Vector3&    wi,
     const Vector3&    wo) const {
      if ((wi.dot(cframe.rotation.getColumn(1)) > 0) &&
          (wo.dot(cframe.rotation.getColumn(1)) > 0)) {
        return k_L / PI;
      }
      else {
        return Color3::zero();
      }
     }
    ...
};
```

Note that there is no projected area factor in $f_s(P, \omega_i, \omega_o) = k_L/\pi$. That geometric factor must be accounted for by the renderer, as shown in Listing 14.11. We divide k_L by π because the integral of the BSDF times the cosine of the angle of incidence must be less than one over the entire hemisphere above a planar surface to ensure energy conservation, and $\int_{\mathbf{S}^2_+} (\omega \cdot \hat{\mathbf{z}})\, d\omega = \pi$.

Because Lambertian appearance arises from well-diffused light, Lambertian reflectance is also called diffuse or perfectly diffuse reflection. We use the term "diffuse" to describe all nonspecular behavior.

14.9.3 Normalized Blinn-Phong

Phong introduced the phenomenological shading model [Pho75] described in Chapter 6. His model describes a surface that exhibits Lambertian reflection with a glossy highlight of adjustable sharpness.

The original Phong model has been reformulated as a BSDF and then extended by many practitioners. The currently preferred form remains phenomenologically based but has some basic properties that are desirable in a scattering model; for example, it conserves energy and obeys the projected area property. For a richer explanation of scattering models see Chapter 27. We simply present the model here in a form suitable for implementation.

This is the modern formalization, in terms of physical units, of the model that was described in Chapter 6. The replacement of C_d and C_s with k_L and k_g is appropriate for three reasons. First, "Lambertian" is a more specific name for the shape of the "diffuse" distribution; anything that isn't an impulse is "diffuse," but Phong prescribes a specific and geometrically well-founded Lambertian distribution for that term. Second, we've defined "specular" as a technical term for a mirror impulse, following its English definition and physics terminology, reserving "glossy" to denote reflection that's somewhat or very concentrated in a particular direction. Third, this formulation is different from the original in both parameters and form. These k parameters are no longer potentially ambiguous RGB triples

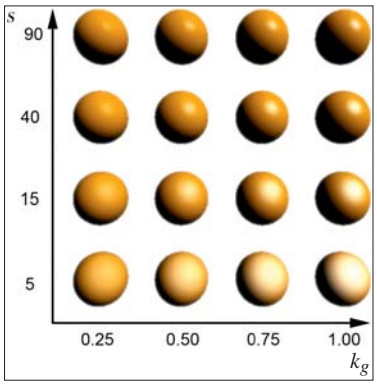


Figure 14.26: Sphere rendered with a single light source, using a Phong BSDF with a white k_g and orange k_L. k_g increases to the right and s increases upward. (Credit: From Creating Games: mechanics, content, and technology *by McGuire, Morgan and Jenkins, Odest Chadwicke © 2009. Reproduced with permission of Taylor & Francis Group LLC - Books in the formats other book and textbook via Copyright Clearance Center)*

(each 0 . . . 1), but constants representing the net probability over all directions of that term contributing to scattering.

The specific variant in Listing 14.9 includes the adjusted highlight term introduced by Blinn, the (implicit) projected area factor demanded by physics, and an approximate normalization factor introduced by Sloan and Hoffman [AMHH08] for energy conservation. Figure 14.26 shows the impact of varying the two glossy parameters of glossy coefficient and smoothness.

Listing 14.9: Normalized Blinn-Phong BSDF without Fresnel coefficients, based on the implementation from Real-Time Rendering [AMHH08].

```
class PhongBSDF {
private:
    // For energy conservation, ensure that k_L + k_g < 1 on each color channel
    Color3 k_L;
    Color3 k_g;

    // "Smoothness" parameter; the exponent on the half-vector dot
    // product.
    float s;

public:

    virtual Color3 evaluateFiniteScatteringDensity
    (const Vector3&    wi,
     const Vector3&    wo) const {

      if ((wi.dot(cframe.rotation.getColumn(1)) <= 0) &&
          (wo.dot(cframe.rotation.getColumn(1)) <= 0)) {
        return Color3::zero();
      }
      const Vector3& w_h = (w_i + w_o).direction();
      return k_L / PI + k_g * (8 + s) / (8 * PI) * pow(max(0.0, n.dot(w_h)), s);
    }

    ...
};
```

The Phong BSDF has three parameters. The Lambertian constant k_L controls the color and intensity of matte reflection. The analogous k_g controls the color and intensity of glossy reflection, which includes highlights produced by glossy reflection of bright light sources. A perfectly smooth reflective surface has a mirrorlike appearance. Rougher surfaces diffuse the mirror image, which produces the glossy appearance. The term including k_g produces a teardrop-shaped lobe near the mirror-reflection direction when f_s is graphed, so k_g is often referred to as the magnitude of the glossy (or specular) lobe.

The smoothness parameter s describes how smooth the surface is, on an arbitrary scale. Low numbers, like $s = 60$, produce fairly broad highlights. This is a good model for surfaces like leather, finished wood, and dull plastics. High numbers, like $s = 2000$, produce sharper reflections. This is a better model for car paint, glazed ceramics, and metals.

The scale of s is not perceptually linear. For example, $s = 120$ does not produce highlights that have half the extent of $s = 60$ ones. It is therefore a good idea to expose a perceptual "shininess" parameter $\sigma \in [0, 1]$ to artists and map it to s with a function such as $s = 8192^{(1-\sigma)}$.

Most insulators exhibit colorless highlights, so k_g is typically chosen to either be constant across color channels or have a hue opposite k_L in order to sum to a gray or white appearance. Metals tend to have nearly zero Lambertian reflectance and a k_g that matches the perceived color of the metal; examples include gold, copper, silver, and brass.

The normalization factor $(8 + s)/(8\pi)$ increases the intensity of highlights as they grow sharper. This makes s and k_G somewhat perceptually orthogonal and makes the energy conservation constraint simply $k_L + k_g \leq 1$. The "8"s appear from rounding the constants in the true solution for the integral of the glossy term over the hemisphere to the nearest integer.

14.10 Translucency and Blending

We say that an object or medium is translucent when we can "see through it," such as with glass, fog, or a window screen. For that to happen, some light from beyond the object must be able to pass through it to reach our eyes.

The phenomenon of **translucency** occurs when multiple scene locations directly contribute to the energy at a point in screen space. Under the ray optics modeled in this chapter, light rays do not interact with one another. For example, two flashlight beams pass through each other. Because they don't interact, we can consider the energy contribution from each light ray independently. We then sum the contribution of all rays to a point. The property of light that describes this behavior (at least macroscopically) is called **superposition.** This property is what allows us to consider different wavelengths (colors) independently as well as describe light scattering for individual rays yet render all the light in a scene.

As with any other scene point, the incoming energy at a point on the image plane may arrive from multiple locations. The camera aperture blocks a majority of incoming directions, and in the limiting case of a pinhole camera, it blocks all but a single direction. In that case, a single ray exiting the virtual camera describes the path (albeit backward) along which light must have arrived. Yet in the presence of translucent surfaces, there may be multiple scene points along that eye ray that contribute because those points need not fully obscure the light coming from beyond them.

Given our model of surfaces, all light that passes through a surface to reach the camera is, by definition, indirect illumination. In other words, we can still render a single surface at each screen-space point. We just allow some light to scatter from behind the surface to in front of it. For a material like green glass, the scattering may "color" the outgoing light by transmitting some frequencies more than others.

Transmission of many kinds can naturally be represented by the BSDF models that we've already discussed. Yet those models are too computationally expensive for current real-time rendering systems. Just as was the case for scattering and surface models, it is common to intentionally introduce both approximations and a more complicated model for transmission to gain both expressive control and improved performance. The common approximation to translucency phenomena is to render individual surfaces in back-to-front order and then compose them by **blending,** a process in which the various colors are combined with weights. The blending functions are arbitrary operators that we seek to employ to create phenomena that resemble those arising from translucency. In general, this model forgoes diffusion and refraction effects in order to operate in parallel at each pixel, although it is certainly possible to include those effects via screen-space sampling (e.g., [Wym05]) or simply using a ray-tracing algorithm. Most graphics APIs include entry points for controlling the blending operation applied as each surface is rendered. For example, in OpenGL these functions are `glBlendFunc` and `glBlendEquation`. We give examples of applying these in specific contexts below.

There are multiple distinct causes for translucency. Distinguishing among them is important for both artistic control and physical accuracy of rendering (either of which may not be important in a particular application). Because all reduce to some kind of blending, there is a risk of conflating them in implementation. The human visual system is sensitive to the presence of translucency but not always to the cause of it, which means that this sort of error can go unnoticed for some time. However, it often leads to unsatisfying results in the long run because one loses independent control over different phenomena. Some symptoms of such errors are overbright pixels where objects overlap, strangely absent or miscolored shadows, and pixels with the wrong hue.

To help make clear how blending can correctly model various phenomena, in this section we give specific examples of applying a blending control similar to OpenGL's `glBlendFunc`. The complete specification of OpenGL blending is beyond what is required here, changes with API version, and is tailored to the details of OpenGL and current GPU architecture. To separate the common concept from these specifics, we define a specific blending function that uses only a subset of the functionality.

If you are already familiar with OpenGL and "alpha," then please read this section with extra care, since it may look deceptively familiar. We seek to tease apart distinct physical ideas that you may have previously seen combined by a single implementation. The following text extends a synopsis originally prepared by McGuire and Enderton [ME11].

14.10.1 Blending

Assume that a *destination* sample (e.g., one pixel of an accumulation-buffer image; see Chapter 36) is to be updated by the contribution of some new *source* sample. These samples may be chosen by rasterization, ray tracing, or any other sampling method, and they correspond to a specific single location in screen space.

Let both source and destination values be functions of frequency, represented by the color channel c. For each color channel c, let the new destination value d'_c be

$$d'_c = \sigma_c(s,d) \cdot s_c + \delta_c(s,d) \cdot d_c, \tag{14.15}$$

where δ and σ are functions that compute the contributions of the old destination value d and the source value s. Let `BlendFunc(senum, denum)` select the implementation of the δ and σ functions. To enable optimization of common cases in the underlying renderer, APIs generally limit the choice of σ and δ to a small set of simple functions. Hence the arguments to `BlendFunc` are enumerated types rather than the functions themselves. For generality, let `senum` and `denum` have the same type.

A partial list of the blending function enumerants and the functions to which they correspond (which we'll extend a bit later) is:

`ONE`:	$b_c(s,d) = 1$
`ZERO`:	$b_c(s,d) = 0$
`SRC_COLOR`:	$b_c(s,d) = s_c$
`DST_COLOR`:	$b_c(s,d) = d_c$
`ONE_MINUS_SRC_COLOR`:	$b_c(s,d) = 1 - s_c$
`ONE_MINUS_DST_COLOR`:	$b_c(s,d) = d_c$

To make the application of `BlendFunc` clear, we now examine two trivial cases in a common scene. There exist alternative and more efficient methods for achieving these specific cases in OpenGL than what we describe here, specifically the blending enable bit and write mask bits, but we describe the general solution to motivate blending functionality.

Consider a static scene containing a wall covered in red latex paint, a pinhole camera, and a thin, flat blue plastic star that is suspended between the wall and the camera, occupying about half of the image in projection. Let these objects be in a vacuum so that we need not consider the impact of a potentially participating medium such as air. Now consider a sample location near the center of the star's projection.

Blue plastic is reflective, so light incident on the star at the corresponding point within the scene is either absorbed or reflected. Assume that we have somehow computed the incident light and the reflective scattering. Let s describe the radiance reflected toward the screen-space sample location. If we are in the midst of rendering, then the image that we are computing may already have some existing value d at this location, either the value with which it was initialized (say, $d_c = 0$ $\mathrm{W}/(\mathrm{sr\ m}^2)$) or perhaps some "red" value if the wall was rendered into the image first.

Any light transported from the background wall along the ray through our sample and the camera's pinhole must necessarily be blocked by the blue star. So we don't care what the preexisting value in d is. We simply want to overwrite it. For this case, we select `BlendFunc(ONE, ZERO)`, yielding the net result

$$\begin{aligned} d'_c &= 1.0 \cdot s_c + 0.0 \cdot d_c && (14.16)\\ &= s_c. && (14.17) \end{aligned}$$

This is not very exciting, and it seems like a silly way of specifying that the new value overwrites the existing one. It makes a little more sense in the context of

a hardware implementation of the blending function. In that case, there is some arithmetic unit tasked with updating the frame buffer with the new value. The unit must always be semantically configured to perform some function, however trivial.

Say that we're rendering the scene by rasterization. Rasterization is just a way of iterating over screen-space sample locations. We typically choose to iterate over the samples arising from the projection of object boundaries (i.e., surfaces). However, one can also choose to rasterize geometry that does not correspond to an object boundary as a way of touching arbitrary samples. For example, deferred shading typically rasterizes a bounding volume around a light source as a conservative method for identifying scene locations that may receive significant direct illumination from the source. If rasterizing some such volume that lies entirely within the vacuum, how do we blend the resultant contribution? One method is `BlendFunc(ZERO, ONE)`, yielding the net result

$$d'_c = 0.0 \cdot s_c + 1.0 \cdot d_c \tag{14.18}$$
$$= d_c, \tag{14.19}$$

which allows the preexisting value in the image to remain. Here, the rasterized surface is perfectly **transparent,** meaning that it is truly invisible to light. Why bother rasterizing when we'll just discard the source color? One answer is that there are more attributes than just radiance stored at a pixel. One might want to mark an area of the depth or stencil buffer without affecting the image itself. This occurs, for example, when implementing stenciled shadow volumes by rasterization. Another answer is that we may want to change the blending weights per-sample to selectively discard some of them, as discussed in Section 14.10.2.

14.10.2 Partial Coverage (α)

Let us return to the scene containing a thin blue star floating in front of a red wall, introduced in Section 14.10.1. One way to model the blue star is with a single two-sided rectangle and a function defined on the rectangle (say, implemented as a texture map) that is 1 at locations inside the star and 0 outside the star. This function, whose value at a sample is often denoted α, describes how the star covers the background.

The coverage in this case is associated with the sample of the *source object* that is being rendered, so we should denote it α_s. An implementation likely contains a class for representing radiance samples at three visible frequencies (red, green, and blue) and a coverage value as

```
1 class Color4 {
2    float r;
3    float g;
4    float b;
5    float a;
6 };
```

To leverage the concept of coverage as a way of masking transparent parts of the rectangle, we introduce two new blending enumerants:

`SRC_ALPHA`: $b_c(s,d) = s_\alpha$
`ONE_MINUS_SRC_ALPHA`: $b_c(s,d) = 1 - s_\alpha$

The blending mode `BlendFunc(SRC_ALPHA, ONE_MINUS_SRC_ALPHA)` yields

$$d'_c = s_\alpha \cdot s_c + (1 - s_\alpha) \cdot d_c, \tag{14.20}$$

which is linear interpolation by s_α. Our coverage value at each point is either 0 or 1, as befits the physical model of the star: At every point sample the rectangle enclosing the star is either completely opaque and blocks all light from the background, or completely transparent and allows the background to be seen.

Rendering under such a binary coverage scheme with a single sample per pixel will generate aliasing, where the edges of the star appear as stair steps in the image. If we increase the number of samples per pixel, we will obtain a better estimate of the fraction of the star that covers an individual pixel. For example, in Figure 14.27, the outlined pixel is about 50% covered by the star and 50% covered by the background. However, the estimate would be poor if we used three samples instead of four, and even the four samples from the diagram can produce an error of $\pm 12.5\%$ coverage for pixels with less even coverage. Taking many samples per pixel is of course an expensive way to evaluate partial coverage by analytically defined shapes.

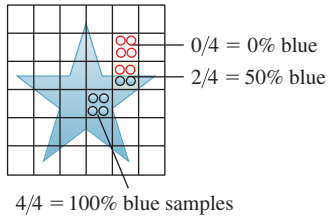


Figure 14.27: An ideal blue vector star shape rasterized on a low-resolution pixel grid. The boxes of the grid are pixels. The circles represent samples at which we are computing coverage.

We've seen this problem before, with texture maps encoding reflectance. The MIP-mapping solution developed in Chapter 20 works for coverage as well as reflectance. Imagine a prefiltered coverage map for the star rectangle in which the outlined pixel in Figure 14.27 is a single **texel** (one pixel in a texture-map image). Its coverage value is the integral of binary coverage over that texel, which is $s_\alpha = 0.3$. This integral is called **partial coverage.** Equation 14.20 holds for partial coverage as well as binary coverage; in this case it is called the **over operator** because it represents the image of a partially covering s lying over the background d.

Order matters, however. The over operator assumes that we're rendering surfaces in back to front order (the Painter's Algorithm described in Section 36.4.1) so that we always composite nearer objects *over* farther ones.

Note that s_α encodes the fraction of coverage, but not the locations of that coverage within the texel. One interpretation of s_α is that it is the probability that a sample chosen uniformly at random within the texel will hit the opaque part of the rectangle instead of the transparent part. For the star, the probability remains at the extremes of 0 and 1 except at texels along the edge. For other shapes, nearly every texel encodes some kind of edge. Consider a screen door, for example. We might paint a texture that has s_α as 1 and 0 in alternating rows and columns at the highest resolution. Such a texture has maximum spatial frequencies, so it could produce significant aliasing. However, after a single MIP level, the texture contains entirely fractional values.

An advantage of the probabilistic interpretation of partial coverage is that it allows us to describe the result of successive applications of blending for different surfaces without explicitly representing the high-frequency coverage mask in the result image d'. For example, we can render the back wall as viewed through two identical screen doors by computing the final destination color d''_c for each channel $c \in \mathrm{r}, \mathrm{g}, \mathrm{b}$ as

$$d'_c = s_\alpha \cdot s_c + (1 - s_\alpha) \cdot d_c \tag{14.21}$$

$$d''_c = s_\alpha \cdot s_c + (1 - s_\alpha) \cdot d'_c \tag{14.22}$$

$$= (1 - s_\alpha) s_\alpha \cdot s_c + (1 - s_\alpha)^2 \cdot d'_c. \tag{14.23}$$

The pitfall of this interpretation is that it assumes statistical independence between the subpixel coverage locations for each s-layer. If the two doors are perfectly aligned (assume a parallel projection to make this easy), then this assumption does not hold because the second door provides no new occlusion of the background. In this case, the second door is precisely behind the first and is invisible to the camera. Therefore, we should have obtained the result $d_c'' = d_c'$, and not the one from Equation 14.23.

Equation 14.23 contains the result we expect on average; that is, if the locations covered by the doors are statistically independent. This is generally what one wants, but if there is some underlying reason that the coverage should be correlated between surfaces, then Equation 14.23 will give incorrect results. For example, when the s_α value results from thin-line rasterization, many thin lines may naturally align in screen space (say, the support cables on a suspension bridge) and yield an incorrect result.

The classic paper by Porter and Duff [PD84] that canonicalized blending carefully analyzes all coverage cases and is specific about the statistical independence issue. Yet it is very easy to implement incorrectly. For example, the OpenGL 3.0 and DirectX 10 APIs contain an **alpha-to-coverage** feature that converts s_α back to a binary visibility mask when placing multiple samples within a pixel. That feature yields incorrect compositing results, as it is specified, because the mask contains a fixed pattern based on the s_α value. Thus, the subpixel locations covered by two surfaces with equal, fractional α are always perfectly correlated in those APIs. This tends to give undesirable results for overlapping translucent surfaces. Enderton et al. [ESSL11] describe the problem and one solution: Choose the coverage based on a hash of the depth value and screen-space position.

We've discussed the coverage of the surface, but what about the coverage of d_α', the result image? Consider a case where we are rendering an image of the contribution of all surfaces between a depth of 1 m and 2 m from the camera. We would then like to composite this over another image containing the result for objects at 2 m and farther. In that case, some pixels in our close image may be completely transparent, and others may have partial or complete coverage. If we again assume statistical independence of subpixel locations covered by different surfaces, we can composite coverage itself by

$$d_\alpha' = s_\alpha + d_\alpha \cdot (1 - s_\alpha), \tag{14.24}$$

thus creating a composite value d that itself acts like a surface with partial coverage.

14.10.2.1 Premultiplied α

Note that in the preceding section, s_c never appeared in isolation. Instead, it was always modulated by s_α. Our interpretation of this was that s is a surface with α coverage of a screen-space area, and that the covered parts had color s_c, or more formally, emitted and scattered radiance s_c toward the viewer. The net contribution from s is thus $s_c s_\alpha$.

It is common practice to store colors with **premultiplied alpha,** in the form $(s_\mathrm{r} s_\alpha,\ s_\mathrm{g} s_\alpha,\ s_\mathrm{b} s_\alpha,\ s_\alpha)$. This has several advantages. For example, it saves a few multiplication operations during compositing and resolves the ambiguity in the meaning of s_c for a surface with $s_\alpha = 0$. The latter point becomes significant in

the (underdetermined) image processing problem of **matting,** where an algorithm tries to recover $s_c s_\alpha$, s_α, and d_c from the composite d'_c.

14.10.3 Transmission

Partial coverage was a model that allowed us to describe fine but macroscopic structures like lace or a window screen using simple geometric shapes and a statistical measure of coverage. In that case, the covered parts of the surface were completely opaque and the uncovered parts transmitted all light because they were filled by the surrounding medium, such as air.

If we ignore refraction (the phenomenon of light bending when it enters a new medium), we can extend partial coverage to microscopic structures. Consider an extremely thin pane of colorless glass. An incident light ray will either strike a glass molecule on the surface and reflect or be absorbed, or pass through the empty space between molecules (this model is physically simplistic, but phenomenologically viable). We can let α represent the coverage of space by glass molecules and render the glass using the partial coverage model. This is in fact done frequently, although adding a bit more sophistication to our model allows us to remove the extremely thin and colorless models to better describe a range of transmissive media.

Green glass appears green because it transmits green light. If you hold a piece of green glass over a black background, then it appears mostly black because it reflects little green light. If we continue with the microscopic partial coverage model, then $s_\mathrm{g} \approx 0$ for the glass. The green glass in fact reflects little light at *any* frequency, so $s_\mathrm{r} \approx s_\mathrm{g} \approx s_\mathrm{b} = 0$. We can't describe the appearance of green glass over a white surface using a single coverage value α, because the coverage must be large for red and blue light (so that they are blocked) and low for green light (so that it transmits). We need to extend our coverage representation. Let s_c be the color of light reflected or emitted at the surface near frequency c, and $1 - t_c$ be the microscopic coverage of frequency c by the surface, that is, t is the amount of light transmitted. We retain the s_α value for representing macroscopic partial coverage by a transmissive medium. We can now express the composition of the surface over the background by holding out the background by $1 - t$ and then adding the contribution due to s.

To implement this in code, we use the `SRC_COLOR` enumerant to selectively block light from the background and then make a second pass to add the contribution from the surface:

```
1  // Selectively block light from the background
2  // where there is coverage
3  SetColor(t * s.a + (1 - s.a));
4  BlendFunc(ZERO, SRC_COLOR);
5  DrawSurface();
6
7  // Add in any contribution from the surface itself,
8  // held out by its own coverage.
9  SetColor(s);
10 BlendFunc(SRC_ALPHA, ONE);
```

Note that this example implements transmission by a thin surface that may itself have only partial macroscopic coverage. In the case where that coverage is complete and the surface itself scatters no light, the entire example reduces to simply:

```
SetColor(t);
BlendFunc(ZERO, SRC_COLOR);
DrawSurface();
```

We've presented this in a form similar to the OpenGL API for real-time rasterization rendering. The mathematics can be applied per-pixel in another rendering framework, such as a ray tracer. Doing so is common practice, although we suggest that if you've already written a ray tracer with well-structured ray-scattering code, then it may be trivial to implement much more accurate transmission with a BSDF than with blending. If you choose to employ the blending model, the code might look something like:

```
Radiance3 shade(Vector3 dirToEye, Point3 P, Color3 t, Color4 s, ...) {
    Radiance3 d;
    if (bsdf has transparency) {
      // Continue the ray out the back of the surface
      d = rayTrace(Ray(P - dirToEye * epsilon, -dirToEye));
    }

    Radiance3 c = directIllumination(P, dirToEye, s.rgb, ...);

    // Perform the blending of this surface's color and the background
    return c * s.alpha + d * (t * s.alpha + 1 - s.alpha);
}
```

Our blending model for transmission at this point supports frequency-varying (colored) transmission and a distinct color scattered by or emitted at the surface. Yet it still assumes an infinitely thin object so that transmission can be computed once at the surface. For an object with nonzero thickness, light should continue to be absorbed within the material so that thicker objects transmit less light than thinner ones of the same material.

Consider the case of two thin objects held together, assuming $s_\alpha = 1$ macroscopic coverage and $1-t_{\text{rgb}}$ microscopic coverage, that is, transmission. We expect the first object to transmit t of the light from the background: $d' = td$; and the second to transmit t *of that*, for a $d'' = t^2 d$ net contribution from the background, as in our previous double-compositing example of macroscopic partial coverage. Now consider the case of three such thin objects; the net light transmitted will be $t^3 d$. Following this pattern, a thick object composed of n thin objects will transmit $t^n d$. The absorption of light is thus exponential in distance, as we suggested in Equation 14.13.

We can still apply the simple compositing model if we precompute an effective net transmission coefficient t for the thick object based on the distance x that light will travel in the medium, that is, its cross section along the ray. Three common methods for computing this thickness are tracing a ray (even within the context of a rasterization algorithm), rendering multiple depth buffers so that the front and back surfaces of the object are both known (e.g., [BCL+07]), or simply assuming a constant thickness. It is common to express the rate of absorption by a constant k. The net transmission by the thick object along the ray is thus $t = e^{-kx}$. The thin-blending model can then be applied with this constant. This exponential falloff is a fairly accurate model and k can be computed from first principles; however, given the rest of the rendering structure that we've assumed in this section, it is more likely to be chosen aesthetically in practice.

14.10.4 Emission

It is often desirable to render objects that appear to glow without actually illuminating other surfaces. For example, car tail lights and the LEDs on computer equipment likely contribute negligible illumination to the rest of the scene but themselves need to appear bright in an image. These effects may be simulated by rendering the scene normally and then additively blending the emissive component as if it were a new surface rendered using `BlendFunc(ONE, ONE)`.

Some particularly attractive effects are due to such emission by a medium that is itself seemingly transparent. Examples include the light from a neon bulb, lightning, science fiction "force fields," and fantasy magical effects. That the underlying surface is invisible in these cases is irrelevant—the additive blending of the emissive component is unchanged.

14.10.5 Bloom and Lens Flare

Lens flare and bloom are effects that occur within the optical path of a real camera. One could model the real optical path, but to merely achieve the phenomena it is much more effective to additively blend contributions due to additional geometry over the rendered frame using `BlendFunc(ONE, ONE)`. Bloom simulates the diffusion of incident light within lenses and the saturation of the sensor. It is typically simulated by blurring only the brightest locations on-screen and adding their contribution back into the frame. Lens flare arises from multiple reflections between lenses within the objective. It is typically simulated by rendering a sequence of iris (e.g., hexagonal or disk) -shaped polygons along a 2D line through bright locations, such as the sun, on-screen.

14.11 Luminaire Models

A computer graphics **luminaire** is a source of light. The luminaires we encounter in daily life vary radically in the spectra of light that they emit, their surface areas, and their intensities. For example, the sun is large and distant, a spotlight is bright and small, and a traffic light is relatively dim and colored. The luminaires we might encounter in a virtual world expand this variation further; for example, a cave of phosphorescent fungus, a magic unicorn's aura, or the navigation lights on a starship.

Before we can present luminaire models, we must first discuss light. You know that light is energy (in photons) that propagates along rays through space and scatters at surfaces. There are many models of light in computer graphics, but they all begin by representing the rate at which energy is passing through a point in space. We give a brief synopsis here and defer extensive coverage until Chapter 26. One *can* render images from the models in this chapter without understanding the motivation and physics behind light transport. However, we recommend that after rendering your first images you read Chapter 26 to build a deeper understanding.

14.11.1 The Radiance Function

Consider a point X in space at which we wish to know the illumination. This is frequently on some surface in the scene, but it need not be. The amount of light incident at X from direction ω is denoted $L(X, \omega)$. This implicitly defines a function L of two variables, X and ω, which is called the **radiance function,** also known as the **plenoptic function.** To be clear, the argument ω denotes the direction of propagation. If there's a photon passing through the point P, traveling in direction ω, then $L(P, \omega) \neq 0$, while it's quite possible that $L(P, -\omega) = 0$. By convention, we'll restrict to the case where ω is a *unit* vector. The units of L are watts per square meter-steradian, $\mathrm{W} \cdot \mathrm{m}^{-2} \cdot \mathrm{sr}^{-1}$. Surface areas are measured in units of square meters, and steradians are the spherical analog of angular measure, called **solid angle.** An angle can measure a 1D region on a 2D unit circle in the plane, and you might express the rate of a quantity passing through that 1D region "per radian." We measure the amount of energy passing through a 2D region (a solid angle) on a unit sphere in 3-space "per steradian."

It is useful to know that radiance is conserved along a ray through empty space. Thus, if we know $L(X, \omega)$, then we also know $L(X + t\omega, \omega)$ for $t > 0$ so long as there is no occluding object within distance t of X along that ray.

14.11.2 Direct and Indirect Light

We distinguish the light that arrives at a point *directly* from a luminaire from light that arrived *indirectly* after reflecting off some surface in the scene. For example, near an outdoor swimming pool, sunlight shines directly on the top of your head, but it also reflects off the water to indirectly strike the bottom of your chin. If there were no indirect light, then the bottom of your chin would appear completely unilluminated. The indirect light arises from interaction between the luminaire and the scene, so we consider it part of the light transport model and not the luminaire model.

14.11.3 Practical and Artistic Considerations

Listing 14.10 gives a typical base class for a set of light-class implementations. Its methods support the practical aspects of incorporating light sources into a renderer, not the physical aspects of light emission.

Listing 14.10: A base class for all light sources, with trivial implementation details omitted.

```
1  /** Base class for light sources */
2  class Light {
3  public:
4      const std::string name() const;
5
6      virtual CoordinateFrame cframe() const;
7
8      /** for turning lights on and off */
9      virtual bool enabled() const;
10
11     /** true for physically-correct lights */
12     virtual bool createsLambertianReflection() const;
13
```

```
14      /** true for physically-correct lights */
15      virtual bool createsGlossyReflection() const;
16
17      /** true for physically-correct lights */
18      virtual bool createsGlobalIllumination() const;
19
20      /** true for physically-correct lights */
21      virtual bool castsShadows() const;
22
23      ////////////////////////////////////////
24      // Direct illumination support
25
26      /** Effective area of this emitter. May be finite,
27        zero, or infinite. */
28      virtual float surfaceArea() = 0;
29
30      /** Select a point uniformly at random on the surface
31          of the emitter in homogeneous coordinates. */
32      virtual Vector4 randomPoint() const = 0;
33
34      /** Biradiance (solid-angle-weighted radiance) at P due
35        to point Q on this light, in W / m^2. Q must be a value
36        previously returned by randomPoint(). */
37      virtual Biradiance3 biradiance
38          (const Vector4& Q, const Point3& P) const = 0;
39
40      ////////////////////////////////////////
41      // Photon emission support
42
43      /** Total power; may be infinite */
44      virtual Power3 totalPower() const = 0;
45
46      /** Returns the position Q, direction of propagation w_o, and
47          normalized spectrum of an emitted photon chosen with
48          probability density proportional to the emission density
49          function for this light. */
50      virtual Color3 emitPhoton(Point3& Q, Vector3& w_o) const = 0;
51  };
```

We assign a reference frame (`cframe`) to each light source. For a light at a finite location, this is the centroid of the emitter and a reference orientation. For infinitely distant sources (i.e., directional sources), this is a reference frame and a convenient location within the scene for displaying GUI affordances to manipulate the source.

14.11.3.1 Nonphysical Tools

It is often useful to manipulate the interaction of lights and the scene in nonphysical ways. These may depart from physics for artistic intent, but they may also be used to compensate for flaws in the rendering model itself. That is, the right model with the wrong data (or vice versa) can't produce the correct image, so sometimes we have to compensate for known limitations and approximations by intentionally violating physics in order to make the net result appear more realistic. The class shown in Figure 14.10 contains several of these tools, in the form of light sources that don't cast shadows, or don't participate in the computation of Lambertian reflection, for instance. Of course, the renderers that use this class must honor such settings for them to have an effect.

Luminaires that do not create glossy reflection (e.g., highlights) provide so-called "fill" or "diffuse" light. These create perceptual cues of three-dimensional shape and softness, approximating global illumination and subsurface scattering. Glossy-only sources create explicit highlights but no other shading. These are useful to model the perceptual cues from **practical lights.** Practical lights are the light sources that appear to be in the scene, as opposed to the invisible ones that are actually lighting most of it. The term comes from film and theatre production; as an example, in a film set of a dining room, very little illumination actually comes from the candles on a table and the scene is more likely to receive illumination primarily from bright off-camera stage lights. Glossy-only sources are particularly useful for creating the perception of windows without dealing with the net impact of those windows on scene light levels. The "Lambertian" and "glossy" reflectance properties properly belong to the surface material, not the light, so using these properties assumes a specific material and shading model.

In the real world, the light from an emitter may experience an unbounded number of scattering events before it is perceived. It is often artistically useful to shine a light on a specific object without incurring the computational cost or global implications of multiple scattering events. A direct, local, or nonglobal light source only scatters light from the first surface it encounters toward the viewer. Beware that the term "local" is also used in lighting models to refer to lights that are at a finite distance from visible parts of the scene.

One may wish to selectively disable shadow casting by lights. This can save the computational cost of computing visibility as well as eliminate shadows that may be visually confusing, such as those cast outward from a carried torch.

14.11.3.2 Applying the Interface to Direct Illumination

The key methods for incorporating the `Light` class into a renderer are `randomPoint` and `radiance`. The `randomPoint` method selects a point on the surface of the emitter uniformly at random with respect to surface area. (The term "selected uniformly at random" is defined precisely in Chapter 30; for now, treat it as meaning "every point is equally likely to be chosen.") Because the point may be infinitely distant from other parts of the scene, we represent the return value as a homogeneous vector. For lights that have varying intensity over their surface, a better choice of interface might be to select the emitter point with probability proportional to the amount of light emitted at that point. Even further elaborations might involve choosing luminaire points in a way that's random but guarantees fairly even distribution of the points and avoids clustering. Stratified sampling, discussed briefly in Chapter 32, is one such method.

The `radiance` method returns the incident radiance at a scene point due to a point on the emitter (which was presumably discovered by calling `randomPoint`), assuming that there is no occluding surface between them. We separate generation of the emitter point from estimating the radiance from it so that shadowing algorithms can be applied. We must know the actual scene point and not just the direction to it in order to compute the radial falloff from noncollimated sources.

Listing 14.11 shows how to apply these methods to compute the radiance scattered toward the viewer due to direct illumination. In the listing, point `P` is the point to be shaded, `w_o` is the unit vector in the direction from `P` to the eye, `n` is the unit surface normal at `P`, and `bsdf` is a model of how the surface scatters light (see Chapter 27 for a full discussion of physically based scattering).

Listing 14.11: Direct illumination from an arbitrary set of lights.

```
/** Computes the outgoing radiance at P in direction w_o */
Radiance3 shadeDirect
(const Vector3& w_o,  const Point3&  P,
 const Vector3& n,    const BSDF&    bsdf,
 const std::vector<Light*>& lightArray) {

  Radiance3 L_o(0.0f);

  for (int i = 0; i < lightArray.size(); ++i) {
    const Light* light = lightArray[i];

    int N = numSamplesPerLight;

    // Don't over-sample point lights
    if (light->surfaceArea() == 0) N = 1;

    for (int s = 0; s < N; ++s) {
      const Vector4& Q = light->randomPoint()
      const Vector3& w_i = (Q.xyz() * P - P * Q.w).direction();

      if (visible(P, Q)) { // shadow test
        const Biradiance3& M_i = light->biradiance(Q, P);
        const Color3& f = bsdf.evaluateFiniteScatteringDensity(w_i, w_o, n);

        L_o += n.dot(w_i) * f * M_i / N;
      }
    }
  }

  return L_o;
}
```

If this is your first encounter with code like that shown in Listing 14.11, simply examine it for now and then treat it as a black box. A fuller explanation of the radiometry that leads to this implementation and motivates the abstractions appears in Chapter 32. A brief explanation of the derivation of this implementation appears in the following section.

14.11.3.3 ◈ Relationship to the Rendering Equation

We now attempt to reconcile the "light" that's used in the traditional graphics pipeline (e.g., in Chapter 6) with the physically based rendering model that is described in Chapter 31. The key idea is that if we measure light in units of *biradiance*,[6] then classic graphics models can function as simplified versions of the physics of light and the rendering equation.

This material appears in this chapter because it constitutes a conventional approximation to the real physics of light, which we wanted to introduce before the full theory. It of course also serves readers encountering this section *after* reading Chapters 26 and 31, or with previous experience in graphics systems.

6. We are not aware of a preexisting name for solid-area weighted radiance measured at the *receiver*, so we introduce the term "biradiance" here to express the idea that it incorporates two points. This quantity is distinct from radiosity (which considers the whole hemisphere), irradiance (which is measured at the emitter's surface), radiant emittance (likewise), and other quantities that commonly arise with the same units.

For a scene with only point lights and Phong BSDFs in which we have, `numSamplesPerLight = 1`, Listing 14.11 degenerates into the familiar OpenGL fixed-function shading algorithm. The framework presented in this chapter provides some explanation of what the lighting parameters in OpenGL "mean." That puts us on a somewhat more solid footing when we attempt to render scenes designed for classic computer graphics point sources within a physically based renderer. It also leads us to the settings perhaps most likely to produce a realistic image under rendering APIs similar to OpenGL.

For all scenes, Listing 14.11 implements direct illumination in the style employed both for algorithms like path tracing and for explicit direct illumination under rasterization. It is an estimator for the terms in the rendering equation due to direct illumination (Figure 14.28 shows the key variables, for reference). The perhaps unexpected radiant emittance units arise from a change of variables from the form in which we often express the rendering equation. That is, we commonly express the scattered direct illumination as

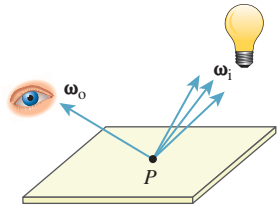


Figure 14.28: Light reflected at P.

$$L(P, \omega_o) = \int_{\Omega^+} L(P, -\omega_i) f_P(\omega_i, \omega_o) \mathbf{n} \cdot \omega_i d\omega_i, \tag{14.25}$$

where the domain of integration is the hemisphere Ω^+ above the point P at which we are computing the illumination.

That would lead us to an implementation like:

```
1  repeat N times:
2     dw_i = 2 * PI / N;
3     L_i = ...;
4     L_o += L_i * bsdf.evaluate(...) * n.dot(w_i) * dw_i;
```

However, path tracing and other algorithms that employ explicit direct illumination sampling tend to sample over the area of light sources, rather than over the directions about the shaded point.

We must change integration domains from Ω^+ to the surfaces of the lights; that entails making the appropriate change of variables. Consider a single light with surface region ΔA and unit surface normal $\mathbf{m}$ at Q (see Figure 14.29).

The distance from P to the surface region ΔA is approximately $r = \|Q - P\|$, while the distance from P to $\Delta\Omega$ is exactly 1. (Recall that $\Delta\Omega$ is a small region on the unit sphere around P.) If the region ΔA near Q were not tilted (i.e., if $\mathbf{m}$ and ω_i were opposites), then its area would be r^2 times the area of $\Delta\Omega$. The tilting principle of Section 7.10.6 says that the area is multiplied by a cosine factor. So the area of ΔA is approximately $r^2|\mathbf{m} \cdot \omega_i|$ times the area of $\Delta\Omega$, with the approximation getting better and better as the region $\Delta\Omega$ shrinks in size. Thus the change of variable, as we go from $d\omega_i$ to dA (often denoted by some symbol like $\frac{d\omega_i}{dA}$) is $\frac{|\mathbf{m} \cdot \omega_i|}{\|Q-P\|^2}$. We can now rewrite Equation 14.25, substituting $\mathcal{S}(Q - P)$ for ω_i,

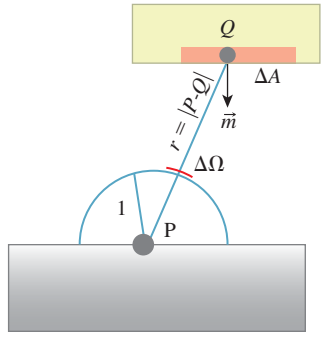


Figure 14.29: The small, solid-angle $\Delta\Omega$ and a corresponding small region ΔA on the luminaire's surface.

$$\begin{aligned} L(P, \omega_o) &= \int_{\Omega^+} L(P, -\omega_i) f_P(\omega_i, \omega_o) \mathbf{n} \cdot \omega_i \, d\omega_i \qquad (14.26) \\ &= \int_{Q \in R} L(P, \mathcal{S}(P - Q)) \, f_P(\mathcal{S}(Q - P), \omega_o) \, \mathbf{n} \cdot \mathcal{S}(Q - P) \cdot \\ &\quad \frac{\mathbf{m} \cdot \mathcal{S}(P - Q)}{\|Q - P\|^2} \, dA, \qquad (14.27) \end{aligned}$$

where we've used $P - Q$ instead of $Q - P$ in some places to avoid excess negative signs or absolute values.

One way to estimate the integral of *any* function g over *any* region is to pick a random point X in the region, evaluate $g(X)$, and multiply by the area of the region. (We discuss this in far more detail in Chapter 30.) If we repeat this process, then each individual estimate will probably not be very good, but their average will get better and better the more (independent) points we choose. Applying this to the integral above, if we choose N points $Q_j \in R$ on the luminaire, we can estimate the reflected light as

$$\begin{aligned} L(P, \omega_o) &= \int_{Q \in R} L(P, \mathcal{S}(P - Q)) f_P(\mathcal{S}(Q - P), \omega_o)\, \mathbf{n} \cdot \mathcal{S}(Q - P) \cdot \\ &\quad \frac{\mathbf{m} \cdot \mathcal{S}(P - Q)}{\|Q - P\|^2}\, dQ \qquad (14.28) \\ &\approx \sum_{j=1}^{N} L(P, \mathcal{S}(P - Q_j)) f_P(\mathcal{S}(Q_j - P), \omega_o)\, \mathbf{n} \cdot \mathcal{S}(Q_j - P) \cdot \\ &\quad \frac{\mathbf{m} \cdot \mathcal{S}(P - Q_j)}{\|Q_j - P\|^2} \frac{A}{N}. \qquad (14.29) \end{aligned}$$

Another interpretation of the factor $\frac{A}{N}$ is that each sample Q_i represents one Nth of the area of the luminaire.

We can simplify the notation in this expression somewhat by letting $\omega_{i,j}$ be the unit vector $\mathcal{S}(Q_j - P)$ from P to the jth sample point Q_j on the luminaire. We then have

$$L(P, \omega_o) \approx \sum_{j=1}^{N} L(P, \omega_{i,j}) f_P(\omega_{i,j}, \omega_o)\, \mathbf{n} \cdot \omega_{i,j} \frac{-\mathbf{m} \cdot \omega_{i,j}}{\|Q_j - P\|^2} \frac{A}{N}. \qquad (14.30)$$

If we denote by $M(P, Q_j, \mathbf{m})$ the value

$$M(P, Q_j, \mathbf{m}) = AL(P, \omega_{i,j}) \frac{-\mathbf{m} \cdot \omega_{i,j}}{\|Q_j - P\|^2}, \qquad (14.31)$$

then the reflected radiance becomes

$$\begin{aligned} L(P, \omega_o) &\approx \frac{1}{N} \sum_{j=1}^{N} M(P, Q_j) f_P(\omega_{i,j}, \omega_o)\, \mathbf{n} \cdot \omega_{i,j} \qquad (14.32) \\ &= \frac{1}{N} \sum_{j=1}^{N} M(P, Q_j, \mathbf{m}) f_P(\mathcal{S}(Q_j - P), \omega_o)\, \mathbf{n} \cdot \mathcal{S}(Q_j - P). \qquad (14.33) \end{aligned}$$

The units of M are m^2 times the units of radiance times the units of $d\omega/dA$, which are steradians per meter squared; the net units are W/m^2. We'll call M the **biradiance,** to indicate its dependence on *two* points, one on the luminaire and one on the receiving surface. We caution that this is not a standard radiometric term.

We'll return to the interpretation of M in a moment, but the structure of this equation leads us to pseudocode that follows the structure of Listing 14.11:

```
repeat N times:
   // Computed by the emitter
   L_i = ...
   M_i = L_i * max(-m.dot(w_i), 0.0f) * A / ||P - Q||^2

   // Computed by the integrator (i.e., renderer)
   if there is an occluder on the line from P to Q then M_i = 0
   L_o += M_i * bsdf.evaluate(...) * n.dot(w_i) / N
```

Thus, one interpretation of the deprecated fixed-function OpenGL Phong shading and (to some degree) shading in WPF is that the light "intensities" correspond to the function M, whose units are watts per square meter. This particular quantity is not one that has a standard name in radiometry, however, and we will not mention it again. It's also worth noting that in most simple rendering using the classical model, the $1/r^2$ falloff is not actually modeled, so the claim that what's used as an intensity is M is somewhat suspect. What's true is that if you wish to use the classical model to approximate physical reality, then you *should* use the $1/r^2$ falloff, and you should assign luminaires "intensity" values computed according to the formula for M.

14.11.3.4 Applying the Interface to Photon Emission

Algorithms such as bidirectional ray tracing and photon mapping track the path of virtual photons forward from the light source into the scene. These virtual photons differ from the real photons that they model in two respects. The first distinction is that their state includes a position, a direction of propagation, and the *power* of the photon. Real photons transport *energy,* but rendering considers steady-state light transport, so it tracks the rate of energy delivery. It is more accurate to say that each virtual photon models a stream of photons, or that it models a segment of a light-transport path. The second distinction from real photons is that trillions of real photons contribute to a single real image, whereas renderers typically sample only a few million virtual photons (although each represents a stream of photons, so in truth many more real photons are implicitly modeled).

The first step of photon tracing is to emit photons from the sources in the scene. Listing 14.12 shows how to use our light interface to sample `numPhotons`-emitted virtual photons with a probability density function proportional to the power of each light source.

Listing 14.12: Generating `numPhotons` photons from a set of lights; for example, for photon mapping.

```
void emitPhotons
(const int                  numPhotons,
 const Array<Light*>&       lightArray,
 Array<Photon>&             photonArray) {

  const Power3& totalPower;
  for (int i = 0; i < lightArray.size(); ++i)
    totalPower += lightArray[i]->power();

  for (int p = 0; p < numPhotons; ++p) {
    // Select Lwith probability L.power.sum()/totalPower.sum()
    const Light* light = chooseLight(lightArray, totalPower);

    Point3       Q;
    Vector3      w_o;
```

```
    const Color3& c = light->emitPhoton(Q, w_o);

    photonArray.append(Photon(c*totalPower/numPhotons, Q, w_o));
  }
}
```

14.11.4 Rectangular Area Light

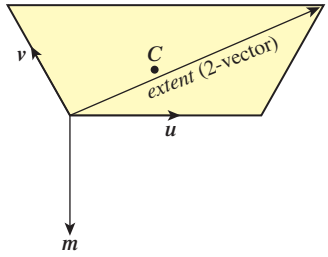


Figure 14.30: Parameterization of a square area light on the ceiling.

We now model a planar rectangular patch that emits light from a single side, as shown in Figure 14.30. This will be a so-called "Lambertian emitter," which means that if we restrict our field of view so that the emitter fills it entirely, then we perceive the emitter's brightness as the same regardless of its orientation or distance. We describe the light source's orientation by an orthonormal reference frame in which `m` is the unit normal to the side that emits photons and `u` and `v` are the axes along the edges. The edges have lengths given by `extent` and the source is centered on point `C`.

The total power emitted by the light is `Phi`. This means that if we increase `extent`, the illumination level in the scene will appear constant but the emitter's surface will appear to become darker since the same power is distributed over a larger area.

The radiance $(\mathrm{W}/(\mathrm{m}^2\ \mathrm{sr}))$ due to a point Q on a Lambertian emitter, emitted in any direction ω, is the total emitted power (W) divided by the area (m^2) of the emitter and the so-called "projected solid angle" (sr)

$$L(Q,\omega) = \frac{\Phi}{A \int_{\Omega+} [\gamma \cdot \mathbf{m}]\, d\gamma} \tag{14.34}$$

$$= \frac{\Phi}{A\ \pi\ \mathrm{sr}}. \tag{14.35}$$

For a discussion of this and other radiometric terms, see Section 26.7.1.

We can now compute the biradiance at P due to Q. Let $\omega_\mathrm{i} = S(Q-P)$. Light leaving Q in direction $-\omega_\mathrm{i}$ arrives at P traveling in direction $-\omega_\mathrm{i}$. The leaving light has radiance $\Phi/(A\pi\ \mathrm{sr})$ by Equation 14.35. So

$$M_\mathrm{i}(Q,P) = L(P,-\omega_\mathrm{i}) \frac{A}{||Q-P||^2} (-\mathbf{m}\cdot\omega_\mathrm{i})\mathrm{sr} \tag{14.36}$$

$$= \frac{\Phi}{A\ \pi\ \mathrm{sr}} \frac{A}{||Q-P||^2} (-\mathbf{m}\cdot\omega_\mathrm{i})\mathrm{sr} \tag{14.37}$$

$$= \frac{(-\mathbf{m}\cdot\omega_\mathrm{i})\ \Phi}{||Q-P||^2\ \pi}, \tag{14.38}$$

assuming that P is on the emitting side of the light source and there is an unoccluded line of sight to Q. Listing 14.13 gives an implementation of the key `Light` methods for such a light source based on this derivation.

Listing 14.13: A model of a single-sided rectangular Lambertian emitter.

```
class RectangularAreaLight : public Light {
private:
  // Orthonormal reference frame of the light
  Vector3 u, v, m;
```

```
5
6     Vector2 extent;
7
8     // Center of the source
9     Point3 C;
10
11    Power3 Phi;
12
13  public:
14
15    ...
16
17    Vector4 randomPoint() const {
18      return Vector4(C +
19          u * (random(-0.5f, 0.5f) * extent.x) +
20          v * (random(-0.5f, 0.5f) * extent.y), 1.0f);
21    }
22
23    float area() const {
24      return extent.x * extent.y;
25    }
26
27    Power3 power() const {
28      return Phi;
29    }
30
31    Biradiance3 biradiance(Vector4 Q, Vector3 P) const {
32      assert(Q.w == 1);
33      const Vector3& w_i = (Q.xyz() - P).direction();
34
35      return Phi * max(-m.dot(w_i), 0.0f) /
36             (PI * (P - Q.xyz()).squaredLength());
37    }
38  };
```

14.11.5 Hemisphere Area Light

A large hemispherical light source that emits inward is a common model of the sky or other distant parts of the environment. Listing 14.14 adapts the concepts from the rectangular area light source to such a dome. Two natural extensions to this model are to incorporate a coordinate frame so that the hemisphere can be arbitrarily centered and oriented, and to modulate the power over the dome by an image to better simulate complex environments and skies with high variability.

Listing 14.14: A model of an inward-facing hemispherical light dome, centered at the origin and with rotational symmetry about the y-axis.

```
1  class HemisphereAreaLight : public Light {
2  private:
3    // Radius
4    float r;
5    Power3 Phi;
6
7  public:
8
9    ...
10
11   Vector4 randomPoint() const {
12     return Vector4(hemiRandom(Vector3(0.0f, 1.0f, 0.0f)) * r, 1.0f);
13   }
```

```
14
15    float area() const {
16      return 2 * PI * r * r;
17    }
18
19    Power3 power() const {
20      return Phi;
21    }
22
23    Biradiance3 biradiance(Vector4 Q, Vector3 P) const {
24      assert(Q.w == 1 && Q.xyz().length() == r);
25
26      const Vector3& m = -Q.xyz().direction();
27      const Vector3& w_i = (Q.xyz() - P).direction();
28
29      return Phi * max(-m.dot(w_i), 0.0f) /
30             (PI * (P - Q.xyz()).squaredLength());
31    }
32  };
```

14.11.6 Omni-Light

An **omnidirectional point light** (**omni-light** or [ambiguously] **point light**) is a luminaire that emits energy equally in all directions and is sufficiently small that it has negligible bounding radius compared to the distance between the source and nearby scene locations. A true point light would have to have a surface that was infinitely bright to produce measurable emission from zero surface area, and so could not exist. However, there are many luminaires whose volume is negligible compared to the scale of the scenes in which they are encountered, such as the bulb in a flashlight or the LED lights on the dashboard of a car. It is also common to approximate a larger light source with an omni-light at its center, and some surrounding proxy geometry that appears to the viewer to be the luminaire but does not actually emit light in the lighting simulation. For example, a campfire might be modeled by a flickering omni-light floating in the midst of the flames, which were themselves rendered by a particle system.

An omni-light is typically modeled by its total power emission in all directions, Φ. This is a scalar measured in watts; it can be represented as a 3-tuple to express power at the red-green-blue frequencies. Real-life experience provides good estimates for the power of omni-lights in our scene, since lightbulbs are labeled in watts of power consumed. As we said earlier, the emitted light from a 100 W bulb is about 4 W. A fluorescent bulb is about six times more efficient, so a bulb labeled "equivalent to a 100 W incandescent bulb" also emits about 4 W of visible light, but it consumes less electric power in doing so.

Let Q be the center of an omni-light. The radiance at P directly from the luminaire must arrive *from* direction $\omega_i = \mathcal{S}(Q - P)$. That is, ω_i points to the light, from the surface. It is known as the **light vector** and is sometimes also denoted $\hat{L}$ (although we avoid that notation because it is confusingly similar to the radiance function notation $L(\cdot)$).

The omni-light is an abstraction of a very small spherical source. We can estimate the radiant emittance due to an omni-light by estimating the effect of ever-smaller spherical sources. The only way that the size of a source enters our formula is in the surface area term A in Equation 14.37. However, that term appears both in numerator and denominator, so it cancels and the end result is independent of

the area. The cosine term also varies from point to point for an area source but is constant for the omni-light's point source; one might say that this is another way that size "matters." Regardless, the conclusion is that we can use exactly the same formula for the biradiance from area and point sources.

The biradiance at P due to the omni-light at Q (see Figure 14.31) is given by

$$M_{\mathrm{i}}(Q, P) = \frac{\Phi}{4\pi \; ||Q - P||^2 \; \mathrm{sr}}, \tag{14.39}$$

if there is no scene point on the open line segment from Q to P; it is 0 if there is an occlusion.

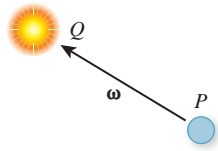


Figure 14.31: Points and directions in point-light equations.

Note that we could say that the effective radiance is

$$L(P, -\omega_{\mathrm{i}}) = \frac{\Phi}{4\pi \; \mathrm{sr} \; ||Q - P||^2}, \tag{14.40}$$

that is, it is proportional to the total power of the luminaire and falls off with the square of distance from the luminaire. In fact, if we were to insert that expression into a renderer it would yield the desired image so long as $||Q - P||$ was "sufficiently large." However, this is not actually a true radiance expression because it is not conserved along a ray—it falls off with distance because our omni-light in fact has the physically impossible zero surface area, which leads it to create a physically impossible radiance field in space. Note that both the radiance and the biradiance approach infinity as $||Q - P|| \to 0$. It is common practice to clamp the maximum biradiance from an omni-light, since when that distance is small our original assumption that the distance to the luminaire is much greater than the size of the luminaire is violated and the resultant estimated light intensity is greater than intended. A less efficient but more accurate correction would be to actually model the luminaire as an object with nonzero surface area (such as a sphere) when the distance is less than some threshold.

There is no illumination from the omni-light at locations that do not have an unobstructed line of sight. These regions form the shadows in an image. The boundary of the shadows from an omni-light will be "hard," with a distinct curve across a surface that distinguishes lit from shadowed. This is unlike the area sources, which produce "soft" shadows with blurry silhouettes. A lighting algorithm such as shadow mapping that evaluates light visibility at lower precision than the radiance magnitude can appear to produce soft shadows from a point light. This is in fact an artifact of reconstruction from an aliased set of samples. Nonetheless, it may be visually pleasing in practice.

14.11.7 Directional Light

For an omnidirectional point light that is far from all locations in the scene, ω_{i} and $L(P, -\omega_{\mathrm{i}})$ due to the light vary little across the scene. A **directional light** is an omni-light with the further simplifying approximation that it is so far from the rest of the scene that ω_{i} and L can be treated as constant throughout the scene. This eliminates some of the precision and modeling challenges of placing a point light very far away, while giving a reasonable model for a distant light source such as the sun.

We could model the total power of the distant point light, but it is typically enormous and "distant" is ambiguous, so it is easier to model the (constant) incident radiance at points in the scene by simply letting $L(P, \omega) = L_0$ for ω_{i} directed

exactly toward the light and 0 for all other directions. A useful constant to remember is that, for points on the surface of the Earth, L_0 (in the visible spectrum) due to the sun is roughly $1.5 \times 10^6\,\mathrm{W/m^2sr}$. The total power of light in the visible spectrum arriving from all points of the sun at a region of the Earth's surface is about $150\,\mathrm{W/m^2}$. Both of these, of course, vary with time of day, season, and latitude.

14.11.8 Spot Light

A spot light models an omni-light shielded by "barn door" flaps or a conical shade. Theatre lights, flashlights, car headlights, and actual spot lights are examples of real-world sources for which this model is appropriate (see Figure 14.32). It is common to choose to model the occluding portion as a perfectly absorptive sphere with a round iris through which light emerges. It is assumed to be perfectly absorptive so that we can neglect the complex reflections that occur on the silvered reflector of a headlight and flashlight. The iris is made round because that allows us to test whether a ray passes through it using a simple threshold of a dot product. A round iris produces a cone of light.

Figure 14.32: A theatre light with square "barn doors" (left), and a spot light with a round iris, matching our model. (Credit: top: Jim Barber/Shutterstock, bottom: Matusciac Alexandru/Shutterstock)

Let Φ be the total power of the omni-light inside the blocker. It is convenient to specify this rather than the total power actually emitted. Doing so allows a lighting artist to adjust the spot-light cone independently of the observed brightness of objects within the cone.

Let $0 \leq \theta \leq \pi$ be the measure of the angle between the axis and side of the emitted light cone. Note that $\theta = \pi$ is an iris fully opened so that there is no blocker and $\theta = 0$ is completely closed. Beware of conventions here. Some APIs use radians, some use degrees. Also, some specify the full cone angle and some the half-angle as we have done here.

The measure of the solid angle subtended by a cone with angle measure $0 \leq \theta \leq \pi$ between the axis and side is

$$2\pi(1 - \cos\theta). \tag{14.41}$$

The fraction of the emitter that is visible through the barn doors is therefore

$$\frac{2\pi(1 - \cos\theta)}{4\pi}, \tag{14.42}$$

so the externally observed power Φ' of an omni-light of power Φ occluded by barn doors is

$$\Phi' = \frac{\Phi}{2}(1 - \cos\theta), \tag{14.43}$$

which is the `Light::power` value required for implementation in an importance-sampling renderer.

Spot lights with $\theta \leq \pi/4$ are used extensively in rendering because they provide reasonable resolution for a planar projection of light space. That is, by placing a camera at the light, facing along its axis and with field of view matching the cone angle, one can render the light's view of the scene with reasonably small distortion at the edges. This technique is used in shadow maps, for example, which are depth images from the light's perspective, and reflective shadow maps, which are color images from the light's perspective. Six spot lights, cropped down to square projections, can cover the six faces of a cube and represent shadowing from an omni-directional light.

Another application of light-space projection is a **projective spot light.** Real theatre lights are often modified by placing a **gobo** or **cookie** slide immediately outside the light source's iris (see Figure 14.33). This colors or selectively blocks emitted light, projecting an image or shape onto the scene. In computer graphics, one can modulate the incident light at a point P in the scene by the value stored in an image at the corresponding location in the light's projection of P to achieve this effect. This is used to create nonround and nonuniform spot-light apertures, to simulate the complex patterns of real spot-light reflectors, and to simulate shadows from off-screen objects, such as a spinning fan in a ventilation duct.

Figure 14.33: A photograph of spiral patterns created by real gobos in spotlights (Credit: R. Gino Santa Maria/Shutterstock.com).

14.11.9 A Unified Point-Light Model

This section describes a model for luminaires with extents that are small in comparison to the distance between them and the points to be shaded. Under this definition of small, each luminaire can be approximated as a single point. This unifies several common light models. It was introduced and favored particularly for fixed-function graphics processors but remains widespread due to its simplicity.

A **fixed-function** unit implements a specific algorithm directly in circuitry or microcode. Such units are often controlled by parameters but cannot perform general-purpose computations the way that a programmable unit or general-purpose processor can. That is, they are not computationally equivalent to Turing machines. A processor typically contains a mixture of programmable and fixed-function units. For example, few architectures allow the programmer to alter the cache replacement strategy, but most allow arbitrary arithmetic expressions within programs. Graphics architectures may embed entire rendering algorithms in fixed-function logic. Fixed-function hardware naturally limits the programmer's expression. However, it is extremely power-efficient and is less expensive to design and produce than general-purpose computation units. Thus, hardware architects face a design tradeoff based on current costs and goals.

At the time of this writing, fixed-function graphics units have gone in and out of fashion several times. Fixed-function lighting logic is currently eschewed in most devices, although at least one (the Nintendo 3DS [KO11]) released in 2011 embraces it.

We do not recommend the unified model presented in this section for new implementations built on programmable shading or software-based rendering APIs. The model is difficult to incorporate in a useful way into a physically based rendering system, limits the flexibility of the lighting model, and is a weaker abstraction than the models presented in subsequent chapters.

It remains important to be familiar with the model inspired by fixed-function logic for several reasons. Both legacy devices and a handful of new devices still use this model. The model may return to favor in the future. Many programmable graphics pipelines are still based around the fixed-function lighting model because they evolved from fixed-function implementations or must work with assets and tools originally designed for those implementations.

The basic idea of the model is that we can represent spot, directional, and omni-lights with a single, branchless lighting equation for a spot light with homogeneous parameters. The center of the spot light is (x, y, z, w), where $w = 1$ indicates a spot or omni-light and $w = 0$ indicates a directional light. If we parameterize the angle between the axis of the spot light and the edge of its cone, then an angle of π radians gives an omnidirectional light. The only remaining issue is

radial falloff. In the real world, the total power observed at distance r from a uniform spherical emitter whose radius is much smaller than r is proportional to $1/r^2$. We can generalize this by computing a value M that involves an inverse quadratic:

$$M = \frac{\Phi}{(a_0 r^0\ \mathrm{m}^2 + a_1 r^1\ \mathrm{m}^1 + a_2 r^2\ \mathrm{m}^0)\ 4\pi}. \tag{14.44}$$

If we define **a** and **r** by

$$\vec{a} = (a_0\ \mathrm{m}^2, a_1\ \mathrm{m}^1, a_2\ \mathrm{m}^0), \text{ and} \tag{14.45}$$

$$\vec{r} = (r^0, r^1, r^2), \tag{14.46}$$

then we can rewrite the formula for M as

$$M = \frac{\Phi}{\vec{a} \cdot \vec{r}\ 4\pi}. \tag{14.47}$$

This strange expression lets us represent a point emitter that experiences nonphysical falloff to approximate a local area source or distant point source . . . or simply to satisfy an artistic vision. In this context, a directional source can be parameterized by the attenuation constant $\vec{a} = (1, 0, 0)$. This source will produce equal intensity at all points in the scene, and that intensity is comparable to what a local source with power Φ one meter from a surface would produce.

The resulting interface (Listing 14.15) follows the spirit of the OpenGL fixed-function lighting model, albeit with slightly varying units and border cases. We do not recommend this model for physically based rendering.

Listing 14.15: A simple unified model for spot, directional, and omni light sources.

```
class PointLight : public Light {
private:

  /** For local lights, this is the total power of the light source.
      For directional lights, this is the power of an equivalent
      local source 1^m from the surface.*/
  Power3       Phi;

  Vector3      axis;

  /** Center of the light in homogeneous coordinates. */
  Vector4      C;

  Vector3      aVec;

  float        spotHalfAngle;

  ...
};
```

Listing 14.16: `PointLight` *methods for direct illumination.*

```
Vector4 PointLight::randomPoint() const {
    return C;
}

```

```
Biradiance3 biradiance
    (const Vector4& Q, const Point3& P) const {
     assert(C == Q);

       // Distance to the light, or zero
       const float r = ((Q.xyz() - P) * Q.w).length();

       // Powers of r
       const Vector3 rVec(1.0f, r, r * r);

       // Direction to the light
       const Vector3& w_i = (Q.xyz() - P * Q.w).direction();

       const bool inSpot = (w_i.dot(axis) >= cos(spotHalfAngle));

       // Apply radial and angular attenuation and mask by the spotlight cone.
       return Phi * float(inSpot) / (rVec.dot(aVec) * 4 * PI);
}
```

Listing 14.17: `PointLight` *methods for photon emission.*

```
Power3 PointLight::totalPower() const {
  // the power actually emitted depends on the solid angle of the cone; it goes to
  // infinity for a directional source
  return Phi * (1 - cos(spotHalfAngle)) / (2 * C.w);
}

Color3 PointLight::emitPhoton(Point3& P, Vector3& w_o) {
  // It doesn't make sense to emit photons from a directional light with unbounded
  // extent because it would have infinite power and emit practically all photons
  // outside the scene.
  assert(C.w == 1.0);

  // Rejection sample the spotlight cone
  do {
     w_o = randomDirection();
  } while (spotAxis.dot(w_o) < cos(spotHalfAngle));

  P = Q.xyz();

  // only the ratios of r:g:b matter
  const Color3& spectrum = Phi / Phi.sum();
  return spectrum;
}
```

14.12 Discussion

Each of the approximations and representations presented in this chapter has its place in graphics. Different constraints—on processor speed, bandwidth, data availability, etc.—create contexts in which they made (or make) sense. And while processors get faster, new constraints, like the limited power of mobile devices, may revive some approximations for a time. You should therefore regard these not only as currently or formerly useful tricks of the trade, but as things that are potentially useful in the future as well, and which provide examples of how to approximate things effectively within a limited resource budget.

14.13 Exercises

Exercise 14.1: Give an example of an arithmetic expression in which the commutativity does not hold for all operations; for example, in which evaluating left-to-right instead of following the usual order of operations gives an incorrect result.

Exercise 14.2: We said that direct mapping of the range $[0, 2^b - 1]$ to $[-1, 1]$ precludes the exact representation of zero. Explain why. (You may find it easiest to start with the case $b = 1$.)

Exercise 14.3: Write a function that converts a triangle strip to a triangle list (a.k.a. triangle soup).

Exercise 14.4: Write a function that converts a triangle fan to a triangle strip. You may need to introduce degenerate triangles along edges.

Exercise 14.5: Consider a program that represents the world using a 3D array of opaque voxels; for simplicity, assume that they are either present or not present. Most rendering APIs use meshes, not voxels, so this program will have to convert the voxels to faces for rendering. Each filled voxel has six faces. But because the voxels are opaque, most of the faces in the scene do not need to be rendered—they are between adjacent filled voxels and can never be seen.

Give an algorithm for iterating through the scene and outputting only the faces that can be observed.

Exercise 14.6: Draw a computer science tree data structure representing the scene graph for an automobile, with the nodes labeled as to the parts that they represent.

Exercise 14.7: Consider a green beer bottle on a white table, in a night club lit with only red lights (assume each represents a narrow frequency range). What are the observed colors of the bottle, the table in the bottle's shadow, and the table out of the bottle's shadow?

Exercise 14.8: Implement disk and sphere Lambertian emitters in the style of Listing 14.13.

Exercise 14.9: Implement an arbitrary mesh Lambertian emitter in the style of Listing 14.13.

This page intentionally left blank

Chapter 15

Ray Casting and Rasterization

15.1 Introduction

Previous chapters considered modeling and interacting with 2D and 3D scenes using an underlying renderer provided by WPF. Now we focus on writing our own physically based 3D renderer.

Rendering is integration. To compute an image, we need to compute how much light arrives at each pixel of the image sensor inside a virtual camera. Photons transport the light energy, so we need to simulate the physics of the photons in a scene. However, we can't possibly simulate *all* of the photons, so we need to sample a few of them and generalize from those to estimate the integrated arriving light. Thus, one might also say that rendering is sampling. We'll tie this integration notion of sampling to the alternative probability notion of sampling presently.

In this chapter, we look at two strategies for sampling the amount of light transported along a ray that arrives at the image plane. These strategies are called **ray casting** and **rasterization.** We'll build software renderers using each of them. We'll also build a third renderer using a hardware rasterization API. All three renderers can be used to sample the light transported to a point from a specific direction. A point and direction define a ray, so in graphics jargon, such sampling is typically referred to as "sampling along a ray," or simply "sampling a ray."

There are many interesting rays along which to sample transport, and the methods in this chapter generalize to all of them. However, here we focus specifically on sampling rays within a cone whose apex is at a point light source or a pinhole camera aperture. The techniques within these strategies can also be modified and combined in interesting ways. Thus, the essential idea of this chapter is that rather than facing a choice between distinct strategies, you stand to gain a set of tools that you can modify and apply to any rendering problem. We emphasize two aspects in the presentation: the principle of sampling as a mathematical tool and the practical details that arise in implementing real renderers.

Of course, we'll take many chapters to resolve the theoretical and practical issues raised here. Since graphics is an active field, some issues will not be thoroughly resolved even by the end of the book. In the spirit of servicing both principles and practice, we present some ideas first with pseudocode and mathematics and then second in actual compilable code. Although minimal, that code follows reasonable software engineering practices, such as data abstraction, to stay true to the feel of a real renderer. If you create your own programs from these pieces (which you should) and add the minor elements that are left as exercises, then you will have three working renderers at the end of the chapter. Those will serve as a scalable code base for your implementation of other algorithms presented in this book.

The three renderers we build will be simple enough to let you quickly understand and implement them in one or two programming sessions each. By the end of the chapter, we'll clean them up and generalize the designs. This generality will allow us to incorporate changes for representing complex scenes and the data structures necessary for scaling performance to render those scenes.

We assume that throughout the subsequent rendering chapters you are implementing each technique as an extension to one of the renderers that began in this chapter. As you do, we recommend that you adopt two good software engineering practices.

1. Make a copy of the renderer before changing it (this copy becomes the **reference renderer**).
2. Compare the image result after a change to the preceding, reference result.

Techniques that enhance performance should generally not reduce image quality. Techniques that enhance simulation accuracy should produce noticeable and measurable improvements. By comparing the "before" and "after" rendering performance and image quality, you can verify that your changes were implemented correctly.

Comparison begins right in this chapter. We'll consider three rendering strategies here, but all should generate identical results. We'll also generalize each strategy's implementation once we've sketched it out. When debugging your own implementations of these, consider how incorrectly mismatched results between programs indicate potential underlying program errors. This is yet another instance of the Visual Debugging principle.

15.2 High-Level Design Overview

We start with a high-level design in this section. We'll then pause to address the practical issues of our programming infrastructure before reducing that high-level design to the specific sampling strategies.

15.2.1 Scattering

Light that enters the camera and is measured arrives from points on surfaces in the scene that either scattered or emitted the light. Those points lie along the rays that we choose to sample for our measurement. Specifically, the points casting light into the camera are the intersections in the scene of rays, whose origins are points on the image plane, that passed through the camera's aperture.

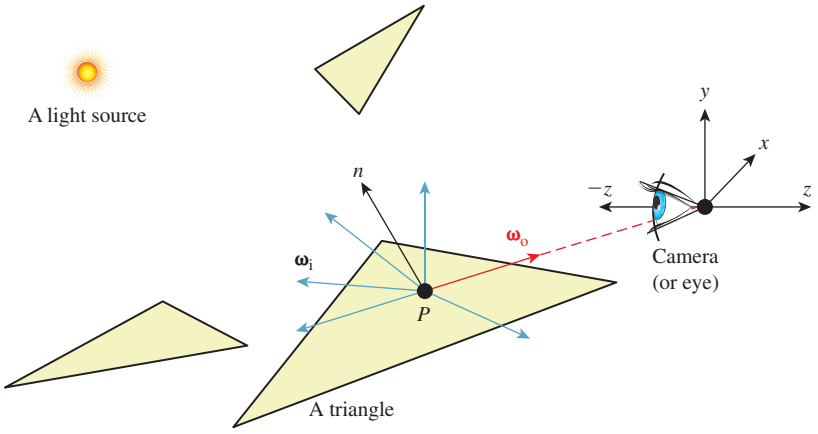


Figure 15.1: A specific surface location P that is visible to the camera, incident light at P from various directions $\{\omega_i\}$*, and the exitant direction* ω_o *toward the camera.*

To keep things simple, we assume a pinhole camera with a virtual image plane *in front* of the center of projection, and an instantaneous exposure. This means that there will be no blur in the image due to defocus or motion. Of course, an image with a truly zero-area aperture and zero-time exposure would capture zero photons, so we take the common graphics approximation of estimating the result of a *small* aperture and exposure from the limiting case, which is conveniently possible in simulation, albeit not in reality.

We also assume that the virtual sensor pixels form a regular square grid and estimate the value that an individual pixel would measure using a single sample at the center of that pixel's square footprint. Under these assumptions, our sampling rays are the ones with origins at the center of projection (i.e., the pinhole) and directions through each of the sensor-pixel centers.[1]

Finally, to keep things simple we chose a coordinate frame in which the center of projection is at the origin and the camera is looking along the negative z-axis. We'll also refer to the center of projection as the eye. See Section 15.3.3 for a formal description and Figure 15.1 for a diagram of this configuration.

The light that arrived at a specific sensor pixel from a scene point P came from some direction. For example, the direction from the brightest light source in the scene provided a lot of light. But not all light arrived from the brightest source. There may have been other light sources in the scene that were dimmer. There was also probably a lot of light that previously scattered at other points and arrived at P indirectly. This tells us two things. First, we ultimately have to consider all possible directions from which light may have arrived at P to generate a correct image. Second, if we are willing to accept some sampling error, then we can select a finite number of discrete directions to sample. Furthermore, we can

1. For the advanced reader, we acknowledge Alvy Ray Smith's "a pixel is not a little square"—that is, no sample is useful without its reconstruction filter—but contend that Smith was so successful at clarifying this issue that today "sample" now is properly used to describe the point-sample data to which Smith referred, and "pixel" now is used to refer to a "little square area" of a display or sensor, whose value may be estimated from samples. We'll generally use "sensor pixel" or "display pixel" to mean the physical entity and "pixel" for the rectangular area on the image plane.

probably rank the importance of those directions, at least for lights, and choose a subset that is likely to minimize sampling error.

> **Inline Exercise 15.1:** We don't expect you to have perfect answers to these, but we want you to think about them now to help develop intuition for this problem: What kind of errors could arise from sampling a finite number of directions? What makes them errors? What might be good sampling strategies? How do the notions of expected value and variance from statistics apply here? What about statistical independence and bias?

Let's start by considering all possible directions for incoming light in pseudocode and then return to the ranking of discrete directions when we later need to implement directional sampling concretely.

To consider the points and directions that affect the image, our program has to look something like Listing 15.1.

Listing 15.1: High-level rendering structure.

```
for each visible point P with direction ω_o from it to pixel center (x,y):
    sum = 0
    for each incident light direction ω_i at P:
        sum += light scattered at P from ω_i to ω_o
    pixel[x, y] = sum
```

15.2.2 Visible Points

Now we devise a strategy for representing points in the scene, finding those that are visible and scattering the light incident on them to the camera.

For the scene representation, we'll work within some of the common rendering approximations described in Chapter 14. None of these are so fundamental as to prevent us from later replacing them with more accurate models.

Assume that we only need to model surfaces that form the boundaries of objects. "Object" is a subjective term; a **surface** is technically the interface between volumes with homogeneous physical properties. Some of these objects are what everyday language recognizes as such, like a block of wood or the water in a pool. Others are not what we are accustomed to considering as objects, such as air or a vacuum.

We'll model these surfaces as triangle meshes. We ignore the surrounding medium of air and assume that all the meshes are closed so that from the outside of an object one can never see the inside. This allows us to consider only single-sided triangles. We choose the convention that the vertices of a triangular face, seen from the outside of the object, are in counterclockwise order around the face. To approximate the shading of a smooth surface using this triangle mesh, we model the surface normal at a point on a triangle pointing in the direction of the barycentric interpolation of prespecified normal vectors at its vertices. These normals only affect shading, so silhouettes of objects will still appear polygonal.

Chapter 27 explores how surfaces scatter light in great detail. For simplicity, we begin by assuming all surfaces scatter incoming light equally in all directions, in a sense that we'll make precise presently. This kind of scattering is called Lambertian, as you saw in Chapter 6, so we're rendering a Lambertian surface. The

color of a surface is determined by the relative amount of light scattered at each wavelength, which we represent with a familiar RGB triple.

This surface mesh representation describes all the *potentially* visible points at the set of locations $\{P\}$. To render a given pixel, we must determine which potentially visible points project to the center of that pixel. We then need to select the scene point closest to the camera. That point is the *actually* visible point for the pixel center. The radiance—a measure of light that's defined precisely in Section 26.7.2, and usually denoted with the letter L—arriving from that point and passing through the pixel is proportional to the light incident on the point and the point's reflectivity.

To find the nearest potentially visible point, we first convert the outer loop of Listing 15.1 (see the next section) into an iteration over both pixel centers (which correspond to rays) and triangles (which correspond to surfaces). A common way to accomplish this is to replace "`for each visible point`" with two nested loops, one over the pixel centers and one over the triangles. Either can be on the outside. Our choice of which is the new outermost loop has significant structural implications for the rest of the renderer.

15.2.3 Ray Casting: Pixels First

Listing 15.2: Ray-casting pseudocode.

```
for each pixel position (x,y):
    let R be the ray through (x,y) from the eye
    for each triangle T:
        let P be the intersection of R and T (if any)
        sum = 0
        for each direction:
            sum += ...
        if P is closer than previous intersections at this pixel:
            pixel[x, y] = sum
```

Consider the strategy where the outermost loop is over pixel centers, shown in Listing 15.2. This strategy is called **ray casting** because it creates one ray per pixel and casts it at every surface. It generalizes to an algorithm called **ray tracing,** in which the innermost loop recursively casts rays at each direction, but let's set that aside for the moment.

Ray casting lets us process each pixel to completion independently. This suggests parallel processing of pixels to increase performance. It also encourages us to keep the entire scene in memory, since we don't know which triangles we'll need at each pixel. The structure suggests an elegant way of eventually processing the aforementioned indirect light: Cast more rays from the innermost loop.

15.2.4 Rasterization: Triangles First

Now consider the strategy where the outermost loop is over triangles shown in Listing 15.3. This strategy is called **rasterization,** because the inner loop is typically implemented by marching along the rows of the image, which are called **rasters.** We could choose to march along columns as well. The choice of rows is historical and has to do with how televisions were originally constructed. Cathode ray tube (CRT) displays scanned an image from left to right, top to bottom, the way that English text is read on a page. This is now a widespread convention:

Unless there is an explicit reason to do otherwise, images are stored in row-major order, where the element corresponding to 2D position (x, y) is stored at index `(x + y * width)` in the array.

Listing 15.3: Rasterization pseudocode; O denotes the origin, or eyepoint.

```
1  for each pixel position (x,y):
2      closest[x,y] = ∞
3
4  for each triangle T:
5      for each pixel position (x,y):
6          let R be the ray through (x,y) from the eye
7          let P be the intersection of R and T
8          if P exists:
9              sum = 0
10             for each direction:
11                 sum += ...
12                 if the distance to P is less than closest[x,y]:
13                     pixel[x,y] = sum
14                     closest[x,y] = |P − O|
```

Rasterization allows us to process each triangle to completion independently.[2] This has several implications. It means that we can render much larger scenes than we can hold in memory, because we only need space for one triangle at a time. It suggests triangles as the level of parallelism. The properties of a triangle can be maintained in registers or cache to avoid memory traffic, and only one triangle needs to be memory-resident at a time. Because we consider adjacent pixels consecutively for a given triangle, we can approximate derivatives of arbitrary expressions across the surface of a triangle by finite differences between pixels. This is particularly useful when we later become more sophisticated about sampling strategies because it allows us to adapt our sampling rate to the rate at which an underlying function is changing in screen space.

Note that the conditional on line 12 in Listing 15.3 refers to the closest *previous* intersection at a pixel. Because that intersection was from a different triangle, that value must be stored in a 2D array that is parallel to the image. This array did not appear in our original pseudocode or the ray-casting design. Because we now touch each pixel many times, we must maintain a data structure for each pixel that helps us resolve visibility between visits. Only two distances are needed for comparison: the distance to the current point and to the previously closest point. We don't care about points that have been previously considered but are farther away than the closest, because they are hidden behind the closest point and can't affect the image. The *closest* array stores the distance to the previously closest point at each pixel. It is called a **depth buffer** or a **z-buffer.** Because computing the distance to a point is potentially expensive, depth buffers are often implemented to encode some other value that has the same comparison properties as distance along a ray. Common choices are $-z_P$, the z-coordinate of the point P, and $-1/z_P$. Recall that the camera is facing along the negative z-axis, so these are related to distance from the $z = 0$ plane in which the camera sits.

2. If you're worried that to process one triangle we have to loop through all the pixels in the image, even though the triangle does not cover most of them, then your worries are well founded. See Section 15.6.2 for a better strategy. We're starting this way to keep the code as nearly parallel to the ray-casting structure as possible.

For now we'll use the more intuitive choice of distance from P to the origin, $|P - O|$.

The depth buffer has the same dimensions as the image, so it consumes a potentially significant amount of memory. It must also be accessed atomically under a parallel implementation, so it is a potentially slow synchronization point. Chapter 36 describes alternative algorithms for resolving visibility under rasterization that avoid these drawbacks. However, depth buffers are by far the most widely used method today. They are extremely efficient in practice and have predictable performance characteristics. They also offer advantages beyond the sampling process. For example, the known depth at each pixel at the end of 3D rendering yields a "2.5D" result that enables compositing of multiple render passes and post-processing filters, such as artificial defocus blur.

This depth comparison turns out to be a fundamental idea, and it is now supported by special fixed-function units in graphics hardware. A huge leap in computer graphics performance occurred when this feature emerged in the early 1980s.

15.3 Implementation Platform

15.3.1 Selection Criteria

The kinds of choices discussed in this section are important. We want to introduce them now, and we want them all in one place so that you can refer to them later. Many of them will only seem natural to you after you've worked with graphics for a while. So read this section now, set it aside, and then read it again in a month.

In your studies of computer graphics you will likely learn many APIs and software design patterns. For example, Chapters 2, 4, 6, and 16 teach the 2D and 3D WPF APIs and some infrastructure built around them.

Teaching that kind of content is expressly *not* a goal of this chapter. This chapter is about creating algorithms for sampling light. The implementation serves to make the algorithms concrete and provide a test bed for later exploration. Although learning a specific platform is not a goal, learning the issues to consider when evaluating a platform *is* a goal; in this section we describe those issues.

We select one specific platform, a subset of the G3D Innovation Engine [http://g3d.sf.net] Version 9, for the code examples. You may use this one, or some variation chosen after considering the same issues weighed by your own goals and computing environment. In many ways it is better if your platform—language, compiler, support classes, hardware API—is *not* precisely the same as the one described here. The platform we select includes only a minimalist set of support classes. This keeps the presentation simple and generic, as suits a textbook. But you're developing software on today's technology, not writing a textbook that must serve independent of currently popular tools.

Since you're about to invest a lot of work on this renderer, a richer set of support classes will make both implementation and debugging easier. You can compile our code directly against the support classes in G3D. However, if you have to rewrite it slightly for a different API or language, this will force you to actually read every line and consider why it was written in a particular manner. Maybe your chosen language has a different syntax than ours for passing a parameter by value

instead of reference, for example. In the process of redeclaring a parameter to make this syntax change, you should think about why the parameter was passed by value in the first place, and whether the computational overhead or software abstraction of doing so is justified.

To avoid distracting details, for the low-level renderers we'll write the image to an array in memory and then stop. Beyond a trivial PPM-file writing routine, we will not discuss the system-specific methods for saving that image to disk or displaying it on-screen in this chapter. Those are generally straightforward, but verbose to read and tedious to configure. The PPM routine is a proof of concept, but it is for an inefficient format and requires you to use an external viewer to check each result. G3D and many other platforms have image-display and image-writing procedures that can present the images that you've rendered more conveniently.

For the API-based hardware rasterizer, we will use a lightly abstracted subset of the OpenGL API that is representative of most other hardware APIs. We'll intentionally skip the system-specific details of initializing a hardware context and exploiting features of a particular API or GPU. Those transient aspects can be found in your favorite API or GPU vendor's manuals.

Although we can largely ignore the surrounding platform, we must still choose a programming language. It is wise to choose a language with reasonably high-level abstractions like classes and operator overloading. These help the algorithm shine through the source code notation.

It is also wise to choose a language that can be compiled to efficient native code. That is because even though performance should not be the ultimate consideration in graphics, it is a fairly important one. Even simple video game scenes contain millions of polygons and are rendered for displays with millions of pixels. We'll start with one triangle and one pixel to make debugging easier and then quickly grow to hundreds of each in this chapter. The constant overhead of an interpreted language or a managed memory system cannot affect the asymptotic behavior of our program. However, it can be the difference between our renderer producing an image in two seconds or two hours . . . and debugging a program that takes two hours to run is very unpleasant.

Computer graphics code tends to combine high-level classes containing significant state, such as those representing scenes and objects, with low-level classes (a.k.a. "records", "structs") for storing points and colors that have little state and often expose that which they do contain directly to the programmer. A real-time renderer can easily process billions of those low-level classes per second. To support that, one typically requires a language with features for efficiently creating, destroying, and storing such classes. Heap memory management for small classes tends to be expensive and thwart cache efficiency, so stack allocation is typically the preferred solution. Language features for passing by value and by constant reference help the programmer to control cloning of both large and small class instances.

Finally, hardware APIs tend to be specified at the machine level, in terms of bytes and pointers (as abstracted by the C language). They also often require manual control over memory allocation, deallocation, types, and mapping to operate efficiently.

To satisfy the demands of high-level abstraction, reasonable performance for hundreds to millions of primitives and pixels, and direct manipulation of memory, we work within a subset of C++. Except for some minor syntactic variations, this subset should be largely familiar to Java and Objective C++ programmers. It is

a superset of C and can be compiled directly as native (nonmanaged) C#. For all of these reasons, and because there is a significant tools and library ecosystem built for it, C++ happens to be the dominant language for implementing renderers today. Thus, our choice is consistent with showing you how renderers are really implemented.

Note that many hardware APIs also have wrappers for higher-level languages, provided by either the API vendor or third parties. Once you are familiar with the basic functionality, we suggest that it may be more productive to use such a wrapper for extensive software development on a hardware API.

15.3.2 Utility Classes

This chapter assumes the existence of obvious utility classes, such as those sketched in Listing 15.4. For these, you can use equivalents of the WPF classes, the Direct3D API versions, the built-in GLSL, Cg, and HLSL shading language versions, or the ones in G3D, or you can simply write your own. Following common practice, the `Vector3` and `Color3` classes denote the axes over which a quantity varies, but not its units. For example, `Vector3` always denotes three spatial axes but may represent a unitless direction vector at one code location and a position in meters at another. We use a type alias to at least distinguish points from vectors (which are differences of points).

Listing 15.4: Utility classes.

```
#define INFINITY (numeric_limits<float>::infinity())

class Vector2 { public: float x, y;    ... };
class Vector3 { public: float x, y, z; ... };
typedef Vector2 Point2;
typedef Vector3 Point3;
class Color3 { public: float r, g, b;  ... };
class Radiance3 Color3;
class Power3 Color3;

class Ray {
private:
    Point3  m_origin;
    Vector3 m_direction;

public:
    Ray(const Point3& org, const Vector3& dir) :
       m_origin(org), m_direction(dir) {}

    const Point3& origin() const { return m_origin; }
    const Vector3& direction() const { return m_direction; }
    ...
};
```

Observe that some classes, such as `Vector3`, expose their representation through public member variables, while others, such as `Ray`, have a stronger abstraction that protects the internal representation behind methods. The exposed classes are the workhorses of computer graphics. Invoking methods to access their fields would add significant syntactic distraction to the implementation of any function. Since the byte layouts of these classes must be known and fixed to interact directly with hardware APIs, they cannot be strong abstractions and it makes

sense to allow direct access to their representation. The classes that protect their representation are ones whose representation we may (and truthfully, will) later want to change. For example, the internal representation of `Triangle` in this listing is an array of vertices. If we found that we computed the edge vectors or face normal frequently, then it might be more efficient to extend the representation to explicitly store those values.

For images, we choose the underlying representation to be an array of `Radiance3`, each array entry representing the radiance incident at the center of one pixel in the image. We then wrap this array in a class to present it as a 2D structure with appropriate utility methods in Listing 15.5.

Listing 15.5: An `Image` *class.*

```
1  class Image {
2  private:
3     int                 m_width;
4     int                 m_height;
5     std::vector<Radiance3> m_data;
6
7     int PPMGammaEncode(float radiance, float displayConstant) const;
8
9  public:
10
11    Image(int width, int height) :
12         m_width(width), m_height(height), m_data(width * height) {}
13
14    int width() const { return m_width; }
15
16    int height() const { return m_height; }
17
18    void set(int x, int y, const Radiance3& value) {
19       m_data[x + y * m_width] = value;
20    }
21
22    const Radiance3& get(int x, int y) const {
23       return m_data[x + y * m_width];
24    }
25
26    void save(const std::string& filename, float displayConstant=15.0f) const;
27 };
```

Under C++ conventions and syntax, the `&` following a type in a declaration indicates that the corresponding variable or return value will be passed by reference. The `m_` prefix avoids confusion between member variables and methods or parameters with similar names. The `std::vector` class is the dynamic array from the standard library.

One could imagine a more feature-rich image class with bounds checking, documentation, and utility functions. Extending the implementation with these is a good exercise.

The `set` and `get` methods follow the historical row-major mapping from a 2D to a 1D array. Although we do not need it here, note that the reverse mapping from a 1D index `i` to the 2D indices `(x, y)` is

```
x = i % width; y = i / width
```

where `%` is the C++ integer modulo operation.

When `width` is a power of two, that is, `width` $= 2^k$, it is possible to perform both the forward and reverse mappings using bitwise operations, since

$$a \mod 2^k = a \,\&\, (2^k - 1) \tag{15.1}$$

$$a/2^k = a \gg k \tag{15.2}$$

$$a \cdot 2^k = a \ll k, \tag{15.3}$$

for fixed-point values. Here we use » as the operator to shift the bits of the left operand to the right by the value of the right operand, and & as the bitwise AND operator.

This is one reason that many graphics APIs historically required power-of-two image dimensions (another is MIP mapping). One can always express a number that is not a power of two as the sum of multiple powers of two. In fact, that's what binary encoding does! For example, $640 = 512 + 128$, so $x + 640y = x + (y\ll 9) + (y\ll 7)$.

Inline Exercise 15.2: Implement forward and backward mappings from integer (x, y) pixel locations to 1D array indices i, for a typical HD resolution of 1920×1080, using only bitwise operations, addition, and subtraction.

Familiarity with the bit-manipulation methods for mapping between 1D and 2D arrays is important now so that you can understand other people's code. It will also help you to appreciate how hardware-accelerated rendering might implement some low-level operations and why a rendering API might have certain constraints. However, this kind of micro-optimization will not substantially affect the performance of your renderer at this stage, so it is not yet worth including.

Our `Image` class stores physically meaningful values. The natural measurement of the light arriving along a ray is in terms of **radiance,** whose definition and precise units are described in Chapter 26. The image typically represents the light *about* to fall onto each pixel of a sensor or area of a piece of film. It doesn't represent the sensor response process.

Displays and image files tend to work with arbitrarily scaled 8-bit display values that map nonlinearly to radiance. For example, if we set the display pixel value to 64, the display pixel does not emit twice the radiance that it does when we set the same pixel to 32. This means that we cannot display our image faithfully by simply rescaling radiance to display values. In fact, the relationship involves an exponent commonly called gamma, as described briefly below and at length in Section 28.12.

Assume some multiplicative factor `d` that rescales the radiance values in an image so that the largest value we wish to represent maps to 1.0 and the smallest maps to 0.0. This fills the role of the camera's shutter and aperture. The user will select this value as part of the scene definition. Mapping it to a GUI slider is often a good idea.

Historically, most images stored 8-bit values whose meanings were ill-specified. Today it is more common to specify what they mean. An image that

actually stores radiance values is informally said to store **linear radiance,** indicating that the pixel value varies linearly with the radiance (see Chapter 17). Since the radiance range of a typical outdoor scene with shadows might span six orders of magnitude, the data would suffer from perceptible quantization artifacts were it reduced to eight bits per channel. However, human perception of brightness is roughly logarithmic. This means that distributing precision nonlinearly can reduce the perceptual error of a small bit-depth approximation. **Gamma encoding** is a common practice for distributing values according to a fractional power law, where $1/\gamma$ is the power. This encoding curve roughly matches the logarithmic response curve of the human visual system. Most computer displays accept input already gamma-encoded along the sRGB standard curve, which is about $\gamma = 2.2$. Many image file formats, such as PPM, also default to this gamma encoding. A routine that maps a radiance value to an 8-bit display value with a gamma value of 2.2 is:

```
int Image::PPMGammaEncode(float radiance, float d) const {
    return int(pow(std::min(1.0f, std::max(0.0f, radiance * d)),
                   1.0f / 2.2f) * 255.0f);
}
```

Note that $x^{1/2.2} \approx \sqrt{x}$. Because they are faster than arbitrary exponentiation on most hardware, square root and square are often employed in real-time rendering as efficient $\gamma = 2.0$ encoding and decoding methods.

The `save` routine is our bare-bones method for exporting data from the renderer for viewing. It saves the image in human-readable PPM format [P+10] and is implemented in Listing 15.6.

Listing 15.6: Saving an image to an ASCII RGB PPM file.

```
void Image::save(const std::string& filename, float d) const {
    FILE* file = fopen(filename.c_str(), "wt");
    fprintf(file, "P3 %d %d 255\n", m_width, m_height);
    for (int y = 0; y < m_height; ++y) {
        fprintf(file, "\n# y = %d\n", y);
        for (int x = 0; x < m_width; ++x) {
            const Radiance3& c(get(x, y));
            fprintf(file, "%d %d %d\n",
                    PPMGammaEncode(c.r, d),
                    PPMGammaEncode(c.g, d),
                    PPMGammaEncode(c.b, d));
        }
    }
    fclose(file);
}
```

This is a useful snippet of code beyond its immediate purpose of saving an image. The structure appears frequently in 2D graphics code. The outer loop iterates over rows. It contains any kind of per-row computation (in this case, printing the row number). The inner loop iterates over the columns of one row and performs the per-pixel operations. Note that if we wished to amortize the cost of computing `y * m_width` inside the `get` routine, we could compute that as a per-row operation and merely accumulate the 1-pixel offsets in the inner loop. We do not do so in this case because that would complicate the code without providing a measurable performance increase, since writing a formatted text file would remain slow compared to performing one multiplication per pixel.

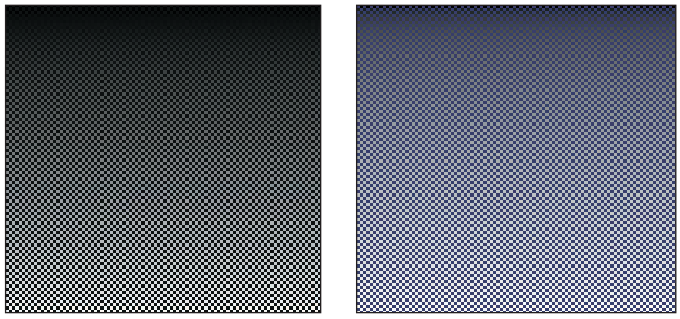


Figure 15.2: A pattern for testing the `Image` *class. The pattern is a checkerboard of* 1*-pixel squares that alternate between* $1/10\,\mathrm{W}/(\mathrm{m}^2\,\mathrm{sr})$ *in the blue channel and a vertical gradient from* 0 *to* 10*. (a) Viewed with* `deviceGamma` = 1.0 *and* `displayConstant` = 1.0*, which makes dim squares appear black and gives the appearance of a linear change in* brightness*. (b) Displayed more correctly with* `deviceGamma` = 2.0*, where the linear* radiance *gradient correctly appears as a nonlinear brightness ramp and the dim squares are correctly visible. (The conversion to a printed image or your online image viewer may further affect the image.)*

The PPM format is slow for loading and saving, and consumes lots of space when storing images. For those reasons, it is rarely used outside academia. However, it is convenient for data interchange between programs. It is also convenient for debugging small images for three reasons. The first is that it is easy to read and write. The second is that many image programs and libraries support it, including Adobe Photoshop and xv. The third is that we can open it in a text editor to look directly at the (gamma-corrected) pixel values.

After writing the image-saving code, we displayed the simple pattern shown in Figure 15.2 as a debugging aid. If you implement your own image saving or display mechanism, consider doing something similar. The test pattern alternates dark blue pixels with ones that form a gradient. The reason for creating the single-pixel checkerboard pattern is to verify that the image was neither stretched nor cropped during display. If it was, then one or more thin horizontal or vertical lines would appear. (If you are looking at this image on an electronic display, you may see such patterns, indicating that your viewing software is indeed stretching it.) The motivation for the gradient is to determine whether gamma correction is being applied correctly. A linear radiance gradient should appear as a nonlinear brightness gradient, when displayed correctly. Specifically, it should primarily look like the brighter shades. The pattern on the left is shown without gamma correction. The gradient appears to have linear brightness, indicating that it is not displayed correctly. The pattern on the right is shown with gamma correction. The gradient correctly appears to be heavily shifted toward the brighter shaders.

Note that we made the darker squares blue, yet in the left pattern—without gamma correction—they appear black. That is because gamma correction helps make darker shades more visible, as in the right image. This hue shift is another argument for being careful to always implement gamma correction, beyond the tone shift. Of course, we don't know the exact characteristics of the display

(although one can typically determine its gamma exponent) or the exact viewing conditions of the room, so precise color correction and tone mapping is beyond our ability here. However, the simple act of applying gamma correction arguably captures some of the most important aspects of that process and is computationally inexpensive and robust.

Inline Exercise 15.3: Two images are shown below. Both have been gamma-encoded with $\gamma = 2.0$ for printing and online display. The image on the left is a gradient that has been rendered to give the impression of linear *brightness*. It should appear as a linear color ramp. The image on the right was rendered with linear *radiance* (it is the checkerboard on the right of Figure 15.2 without the blue squares). It should appear as a nonlinear color ramp. The image was rendered at 200×200 pixels. What equation did we use to compute the value (in $[0, 1]$) of the pixel at (x, y) for the gradient image on the left?

Linear brightness

Linear radiance

15.3.3 Scene Representation

Listing 15.7 shows a `Triangle` class. It stores each triangle by explicitly storing each vertex. Each vertex has an associated normal that is used exclusively for shading; the normals do not describe the actual geometry. These are sometimes called **shading normals.** When the vertex normals are identical to the normal to the plane of the triangle, the triangle's shading will appear consistent with its actual geometry. When the normals diverge from this, the shading will mimic that of a curved surface. Since the silhouette of the triangle will still be polygonal, this effect is most convincing in a scene containing many small triangles.

Listing 15.7: Interface for a `Triangle` *class.*

```
1  class Triangle {
2  private:
3      Point3    m_vertex[3];
4      Vector3   m_normal[3];
5      BSDF      m_bsdf;
6
7  public:
8
9      const Point3& vertex(int i) const { return m_vertex[i]; }
10     const Vector3& normal(int i) const { return m_normal[i]; }
11     const BSDF& bsdf() const { return m_bsdf; }
12     ...
13 };
```

We also associate a `BSDF` class value with each triangle. This describes the material properties of the surface modeled by the triangle. It is described in Section 15.4.5. For now, think of this as the color of the triangle.

The representation of the triangle is concealed by making the member variables private. Although the implementation shown contains methods that simply return those member variables, you will later use this abstraction boundary to create a more efficient implementation of the triangle. For example, many triangles may share the same vertices and bidirectional scattering distribution functions (BSDFs), so this representation is not very space-efficient. There are also properties of the triangle, such as the edge lengths and geometric normal, that we will find ourselves frequently recomputing and could benefit from storing explicitly.

Inline Exercise 15.4: Compute the size in bytes of one `Triangle`. How big is a 1M triangle mesh? Is that reasonable? How does this compare with the size of a stored mesh file, say, in the binary 3DS format or the ASCII OBJ format? What are other advantages, beyond space reduction, of sharing vertices between triangles in a mesh?

Listing 15.8 shows our implementation of an omnidirectional point light source. We represent the power it emits at three wavelengths (or in three wavelength bands), and the center of the emitter. Note that emitters are infinitely small in our representation, so they are not themselves visible. If we wish to see the source appear in the final rendering we need to either add geometry around it or explicitly render additional information into the image. We will do neither explicitly in this chapter, although you may find that these are necessary when debugging your illumination code.

Listing 15.8: Interface for a uniform point luminaire—a light source.

```
1 class Light {
2 public:
3     Point3    position;
4
5     /** Over the entire sphere. */
6     Power3    power;
7 };
```

Listing 15.9 describes the scene as sets of triangles and lights. Our choice of arrays for the implementation of these sets imposes an ordering on the scene. This is convenient for ensuring a reproducible environment for debugging. However, for now we are going to create that ordering in an arbitrary way, and that choice may affect performance and even our image in some slight ways, such as resolving ties between which surface is closest at an intersection. More sophisticated scene data structures may include additional structure in the scene and impose a specific ordering.

Listing 15.9: Interface for a scene represented as an unstructured list of triangles and light sources.

```
1 class Scene {
2 public:
3     std::vector<Triangle> triangleArray;
4     std::vector<Light>    lightArray;
5 };
```

Listing 15.10 represents our camera. The camera has a pinhole aperture, an instantaneous shutter, and artificial near and far planes of constant (negative) z values. We assume that the camera is located at the origin and facing along the $-z$-axis.

Listing 15.10: Interface for a pinhole camera at the origin.

```
1 class Camera {
2 public:
3     float zNear;
4     float zFar;
5     float fieldOfViewX;
6
7     Camera() : zNear(-0.1f), zFar(-100.0f), fieldOfViewX(PI / 2.0f) {}
8 };
```

We constrain the horizontal field of view of the camera to be `fieldOfViewX`. This is the measure of the angle from the center of the leftmost pixel to the center of the rightmost pixel along the horizon in the camera's view in radians (it is shown later in Figure 15.3). During rendering, we will compute the aspect ratio of the target image and implicitly use that to determine the vertical field of view. We could alternatively specify the vertical field of view and compute the horizontal field of view from the aspect ratio.

15.3.4 A Test Scene

We'll test our renderers on a scene that contains one triangle whose vertices are

`Point3(0,1,-2)`, `Point3(-1.9,-1,-2)`, and `Point3(1.6,-0.5,-2)`,

and whose vertex normals are

`Vector3( 0.0f, 0.6f, 1.0f).direction()`,
`Vector3(-0.4f,-0.4f, 1.0f).direction()`, and
`Vector3( 0.4f,-0.4f, 1.0f).direction()`.

We create one light source in the scene, located at `Point3(1.0f,3.0f,1.0f)` and emitting power `Power3(10, 10, 10)`. The camera is at the origin and is facing along the $-z$-axis, with y increasing upward in screen space and x increasing to the right. The image has size 800×500 and is initialized to dark blue.

This choice of scene data was deliberate, because when debugging it is a good idea to choose configurations that use nonsquare aspect ratios, nonprimary colors, asymmetric objects, etc. to help find cases where you have accidentally swapped axes or color channels. Having distinct values for the properties of each vertex also makes it easier to track values through code. For example, on this triangle, you can determine which vertex you are examining merely by looking at its x-coordinate.

On the other hand, the camera is the standard one, which allows us to avoid transforming rays and geometry. That leads to some efficiency and simplicity in the implementation and helps with debugging because the input data maps exactly to the data rendered, and in practice, most rendering algorithms operate in the camera's reference frame anyway.

Inline Exercise 15.5: *Mandatory; do not continue until you have done this:* Draw a schematic diagram of this scene from three viewpoints.

1. The orthographic view from infinity facing along the x-axis. Make z increase to the right and y increase upward. Show the camera and its field of view.
2. The orthographic view from infinity facing along the $-y$-axis. Make x increase to the right and z increase downward. Show the camera and its field of view. Draw the vertex normals.
3. The perspective view from the camera, facing along the $-z$-axis; the camera should not appear in this image.

15.4 A Ray-Casting Renderer

We begin the ray-casting renderer by expanding and implementing our initial pseudocode from Listing 15.2. It is repeated in Listing 15.11 with more detail.

Listing 15.11: Detailed pseudocode for a ray-casting renderer.

```
for each pixel row y:
   for each pixel column x:
      let R = ray through screen space position (x + 0.5, y + 0.5)
      closest = ∞
      for each triangle T:
         d = intersect(T, R)
         if (d < closest)
              closest = d
              sum = 0
              let P be the intersection point
              for each direction ω_i:
                 sum += light scattered at P from ω_i to ω_o

              image[x, y] = sum
```

The three loops iterate over every ray and triangle combination. The body of the for-each-triangle loop verifies that the new intersection is closer than previous observed ones, and then shades the intersection. We will abstract the operation of ray intersection and sampling into a helper function called `sampleRayTriangle`. Listing 15.12 gives the interface for this helper function.

Listing 15.12: Interface for a function that performs ray-triangle intersection and shading.

```
bool sampleRayTriangle(const Scene& scene, int x, int y,
    const Ray& R, const Triangle& T,
    Radiance3& radiance, float& distance);
```

The specification for `sampleRayTriangle` is as follows. It tests a particular ray against a triangle. If the intersection exists and is closer than all previously observed intersections for this ray, it computes the radiance scattered toward the viewer and returns `true`. The innermost loop therefore sets the value of pixel (x, y)

to the radiance `L_o` passing through its center from the closest triangle. Radiance from farther triangles is not interesting because it will (conceptually) be blocked by the back of the closest triangle and never reach the image. The implementation of `sampleRayTriangle` appears in Listing 15.15.

To render the entire image, we must invoke `sampleRayTriangle` once for each pixel center and for each triangle. Listing 15.13 defines `rayTrace`, which performs this iteration. It takes as arguments a box within which to cast rays (see Section 15.4.4). We use `L_o` to denote the radiance from the triangle; the subscript "o" is for "outgoing".

Listing 15.13: Code to trace one ray for every pixel between (`x0`, `y0`) *and* (`x1-1`, `y1-1`)*, inclusive.*

```
/** Trace eye rays with origins in the box from [x0, y0] to (x1, y1).*/
void rayTrace(Image& image, const Scene& scene,
  const Camera& camera, int x0, int x1, int y0, int y1) {

  // For each pixel
  for (int y = y0; y < y1; ++y) {
    for (int x = y0; x < x1; ++x) {

      // Ray through the pixel
      const Ray& R = computeEyeRay(x + 0.5f, y + 0.5f, image.width(),
                          image.height(), camera);

      // Distance to closest known intersection
      float distance = INFINITY;
      Radiance3 L_o;

      // For each triangle
      for (unsigned int t = 0; t < scene.triangleArray.size(); ++t){
        const Triangle& T = scene.triangleArray[t];

        if (sampleRayTriangle(scene, x, y, R, T, L_o, distance)) {
          image.set(x, y, L_o);
        }
      }
    }
  }
}
```

To invoke `rayTrace` on the entire image, we will use the call:

```
rayTrace(image, scene, camera, 0, image.width(), 0, image.height());
```

15.4.1 Generating an Eye Ray

Assume the camera's center of projection is at the origin, $(0, 0, 0)$, and that, in the camera's frame of reference, the y-axis points upward, the x-axis points to the right, and the z-axis points out of the screen. Thus, the camera is facing along its own $-z$-axis, in a right-handed coordinate system. We can transform any scene to this coordinate system using the transformations from Chapter 11.

We require a utility function, `computeEyeRay`, to find the ray through the center of a pixel, which in screen space is given by $(x + 0.5, y + 0.5)$ for integers x and y. Listing 15.14 gives an implementation. The key geometry is depicted in

Figure 15.3. The figure is a top view of the scene in which x increases to the right and z increases downward. The near plane appears as a horizontal line, and the `start` point is on that plane, along the line from the camera at the origin to the center of a specific pixel.

To implement this function we needed to parameterize the camera by either the image plane depth or the desired field of view. Field of view is a more intuitive way to specify a camera, so we previously chose that parameterization when building the scene representation.

Listing 15.14: Computing the ray through the center of pixel (x, y) on a `width` × `height` *image.*

```
Ray computeEyeRay(float x, float y, int width, int height, const Camera& camera) {
    const float aspect = float(height) / width;

    // Compute the side of a square at z = -1 based on our
    // horizontal left-edge-to-right-edge field of view
    const float s = -2.0f * tan(camera.fieldOfViewX * 0.5f);

    const Vector3& start =
        Vector3( (x / width - 0.5f) * s,
                 -(y / height - 0.5f) * s * aspect, 1.0f) * camera.zNear;

    return Ray(start, start.direction());
}
```

We choose to place the ray origin on the near (sometimes called hither) clipping plane, at $z =$ `camera.zNear`. We could start rays at the origin instead of the near plane, but starting at the near plane will make it easier for results to line up precisely with our rasterizer later.

The ray direction is the direction from the center of projection (which is at the origin, $(0, 0, 0)$) to the ray `start` point, so we simply normalize `start` point.

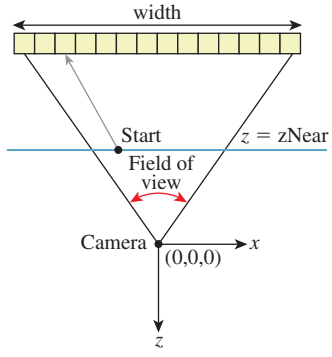


Figure 15.3: The ray through a pixel center in terms of the image resolution and the camera's horizontal field of view.

Inline Exercise 15.6: By the rules of Chapter 7, we should compute the ray direction as `(start - Vector3(0,0,0)).direction()`. That makes the camera position explicit, so we are less likely to introduce a bug if we later change the camera. This arises simply from strongly typing the code to match the underlying mathematical types. On the other hand, our code is going to be full of lines like this, and consistently applying correct typing might lead to more harm from obscuring the algorithm than benefit from occasionally finding an error. It is a matter of personal taste and experience (we can somewhat reconcile our typing with the math by claiming that `P.direction()` on a point `P` returns the direction to the point, rather than "normalizing" the point).

Rewrite `computeEyeRay` using the distinct `Point` and `Vector` abstractions from Chapter 7 to get a feel for how this affects the presentation and correctness. If this inspires you, it's quite reasonable to restructure all the code in this chapter that way, and doing so is a valuable exercise.

Note that the y-coordinate of the `start` is negated. This is because `y` is in 2D screen space, with a "$y =$ down" convention, and the ray is in a 3D coordinate system with a "$y =$ up" convention.

To specify the vertical field of view instead of the horizontal one, replace `fieldOfViewX` with `fieldOfViewY` and insert the line `s /= aspect`.

15.4.1.1 Camera Design Notes

The C++ language offers both functions and methods as procedural abstractions. We have presented `computeEyeRay` as a function that takes a `Camera` parameter to distinguish the "support code" `Camera` class from the ray-tracer-specific code that you are adding. As you move forward through the book, consider refactoring the support code to integrate auxiliary functions like this directly into the appropriate classes. (If you are using an existing 3D library as support code, it is likely that the provided camera class already contains such a method. In that case, it is worth implementing the method once as a function here so that you have the experience of walking through and debugging the routine. You can later discard your version in favor of a canonical one once you've reaped the educational value.)

A software engineering tip: Although we have chosen to forgo small optimizations, it is still important to be careful to use references (e.g., `Image&`) to avoid excess copying of arguments and intermediate results. There are two related reasons for this, and neither is about the performance of *this* program.

The first reason is that we want to be in the habit of avoiding excessive copying. A `Vector3` occupies 12 bytes of memory, but a full-screen `Image` is a few megabytes. If we're conscientious about never copying data unless we want copy semantics, then we won't later accidentally copy an `Image` or other large structure. Memory allocation and copy operations can be surprisingly slow and will bloat the memory footprint of our program. The time cost of copying data isn't just a constant overhead factor on performance. Copying the image once per pixel, in the inner loop, would change the ray caster's asymptotic run time from $O(n)$ in the number of pixels to $O(n^2)$.

The second reason is that experienced programmers rely on a set of idioms that are chosen to avoid bugs. Any deviation from those attracts attention, because it is a potential bug. One such convention in C++ is to pass each value as a const reference unless otherwise required, for the long-term performance reasons just described. So code that doesn't do so takes longer for an experienced programmer to review because of the need to check that there isn't an error or performance implication whenever an idiom is not followed. If you are an experienced C++ programmer, then such idioms help you to read the code. If you are not, then either ignore all the ampersands and treat this as pseudocode, or use it as an opportunity to become a better C++ programmer.

15.4.1.2 Testing the Eye-Ray Computation

We need to test `computeEyeRay` before continuing. One way to do this is to write a unit test that computes the eye rays for specific pixels and then compares them to manually computed results. That is always a good testing strategy. In addition to that, we can visualize the eye rays. Visualization is a good way to quickly see the result of many computations. It allows us to more intuitively check results, and to identify patterns of errors if they do not match what we expected.

In this section, we'll visualize the directions of the rays. The same process can be applied to the origins. The directions are the more common location for an error and conveniently have a bounded range, which make them both more important and easier to visualize.

A natural scheme for visualizing a direction is to interpret the (x, y, z) fields as (r, g, b) color triplets. The conversion of ray direction to pixel color is of course a gross violation of units, but it is a really useful debugging technique and we aren't expecting anything principled here anyway.

Because each ordinate is on the interval $[-1, 1]$, we rescale them to the range $[0, 1]$ by $r = (x + 1)/2$. Our image display routines also apply an exposure function, so we need to scale the resultant intensity down by a constant on the order of the inverse of the exposure value. Temporarily inserting the following line:

```
image.set(x, y, Color3(R.direction() + Vector3(1, 1, 1)) / 5);
```

into `rayTrace` in place of the `sampleRayTriangle` call should yield an image like that shown in Figure 15.4. (The factor of $1/5$ scales the debugging values to a reasonable range for our output, which was originally calibrated for radiance; we found a usable constant for this particular example by trial and error.) We expect the x-coordinate of the ray, which here is visualized as the color red, to increase from a minimum on the left to a maximum on the right. Likewise, the (3D) y-coordinate, which is visualized as green, should increase from a minimum at the bottom of the image to a maximum at the top. If your result varies from this, examine the pattern you observe and consider what kind of error could produce it. We will revisit visualization as a debugging technique later in this chapter, when testing the more complex intersection routine.

Figure 15.4: Visualization of eye-ray directions.

15.4.2 Sampling Framework: Intersect and Shade

Listing 15.15 shows the code for sampling a triangle with a ray. This code doesn't perform any of the heavy lifting itself. It just computes the values needed for `intersect` and `shade`.

Listing 15.15: Sampling the intersection and shading of one triangle with one ray.

```
bool sampleRayTriangle(const Scene& scene, int x, int y, const Ray& R,
     const Triangle& T, Radiance3& radiance, float& distance) {

    float weight[3];
    const float d = intersect(R, T, weight);

    if (d >= distance) {
        return false;
    }

    // This intersection is closer than the previous one
    distance = d;

    // Intersection point
    const Point3& P = R.origin() + R.direction() * d;

    // Find the interpolated vertex normal at the intersection
    const Vector3& n = (T.normal(0) * weight[0] +
                        T.normal(1) * weight[1] +
                        T.normal(2) * weight[2]).direction();

    const Vector3& w_o = -R.direction();

```

```
23      shade(scene, T, P, n, w_o, radiance);
24
25      // Debugging intersect: set to white on any intersection
26      //radiance = Radiance3(1, 1, 1);
27
28      // Debugging barycentric
29      //radiance = Radiance3(weight[0], weight[1], weight[2]) / 15;
30
31      return true;
32  }
```

The `sampleRayTriangle` routine returns `false` if there was no intersection closer than `distance`; otherwise, it updates `distance` and `radiance` and returns `true`.

When invoking this routine, the caller passes the `distance` to the closest currently known intersection, which is initially `INFINITY` (let `INFINITY = std::numeric_limits<T>::infinity()` in C++, or simply `1.0/0.0`). We will design the `intersect` routine to return `INFINITY` when no intersection exists between `R` and `T` so that a missed intersection will never cause `sampleRayTriangle` to return `true`.

Placing the `(d >= distance)` test before the shading code is an optimization. We would still obtain correct results if we always computed the shading before testing whether the new intersection is in fact the closest. This is an important optimization because the `shade` routine may be arbitrarily expensive. In fact, in a full-featured ray tracer, almost all computation time is spent inside `shade`, which recursively samples additional rays. We won't discuss further shading optimizations in this chapter, but you should be aware of the importance of an early termination when another surface is known to be closer.

Note that the order of the triangles in the calling routine (`rayTrace`) affects the performance of the routine. If the triangles are in back-to-front order, then we will shade each one, only to reject all but the closest. This is the worst case. If the triangles are in front-to-back order, then we will shade the first and reject the rest without further shading effort. We could ensure the best performance always by separating `sampleRayTriangle` into two auxiliary routines: one to find the closest intersection and one to shade that intersection. This is a common practice in ray tracers. Keep this in mind, but do not make the change yet. Once we have written the rasterizer renderer, we will consider the space and time implications of such optimizations under both ray casting and rasterization, which gives insights into many variations on each algorithm.

We'll implement and test `intersect` first. To do so, comment out the call to `shade` on line 23 and uncomment either of the debugging lines below it.

15.4.3 Ray-Triangle Intersection

We'll find the intersection of the eye ray and a triangle in two steps, following the method described in Section 7.9 and implemented in Listing 15.16. This method first intersects the line containing the ray with the plane containing the triangle. It then solves for the barycentric weights to determine if the intersection is within the triangle. We need to ignore intersections with the back of the single-sided triangle and intersections that occur along the part of the line that is not on the ray.

The same weights that we use to determine if the intersection is within the triangle are later useful for interpolating per-vertex properties, such as shading

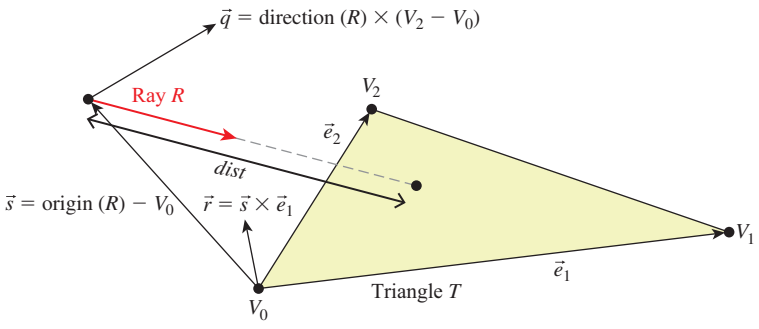


Figure 15.5: Variables for computing the intersection of a ray and a triangle (see Listing 15.16).

normals. We structure our implementation to return the weights to the caller. The caller could use either those or the distance traveled along the ray to find the intersection point. We return the distance traveled because we know that we will later need that anyway to identify the closest intersection to the viewer in a scene with many triangles. We return the barycentric weights for use in interpolation.

Figure 15.5 shows the geometry of the situation. Let R be the ray and T be the triangle. Let $\vec{e}_1$ be the edge vector from V_0 to V_1 and $\vec{e}_2$ be the edge vector from V_0 to V_2. Vector $\vec{q}$ is orthogonal to both the ray and $\vec{e}_2$. Note that if $\vec{q}$ is also orthogonal to $\vec{e}_1$, then the ray is parallel to the triangle and there is no intersection. If $\vec{q}$ is in the negative hemisphere of $\vec{e}_1$ (i.e., "points away"), then the ray travels away from the triangle.

Vector $\vec{s}$ is the displacement of the ray origin from V_0, and vector $\vec{r}$ is the cross product of $\vec{s}$ and $\vec{e}_1$. These vectors are used to construct the barycentric weights, as shown in Listing 15.16.

Variable `a` is the rate at which the ray is approaching the triangle, multiplied by twice the area of the triangle. This is not obvious from the way it is computed here, but it can be seen by applying a triple-product identity relation:

$$\begin{aligned}
&\text{Let } d = \texttt{R.direction()} \\
&\text{Let } area = |\vec{e}_2 \times \vec{e}_1|/2 \\
&a = \vec{e}_1 \cdot q = \vec{e}_1 \cdot d \times \vec{e}_2 = d \cdot \vec{e}_2 \times \vec{e}_1 = -(d \cdot n) \cdot 2 \cdot area, \qquad (15.4)
\end{aligned}$$

since the direction of $\vec{e}2 \times \vec{e}1$ is opposite the triangle's geometric normal n. The particular form of this expression chosen in the implementation is convenient because the `q` vector is needed again later in the code for computing the barycentric weights.

There are several cases where we need to compare a value against zero. The two `epsilon` constants guard these comparisons against errors arising from limited numerical precision.

The comparison `a <= epsilon` detects two cases. If `a` is zero, then the ray is parallel to the triangle and never intersects it. In this case, the code divided by zero many times, so other variables may be infinity or not-a-number. That's irrelevant, since the first test expression will still make the entire test expression `true`. If `a` is negative, then the ray is traveling away from the triangle and will never intersect it. Recall that `a` is the rate at which the ray approaches the triangle, multiplied by the area of the triangle. If `epsilon` is too large, then intersections with triangles

Listing 15.16: Ray-triangle intersection (derived from [MT97])

```
float intersect(const Ray& R, const Triangle& T, float weight[3]) {
    const Vector3& e1 = T.vertex(1) - T.vertex(0);
    const Vector3& e2 = T.vertex(2) - T.vertex(0);
    const Vector3& q = R.direction().cross(e2);

    const float a = e1.dot(q);

    const Vector3& s = R.origin() - T.vertex(0);
    const Vector3& r = s.cross(e1);

    // Barycentric vertex weights
    weight[1] = s.dot(q) / a;
    weight[2] = R.direction().dot(r) / a;
    weight[0] = 1.0f - (weight[1] + weight[2]);

    const float dist = e2.dot(r) / a;

    static const float epsilon = 1e-7f;
    static const float epsilon2 = 1e-10;

    if ((a <= epsilon) || (weight[0] < -epsilon2) ||
        (weight[1] < -epsilon2) || (weight[2] < -epsilon2) ||
        (dist <= 0.0f)) {
        // The ray is nearly parallel to the triangle, or the
        // intersection lies outside the triangle or behind
        // the ray origin: "infinite" distance until intersection.
        return INFINITY;
    } else {
        return dist;
    }
}
```

will be missed at glancing angles, and this missed intersection behavior will be more likely to occur at triangles with large areas than at those with small areas. Note that if we changed the test to `fabs(a)<= epsilon`, then triangles would have two sides. This is not necessary for correct models of real, opaque objects; however, for rendering mathematical models or models with errors in them it can be convenient. Later we will depend on optimizations that allow us to quickly cull the (approximately half) of the scene representing back faces, so we choose to render single-sided triangles here for consistency.

The `epsilon2` constant allows a ray to intersect a triangle slightly outside the bounds of the triangle. This ensures that triangles that share an edge completely cover pixels along that edge despite numerical precision limits. If `epsilon2` is too small, then single-pixel holes will very occasionally appear on that edge. If it is too large, then all triangles will visibly appear to be too large.

Depending on your processor architecture, it may be faster to perform an early test and potential return rather than allowing not-a-number and infinity propagation in the ill-conditioned case where `a` ≈ 0. Many values can also be precomputed, for example, the edge lengths of the triangle, or at least be reused within a single intersection, for example, `1.0f / a`. There's a cottage industry of optimizing this intersection code for various architectures, compilers, and scene types (e.g., [MT97] for scalar processors versus [WBB08] for vector processors). Let's forgo those low-level optimizations and stick to high-level algorithmic decisions. In practice, most ray casters spend very little time in the ray intersection code anyway. The fastest way to determine if a ray intersects a triangle is to never ask

that question in the first place. That is, in Chapter 37, we will introduce data structures that quickly and conservatively eliminate whole sets of triangles that the ray could not possibly intersect, without ever performing the ray-triangle intersection. So optimizing this routine now would only complicate it without affecting our long-term performance profile.

Our renderer only processes triangles. We could easily extend it to render scenes containing any kind of primitive for which we can provide a ray intersection solution. Surfaces defined by low-order equations, like the plane, rectangle, sphere, and cylinder, have explicit solutions. For others, such as bicubic patches, we can use root-finding methods.

15.4.4 Debugging

We now verify that the intersection code code is correct. (The code we've given you *is* correct, but if you invoked it with the wrong parameters, or introduced an error when porting to a different language or support code base, then you need to learn how to find that error.) This is a good opportunity for learning some additional graphics debugging tricks, all of which demonstrate the Visual Debugging principle.

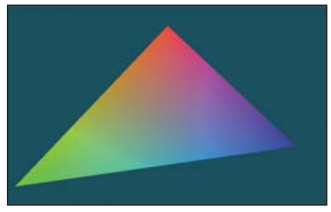


Figure 15.6: The single triangle scene visualized with color equal to barycentric weight for debugging the intersection code.

It would be impractical to manually examine every intersection result in a debugger or printout. That is because the `rayTrace` function invokes `intersect` thousands of times. So instead of examining individual results, we visualize the barycentric coordinates by setting the radiance at a pixel to be proportional to the barycentric coordinates following the Visual Debugging principle. Figure 15.6 shows the correct resultant image. If your program produces a comparable result, then your program is probably nearly correct.

What should you do if your result looks different? You can't examine every result, and if you place a breakpoint in `intersect`, then you will have to step through hundreds of ray casts that miss the triangle before coming to the interesting intersection tests.

This is why we structured `rayTrace` to trace within a caller-specified rectangle, rather than the whole image. We can invoke the ray tracer on a single pixel from `main()`, or better yet, create a debugging interface where clicking on a pixel with the mouse invokes the single-pixel trace on the selected pixel. By setting breakpoints or printing intermediate results under this setup, we can investigate why an artifact appears at a specific pixel. For one pixel, the math is simple enough that we can also compute the desired results by hand and compare them to those produced by the program.

In general, even simple graphics programs tend to have large amounts of data. This may be many triangles, many pixels, or many frames of animation. The processing for these may also be running on many threads, or on a GPU. Traditional debugging methods can be hard to apply in the face of such numerous data and massive parallelism. Furthermore, the graphics development environment may preclude traditional techniques such as printing output or setting breakpoints. For example, under a hardware rendering API, your program is executing on an embedded processor that frequently has no access to the console and is inaccessible to your debugger.

Fortunately, three strategies tend to work well for graphics debugging.

1. Use assertions liberally. These cost you nothing in the optimized version of the program, pass silently in the debug version when the program operates

correctly, and break the program at the test location when an assertion is violated. Thus, they help to identify failure cases without requiring that you manually step through the correct cases.

2. Immediately reduce to the minimal test case. This is often a single-triangle scene with a single light and a single pixel. The trick here is to find the combination of light, triangle, and pixel that produces incorrect results. Assertions and the GUI click-to-debug scheme work well for that.
3. Visualize intermediate results. We have just rendered an image of the barycentric coordinates of eye-ray intersections with a triangle for a 400,000-pixel image. Were we to print out these values or step through them in the debugger, we would have little chance of recognizing an incorrect value in that mass of data. If we see, for example, a black pixel, or a white pixel, or notice that the red and green channels are swapped, then we may be able to deduce the nature of the error that caused this, or at least know which inputs cause the routine to fail.

15.4.5 Shading

We are now ready to implement `shade`. This routine computes the incident radiance at the intersection point `P` and how much radiance scatters back along the eye ray to the viewer.

Let's consider only light transport paths directly from the source to the surface to the camera. Under this restriction, there is no light arriving at the surface from any directions except those to the lights. So we only need to consider a finite number of ω_i values. Let's also assume for the moment that there is always a line of sight to the light. This means that there will (perhaps incorrectly) be no shadows in the rendered image.

Listing 15.17 iterates over the light sources in the scene (note that we have only one in our test scene). For each light, the loop body computes the distance and direction to that light from the point being shaded. Assume that lights emit uniformly in all directions and are at finite locations in the scene. Under these assumptions, the incident radiance `L_i` at point `P` is proportional to the total power of the source divided by the square of the distance between the source and `P`. This is because at a given distance, the light's power is distributed equally over a sphere of that radius. Because we are ignoring shadowing, let the `visible` function always return `true` for now. In the future it will return `false` if there is no line of sight from the source to `P`, in which case the light should contribute no incident radiance.

The outgoing radiance to the camera, `L_o`, is the sum of the fraction of incident radiance that scatters in that direction. We abstract the scattering function into a BSDF. We implement this function as a class so that it can maintain state across multiple invocations and support an inheritance hierarchy. Later in this book, we will also find that it is desirable to perform other operations beyond invoking this function; for example, we might want to sample with respect to the probability distribution it defines. Using a class representation will allow us to later introduce additional methods for these operations.

The `evaluateFiniteScatteringDensity` method of that class evaluates the scattering function for the given incoming and outgoing angles. We always then take the product of this and the incoming radiance, modulated by the cosine

Listing 15.17: The single-bounce shading code.

```
void shade(const Scene& scene, const Triangle& T, const Point3& P,
   const Vector3& n, const Vector3& w_o, Radiance3& L_o) {

    L_o = Color3(0.0f, 0.0f, 0.0f);

    // For each direction (to a light source)
    for (unsigned int i = 0; i < scene.lightArray.size(); ++i) {
       const Light& light = scene.lightArray[i];

       const Vector3& offset = light.position - P;
       const float distanceToLight = offset.length();
       const Vector3& w_i = offset / distanceToLight;

       if (visible(P, w_i, distanceToLight, scene)) {
           const Radiance3& L_i = light.power / (4 * PI * square(distanceToLight));

             // Scatter the light
             L_o +=
                L_i *
                T.bsdf(n).evaluateFiniteScatteringDensity(w_i, w_o) *
                max(0.0, dot(w_i, n));
       }
    }
}
```

of the angle between `w_i` and `n` to account for the projected area over which incident radiance is distributed (by the Tilting principle).

15.4.6 Lambertian Scattering

The simplest implementation of the BSDF assumes a surface appears to be the same brightness independent of the viewer's orientation. That is, `evaluateFiniteScatteringDensity` returns a constant. This is called **Lambertian reflectance,** and it is a good model for matte surfaces such as paper and flat wall paint. It is also trivial to implement. Listing 15.18 gives the implementation (see Section 14.9.1 for a little more detail and Chapter 29 for a lot more). It has a single member, `lambertian`, that is the "color" of the surface. For energy conservation, this value should have all fields on the range $[0, 1]$.

Listing 15.18: Lambertian BSDF implementation, following Listing 14.6.

```
class BSDF {
public:
    Color3 k_L;

    /** Returns f = L_o / (L_i * w_i.dot(n)) assuming
    incident and outgoing directions are both in the
    positive hemisphere above the normal */
    Color3 evaluateFiniteScatteringDensity
      (const Vector3& w_i, const Vector3& w_o) const {
        return k_L / PI;
    }
};
```

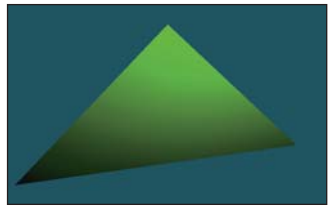


Figure 15.7: A green Lambertian triangle.

Figure 15.7 shows our triangle scene rendered with the Lambertian BSDF using `k_L=Color3(0.0f, 0.0f, 0.8f)`. Because our triangle's vertex

normals are deflected away from the plane defined by the vertices, the triangle appears curved. Specifically, the bottom of the triangle is darker because the `w_i.dot(n)` term in line 20 of Listing 15.17 falls off toward the bottom of the triangle.

15.4.7 Glossy Scattering

The Lambertian surface appears dull because it has no highlight. A common approach for producing a more interesting shiny surface is to model it with something like the Blinn-Phong scattering function. An implementation of this function with the energy conservation factor from Sloan and Hoffmann [AMHH08, 257] is given in Listing 15.19. See Chapter 27 for a discussion of the origin of this function and alternatives to it. This is a variation on the shading function that we saw back in Chapter 6 in WPF, only now we are implementing it instead of just adjusting the parameters on a black box. The basic idea is simple: Extend the Lambertian BSDF with a large radial peak when the normal lies close to halfway between the incoming and outgoing directions. This peak is modeled by a cosine raised to a power since that is easy to compute with dot products. It is scaled so that the outgoing radiance never exceeds the incoming radiance and so that the sharpness and total intensity of the peak are largely independent parameters.

Listing 15.19: Blinn-Phong BSDF scattering density.

```
1  class BSDF {
2  public:
3      Color3  k_L;
4      Color3  k_G;
5      float   s;
6      Vector3 n;
7      ...
8
9      Color3 evaluateFiniteScatteringDensity(const Vector3& w_i,
10         const Vector3& w_o) const {
11         const Vector3& w_h = (w_i + w_o).direction();
12         return
13             (k_L + k_G * ((s + 8.0f) *
14                 powf(std::max(0.0f, w_h.dot(n)), s) / 8.0f)) /
15             PI;
16
17     }
18 };
```

For this BSDF, choose `lambertian` + `glossy` < 1 at each color channel to ensure energy conservation, and `glossySharpness` typically in the range $[0, 2000]$. The `glossySharpness` is on a logarithmic scale, so it must be moved in larger increments as it becomes larger to have the same perceptual impact.

Figure 15.8 shows the green triangle rendered with the normalized Blinn-Phong BSDF. Here, `k_L=Color3(0.0f, 0.0f, 0.8f)`, `k_G=Color3(0.2f, 0.2f, 0.2f)`, and `s=100.0f`.

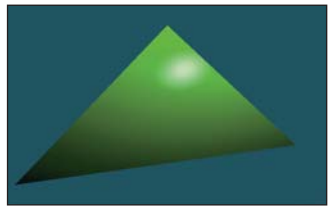


Figure 15.8: Triangle rendered with a normalized Blinn-Phong BSDF.

15.4.8 Shadows

The `shade` function in Listing 15.17 only adds the illumination contribution from a light source if there is an unoccluded line of sight between that source and the point P being shaded. Areas that *are* occluded are therefore darker. This absence of light is the phenomenon that we recognize as a shadow.

In our implementation, the line-of-sight visibility test is performed by the `visible` function, which is supposed to return `true` if and only if there is an unoccluded line of sight. While working on the shading routine we temporarily implemented `visible` to *always* return `true`, which means that our images contain no shadows. We now revisit the `visible` function in order to implement shadows.

We already have a powerful tool for evaluating line of sight: the `intersect` function. The light source is not visible from P if there is some intersection with another triangle. So we can test visibility simply by iterating over the scene again, this time using the *shadow ray* from P to the light instead of from the camera to P. Of course, we could also test rays from the light *to* P.

Listing 15.20 shows the implementation of `visible`. The structure is very similar to that of `sampleRayTriangle`. It has three major differences in the details. First, instead of shading the intersection, if we find any intersection we immediately return `false` for the visibility test. Second, instead of casting rays an infinite distance, we terminate when they have passed the light source. That is because we don't care about triangles past the light—they could not possibly cast shadows on P. Third and finally, we don't really start our shadow ray cast at P. Instead, we offset it slightly along the ray direction. This prevents the ray from reintersecting the surface containing P as soon as it is cast.

Listing 15.20: Line-of-sight visibility test, to be applied to shadow determination.

```
bool visible(const Vector3& P, const Vector3& direction, float
    distance, const Scene& scene){
    static const float rayBumpEpsilon = 1e-4;
    const Ray shadowRay(P + direction * rayBumpEpsilon, direction);

    distance -= rayBumpEpsilon;

    // Test each potential shadow caster to see if it lies between P and the light
    float ignore[3];
    for (unsigned int s = 0; s < scene.triangleArray.size(); ++s) {
        if (intersect(shadowRay, scene.triangleArray[s], ignore) < distance) {
            // This triangle is closer than the light
            return false;
        }
    }

    return true;
}
```

Our single-triangle scene is insufficient for testing shadows. We require one object to cast shadows and another to receive them. A simple extension is to add a quadrilateral "ground plane" onto which the green triangle will cast its shadow. Listing 15.21 gives code to create this scene. Note that this code also adds another triangle with the same vertices as the green one but the opposite winding order. Because our triangles are single-sided, the green triangle would not cast a shadow. We need to add the back of that surface, which will occlude the rays cast upward toward the light from the ground.

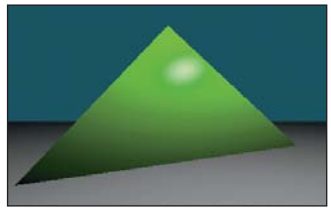


Figure 15.9: The green triangle scene extended with a two-triangle gray ground "plane." A back surface has also been added to the green triangle.

Inline Exercise 15.7: Walk through the intersection code to verify the claim that without the second "side," the green triangle would cast no shadow.

Figure 15.9 shows how the new scene should render *before* you implement shadows. If you do not see the ground plane under your own implementation, the most likely error is that you failed to loop over all triangles in one of the ray-casting routines.

Listing 15.21: Scene-creation code for a two-sided triangle and a ground plane.

```
void makeOneTriangleScene(Scene& s) { s.triangleArray.resize(1);

    s.triangleArray[0] =
      Triangle(Vector3(0,1,-2), Vector3(-1.9,-1,-2), Vector3(1.6,-0.5,-2),
         Vector3(0,0.6f,1).direction(),
         Vector3(-0.4f,-0.4f, 1.0f).direction(),
         Vector3(0.4f,-0.4f, 1.0f).direction(),
         BSDF(Color3::green() * 0.8f,Color3::white() * 0.2f, 100));

    s.lightArray.resize(1);
    s.lightArray[0].position = Point3(1, 3, 1);
    s.lightArray[0].power = Color3::white() * 10.0f;
}

void makeTrianglePlusGroundScene(Scene& s) {
    makeOneTriangleScene(s);

    // Invert the winding of the triangle
    s.triangleArray.push_back
      (Triangle(Vector3(-1.9,-1,-2), Vector3(0,1,-2),
          Vector3(1.6,-0.5,-2), Vector3(-0.4f,-0.4f, 1.0f).direction(),
          Vector3(0,0.6f,1).direction(), Vector3(0.4f,-0.4f, 1.0f).direction(),
          BSDF(Color3::green() * 0.8f,Color3::white() * 0.2f, 100)));

    // Ground plane
    const float groundY = -1.0f;
    const Color3 groundColor = Color3::white() * 0.8f;
    s.triangleArray.push_back
      (Triangle(Vector3(-10, groundY, -10), Vector3(-10, groundY, -0.01f),
          Vector3(10, groundY, -0.01f),
          Vector3::unitY(), Vector3::unitY(), Vector3::unitY(), groundColor));

    s.triangleArray.push_back
      (Triangle(Vector3(-10, groundY, -10), Vector3(10, groundY, -0.01f),
          Vector3(10, groundY, -10),
          Vector3::unitY(), Vector3::unitY(), Vector3::unitY(), groundColor));
}
```

Figure 15.10 shows the scene rendered with `visible` implemented correctly. If the `rayBumpEpsilon` is too small, then **shadow acne** will appear on the green triangle. This artifact is shown in Figure 15.11. An alternative to starting the ray artificially far from P is to explicitly exclude the previous triangle from the shadow ray intersection computation. We chose not to do that because, while appropriate for unstructured triangles, it would be limiting to maintain that custom ray intersection code as our scene became more complicated. For example, we would like to later abstract the scene data structure from a simple array of triangles. The abstract data structure might internally employ a hash table or tree and have complex methods. Pushing the notion of excluding a surface into such a data structure could complicate that data structure and compromise its general-purpose use. Furthermore, although we are rendering only triangles now,

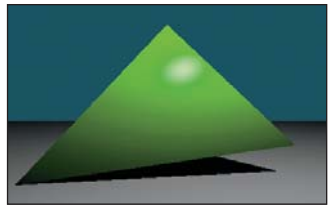


Figure 15.10: A four-triangle scene, with ray-cast shadows implemented via the `visible` *function. The green triangle is two-sided.*

we might wish to render other primitives in the future, such as spheres or implicit surfaces. Such primitives can intersect a ray multiple times. If we assume that the shadow ray never intersects the current surface, those objects would never self-shadow.

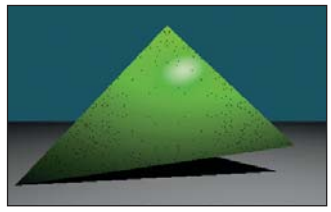


*Figure 15.11: The dark dots on the green triangle are **shadow acne** caused by self-shadowing. This artifact occurs when the shadow ray immediately intersects the triangle that was being shaded.*

15.4.9 A More Complex Scene

Now that we've built a renderer for one or two triangles, it is no more difficult to render scenes containing many triangles. Figure 15.12 shows a shiny, gold-colored teapot on a white ground plane. We parsed a file containing the vertices of the corresponding triangle mesh, appended those triangles to the `Scene`'s triangle array, and then ran the existing renderer on it. This scene contains about 100 triangles, so it renders about 100 times slower than the single-triangle scene. We can make arbitrarily more complex geometry and shading functions for the renderer. We are only limited by the quality of our models and our rendering performance, both of which will be improved in subsequent chapters.

Figure 15.12: A scene composed of many triangles.

This scene looks impressive (at least, relative to the single triangle) for two reasons. First, we see some real-world phenomena, such as shiny highlights, shadows, and nice gradients as light falls off. These occurred naturally from following the geometric relationships between light and surfaces in our implementation.

Second, the image resembles a recognizable object, specifically, a teapot. Unlike the illumination phenomena, nothing in *our* code made this look like a teapot. We simply loaded a triangle list from a data file that someone (originally, Jim Blinn) happened to have manually constructed. This teapot triangle list is a classic model in graphics. You can download the triangle mesh version used here from `http://graphics.cs.williams.edu/data` among other sources. Creating models like this is a separate problem from rendering, discussed in Chapter 22 and many others. Fortunately, there are many such models available, so we can defer the modeling problem while we discuss rendering.

We can learn a lesson from this. A strength and weakness of computer graphics as a technical field is that often the data contributes more to the quality of the final image than the algorithm. The same algorithm rendered the teapot and the green triangle, but the teapot looks more impressive because the data is better. Often a truly poor approximation algorithm will produce stunning results when a master artist creates the input—the commercial success of the film and game industries has largely depended on this fact. Be aware of this when judging algorithms based on rendered results, and take advantage of it by importing good artwork to demonstrate your own algorithms.

15.5 Intermezzo

To render a scene, we needed to iterate over both triangles and pixels. In the previous section, we arbitrarily chose to arrange the pixel loop on the outside and the triangle loop on the inside. That yielded the ray-casting algorithm. The ray-casting algorithm has three nice properties: It somewhat mimics the underlying physics, it separates the visibility routine from the shading routine, and it leverages the same ray-triangle intersection routine for both eye rays and shadow rays.

Admittedly, the relationship between ray casting and physics at the level demonstrated here is somewhat tenuous. Real photons propagate along rays from the light source to a surface to an eye, and we traced that path backward. Real photons don't all scatter into the camera. Most photons from the light source scatter away from the camera, and much of the light that *is* scattered toward the camera from a surface didn't arrive at that surface directly from the light. Nonetheless, an algorithm for sampling light along rays is a very good starting point for sampling photons, and it matches our intuition about how light should propagate. You can probably imagine improvements that would better model the true scattering behavior of light. Much of the rest of this book is devoted to such models.

In the next section, we invert the nesting order of the loops to yield a **rasterizer algorithm.** We then explore the implications of that change. We already have a working ray tracer to compare against. Thus, we can easily test the correctness of our changes by comparing against the ray-traced image and intermediate results. We also have a standard against which to measure the properties of the new algorithm. As you read the following section and implement the program that it describes, consider how the changes you are making affect code clarity, modularity, and efficiency. Particularly consider efficiency in both a wall-clock time and an asymptotic run time sense. Think about applications for which one of rasterization and ray casting is a better fit than the other.

These issues are not restricted to our choice of the outer loop. All high-performance renderers subdivide the scene and the image in sophisticated ways. The implementer must choose how to make these subdivisions and for each must again revisit whether to iterate first over pixels (i.e., ray directions) or triangles. The same considerations arise at every level, but they are evaluated differently based on the expected data sizes at that level and the machine architecture.

15.6 Rasterization

We now move on to implement the rasterizing renderer, and compare it to the ray-casting renderer, observing places where each is more efficient and how the restructuring of the code allows for these efficiencies. The relatively tiny change turns out to have substantial impact on computation time, communication demands, and cache coherence.

15.6.1 Swapping the Loops

Listing 15.22 shows an implementation of `rasterize` that corresponds closely to `rayTrace` with the nesting order inverted. The immediate implication of inverting the loop order is that we must store the distance to the closest known intersection at each pixel in a large buffer (`depthBuffer`), rather than in a single float. This is because we no longer process a single pixel to completion before moving to another pixel, so we must store the intermediate processing state. Some implementations store the depth as a distance along the z-axis, or as the inverse of that distance. We choose to store distance along an eye ray to more closely match the ray-caster structure.

The same intermediate state problem arises for the ray `R`. We could create a buffer of rays. In practice, the rays are fairly cheap to recompute and don't justify storage, and we will soon see alternative methods for eliminating the per-pixel ray computation altogether.

Listing 15.22: Rasterizer implemented by simply inverting the nesting order of the loops from the ray tracer, but adding a `DepthBuffer`.

```
void rasterize(Image& image, const Scene& scene, const Camera& camera){

  const int w = image.width(), h = image.height();
  DepthBuffer depthBuffer(w, h, INFINITY);

  // For each triangle
  for (unsigned int t = 0; t < scene.triangleArray.size(); ++t) {
    const Triangle& T = scene.triangleArray[t];

    // Very conservative bounds: the whole screen
    const int x0 = 0;
    const int x1 = w;

    const int y0 = 0;
    const int y1 = h;

    // For each pixel
    for (int y = y0; y < y1; ++y) {
      for (int x = x0; x < x1; ++x) {
        const Ray& R = computeEyeRay(x, y, w, h, camera);

        Radiance3 L_o;
        float distance = depthBuffer.get(x, y);
        if (sampleRayTriangle(scene, x, y, R, T, L_o, distance)) {
          image.set(x, y, L_o);
          depthBuffer.set(x, y, distance);
        }
      }
    }
  }
}
```

The `DepthBuffer` class is similar to `Image`, but it stores a single float at each pixel. Buffers over the image domain are common in computer graphics. This is a good opportunity for code reuse through polymorphism. In C++, the main polymorphic language feature is the template, which corresponds to templates in C# and generics in Java. One could design a templated `Buffer` class and then instantiate it for `Radiance3`, `float`, or whatever per-pixel data was desired. Since methods for saving to disk or gamma correction may not be appropriate for all template parameters, those are best left to subclasses of a specific template instance.

For the initial rasterizer implementation, this level of design is not required. You may simply implement `DepthBuffer` by copying the `Image` class implementation, replacing `Radiance3` with `float`, and deleting the display and save methods. We leave the implementation as an exercise.

Inline Exercise 15.8: Implement `DepthBuffer` as described in the text.

After implementing Listing 15.22, we need to test the rasterizer. At this time, we trust our ray tracer's results. So we run the rasterizer and ray tracer on the same scene, for which they should generate identical pixel values. As before, if the results are not identical, then the differences may give clues about the nature of the bug.

15.6.2 Bounding-Box Optimization

So far, we implemented rasterization by simply inverting the order of the for-each-triangle and for-each-pixel loops in a ray tracer. This performs many ray-triangle intersection tests that will fail. This is referred to as **poor sample test efficiency**.

We can significantly improve sample test efficiency, and therefore performance, on small triangles by only considering pixels whose centers are near the projection of the triangle. To do this we need a heuristic for efficiently bounding each triangle's projection. The bound must be conservative so that we never miss an intersection. The initial implementation already used a very conservative bound. It assumed that every triangle's projection was "near" *every* pixel on the screen. For large triangles, that may be true. For triangles whose true projection is small in screen space, that bound is too conservative.

The best bound would be a triangle's true projection, and many rasterizers in fact use that. However, there are significant amounts of boilerplate and corner cases in iterating over a triangular section of an image, so here we will instead use a more conservative but still reasonable bound: the 2D axis-aligned bounding box about the triangle's projection. For a large nondegenerate triangle, this covers about twice the number of pixels as the triangle itself.

> **Inline Exercise 15.9:** Why is it true that a large-area triangle covers at most about half of the samples of its bounding box? What happens for a *small* triangle, say, with an area smaller than one pixel? What are the implications for sample test efficiency if you know the size of triangles that you expect to render?

The axis-aligned bounding box, however, is straightforward to compute and will produce a significant speedup for many scenes. It is also the method favored by many hardware rasterization designs because the performance is very predictable, and for very small triangles the cost of computing a more accurate bound might dominate the ray-triangle intersection test.

The code in Listing 15.23 determines the bounding box of a triangle `T`. The code projects each vertex from the camera's 3D reference frame onto the plane $z = -1$, and then maps those vertices into the screen space 2D reference frame. This operation is handled entirely by the `perspectiveProject` helper function. The code then computes the minimum and maximum screen-space positions of the vertices and rounds them (by adding 0.5 and then casting the floating-point values to integers) to integer pixel locations to use as the for-each-pixel bounds.

The interesting work is performed by `perspectiveProject`. This inverts the process that `computeEyeRay` performed to find the eye-ray origin (before advancing it to the near plane). A direct implementation following that derivation is given in Listing 15.24. Chapter 13 gives a derivation for this operation as a matrix-vector product followed by a homogeneous division operation. That implementation is more appropriate when the perspective projection follows a series of other transformations that are also expressed as matrices so that the cost of the matrix-vector product can be amortized over all transformations. This version is potentially more computationally efficient (assuming that the constant subexpressions are precomputed) for the case where there are no other transformations; we also give this version to remind you of the derivation of the perspective projection matrix.

Listing 15.23: Projecting vertices and computing the screen-space bounding box.

```
Vector2 low(image.width(), image.height());
Vector2 high(0, 0);

for (int v = 0; v < 3; ++v) {
     const Vector2& X = perspectiveProject(T.vertex(v), image.width
         (), image.height(), camera);
     high = high.max(X);
     low = low.min(X);
}

const int x0 = (int)(low.x + 0.5f);
const int x1 = (int)(high.x + 0.5f);

const int y0 = (int)(low.y + 0.5f);
const int y1 = (int)(high.y + 0.5f);
```

Listing 15.24: Perspective projection.

```
Vector2 perspectiveProject(const Vector3& P, int width, int height,
     const Camera& camera) {
    // Project onto z = -1
    Vector2 Q(-P.x / P.z, -P.y / P.z);

    const float aspect = float(height) / width;

    // Compute the side of a square at z = -1 based on our
    // horizontal left-edge-to-right-edge field of view
    const float s = -2.0f * tan(camera.fieldOfViewX * 0.5f);

    Q.x = width * (-Q.x / s + 0.5f);
    Q.y = height * (Q.y / (s * aspect) + 0.5f);

    return Q;
}
```

Integrate the listings from this section into your rasterizer and run it. The results should exactly match the ray tracer and simpler rasterizer. Furthermore, it should be measurably faster than the simple rasterizer (although both are likely so fast for simple scenes that rendering seems instantaneous).

Simply verifying that the output matches is insufficient testing for this optimization. We're computing bounds, and we could easily have computed bounds that were way too conservative but still happened to cover the triangles for the test scene.

A good follow-up test and debugging tool is to plot the 2D locations to which the 3D vertices projected. To do this, iterate over all triangles again, after the scene has been rasterized. For each triangle, compute the projected vertices as before. But this time, instead of computing the bounding box, directly render the projected vertices by setting the corresponding pixels to white (of course, if there were bright white objects in the scene, another color, such as red, would be a better choice!). Our single-triangle test scene was chosen to be asymmetric. So this test should reveal common errors such as inverting an axis, or a half-pixel shift between the ray intersection and the projection routine.

15.6.3 Clipping to the Near Plane

Note that we can't apply `perspectiveProject` to points for which $z \geq 0$ to generate correct bounds in the invoking rasterizer. A common solution to this problem is to introduce some "near" plane $z = z_{\mathrm{n}}$ for $z_{\mathrm{n}} < 0$ and clip the triangle to it. This is the same as the near plane (`zNear` in the code) that we used earlier to compute the ray origin—since the rays began at the near plane, the ray tracer was also clipping the visible scene to the plane.

Clipping may produce a triangle, a degenerate triangle that is a line or point at the near plane, no intersection, or a quadrilateral. In the latter case we can divide the quadrilateral along one diagonal so that the output of the clipping algorithm is always either empty or one or two (possibly degenerate) triangles.

Clipping is an essential part of many rasterization algorithms. However, it can be tricky to implement well and distracts from our first attempt to simply produce an image by rasterization. While there are rasterization algorithms that never clip [Bli93, OG97], those are much more difficult to implement and optimize. For now, we'll ignore the problem and require that the entire scene is on the opposite side of the near plane from the camera. See Chapter 36 for a discussion of clipping algorithms.

15.6.4 Increasing Efficiency

15.6.4.1 2D Coverage Sampling

Having refactored our renderer so that the inner loop iterates over pixels instead of triangles, we now have the opportunity to substantially amortize much of the work of the ray-triangle intersection computation. Doing so will also build our insight for the relationship between a 3D triangle and its projection, and hint at how it is possible to gain the large constant performance factors that make the difference between offline and interactive rendering.

The first step is to transform the 3D ray-triangle intersection test by projection into a 2D point-in-triangle test. In rasterization literature, this is often referred to as **the visibility problem** or **visibility testing.** If a pixel center does not lie in the projection of a triangle, then the triangle is certainly "invisible" when we look through the center of projection of that pixel. However, the triangle might also be invisible for other reasons, such as a nearer triangle that occludes it, which is not considered here. Another term that has increasing popularity is more accurate: **coverage testing,** as in "Does the triangle *cover* the sample?" Coverage is a necessary but not sufficient condition for visibility.

We perform the coverage test by finding the 2D barycentric coordinates of every pixel center within the bounding box. If the 2D barycentric coordinates at a pixel center show that the pixel center lies within the projected triangle, then the 3D ray through the pixel center will also intersect the 3D triangle [Pin88]. We'll soon see that computing the 2D barycentric coordinates of several adjacent pixels can be done very efficiently compared to computing the corresponding 3D ray-triangle intersections.

15.6.4.2 Perspective-Correct Interpolation

For shading we will require the 3D barycentric coordinates of every ray-triangle intersection that we use, or some equivalent way of interpolating vertex attributes such as surface normals, texture coordinates, and per-vertex colors. We cannot

directly use the 2D barycentric coordinates from the coverage test for shading. That is because the 3D barycentric coordinates of a point on the triangle and the 2D barycentric coordinates of the *projection* of that point within the *projection* of the triangle are generally not equal. This can be seen in Figure 15.13. The figure shows a square in 3D with vertices A, B, C, and D, viewed from an oblique perspective so that its 2D projection is a trapezoid. The centroid of the 3D square is point E, which lies at the intersection of the diagonals. Point E is halfway between 3D edges AB and CD, yet in the 2D projection it is clearly much closer to edge CD. In terms of triangles, for triangle ABC, the 3D barycentric coordinates of E must be $w_A = \frac{1}{2}, w_B = 0, w_C = \frac{1}{2}$. The projection of E is clearly not halfway along the 2D line segment between the projections of A and C. (We saw this phenomenon in Chapter 10 as well.)

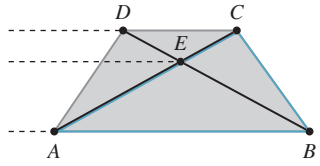


Figure 15.13: E is the centroid of square ABCD in 3D, but its projection is not the centroid of the projection of the square. This can be seen from the fact that the three dashed lines are not evenly spaced in 2D.

Fortunately, there is an efficient analog to 2D linear interpolation for projected 3D linear interpolation. This is interchangeably called **hyperbolic interpolation** [Bli93], **perspective-correct interpolation** [OG97], and **rational linear interpolation** [Hec90].

The perspective-correct interpolation method is simple. We can express it intuitively as, for each scalar vertex attribute u, linearly interpolate both $u' = u/z$ and $1/z$ in screen space. At each pixel, recover the 3D linearly interpolated attribute value from these by $u = u'/(1/z)$. See the following sidebar for a more formal explanation of why this works.

Let $u(x, y, z)$ be some scalar attribute (e.g., albedo, u texture coordinate) that varies linearly over the polygon. Two equivalent definitions may be more intuitive: (a) u is defined at vertices by specific values and varies by barycentric interpolation between them; (b) u has the form of a 3D plane equation, $u(x, y, z) = ax + by + cz + d$.

When the polygon is projected into screen space by the transformation $(x, y, z) \to (-x/z, -y/z, -1)$ for an image plane at $z = -1$, then **the function $-u(x, y, z)/z$ varies linearly in screen space.** Instead of linear interpolation in screen space, we need to perform a kind of "hyperbolic interpolation" to correctly evaluate u as follows.

Let P and Q be points on the 3D polygon, and let $u(P)$ and $u(Q)$ be some function that varies linearly across the plane of the 3D polygon evaluated at those points. Let $P' = -P/z_P$ be the projection of P and $Q' = -Q/z_Q$ be the projection of Q. At point M on line PQ that projects to $M' = \alpha P' + (1-\alpha)Q'$, the value of $u(M)$ satisfies

$$\frac{u(M)}{-z_M} = \alpha\frac{u(P)}{-z_P} + (1-\alpha)\frac{u(Q)}{-z_Q}, \tag{15.5}$$

while $-1/z_M$ satisfies

$$\frac{1}{-z_M} = \alpha\frac{1}{-z_P} + (1-\alpha)\frac{1}{-z_Q}. \tag{15.6}$$

Solving for $u(M)$ yields

$$u(M) = \frac{\alpha\frac{u(P)}{-z_P} + (1-\alpha)\frac{u(Q)}{-z_Q}}{\alpha\frac{1}{-z_P} + (1-\alpha)\frac{1}{-z_Q}}. \tag{15.7}$$

Because for each screen raster (i.e., row of pixels) we hold P and Q constant and vary α linearly, we can simplify the expression above to define a directly parameterized function $u'(\alpha)$:

$$u'(\alpha) = \frac{\alpha \cdot z_Q \cdot u(P) + (1-\alpha) z_P \cdot u(Q)}{\alpha \cdot z_Q + (1-\alpha) z_P}. \tag{15.8}$$

This is often more casually, but memorably, phrased as "In screen space, the perspective-correct interpolation of u is the quotient of the *linear interpolation* of u/z by the linear interpolation of $1/z$."

We can apply the perspective-correct interpolation strategy to any number of per-vertex attributes, including the vertex normals and texture coordinates. That leaves us with input data for our `shade` function, which remains unchanged from its implementation in the ray tracer.

15.6.4.3 2D Barycentric Weights

To implement the perspective-correct interpolation strategy, we need only find an expression for the 2D barycentric weights at the center of each pixel. Consider the barycentric weight corresponding to vertex A of a point Q within a triangle ABC. Recall from Section 7.9 that this weight is the ratio of the distance from Q to the line containing BC to the distance from A to the line containing BC, that is, it is the relative distance across the triangle from the opposite edge. Listing 15.25 gives code for computing a barycentric weight in 2D.

Listing 15.25: Computing one barycentric weight in 2D.

```
/** Returns the distance from Q to the line containing B and A. */
float lineDistance2D(const Point2& A, const Point2& B, const Point2& Q) {
    // Construct the line align:
    const Vector2 n(A.y - B.y, B.x - A.x);
    const float d = A.x * B.y - B.x * A.y;
    return (n.dot(Q) + d) / n.length();
}

/** Returns the barycentric weight corresponding to vertex A of Q in triangle ABC */
float bary2D(const Point2& A, const Point2& B, const Point2& C, const Point2& Q) {
    return lineDistance2D(B, C, Q) / lineDistance2D(B, C, A);
}
```

Inline Exercise 15.10: Under what condition could `lineDistance2D` return `0`, or `n.length()` be `0`, leading to a division by zero? Change your rasterizer to ensure that this condition never occurs. Why does this not affect the final rendering? What situation does this correspond to in a ray caster? How did we resolve that case when ray casting?

The rasterizer structure now requires a few changes from our previous version. It will need the post-projection vertices of the triangle after computing the bounding box in order to perform interpolation. We could either retain them from the bounding-box computation or just compute them again when needed later. We'll recompute the values when needed because it trades a small amount of efficiency

for a simpler interface to the bounding function, which makes the code easier to write and debug. Listing 15.26 shows the bounding box-function. The rasterizer must compute versions of the vertex attributes, which in our case are just the vertex normals, that are scaled by the $\frac{1}{z}$ value (which we call w) for the corresponding post-projective vertex. Both of those are per-triangle changes to the code. Finally, the inner loop must compute visibility from the 2D barycentric coordinates instead of from a ray cast. The actual shading computation remains unchanged from the original ray tracer, which is good—we're only looking at strategies for visibility, not shading, so we'd like each to be as modular as possible. Listing 15.27 shows the loop setup of the original rasterizer updated with the bounding-box and 2D barycentric approach. Listing 15.28 shows how the inner loops change.

Listing 15.26: Bounding box for the projection of a triangle, invoked by `rasterize3` *to establish the pixel iteration bounds.*

```
void computeBoundingBox(const Triangle& T, const Camera& camera,
                        const Image& image,
                        Point2 V[3], int& x0, int& y0, int& x1, int& y1) {

    Vector2 high(image.width(), image.height());
    Vector2 low(0, 0);

    for (int v = 0; v < 3; ++v) {
        const Point2& X = perspectiveProject(T.vertex(v), image.width(),
            image.height(), camera);
        V[v] = X;
        high = high.max(X);
        low = low.min(X);
    }

    x0 = (int)floor(low.x);
    x1 = (int)ceil(high.x);

    y0 = (int)floor(low.y);
    y1 = (int)ceil(high.y);
}
```

Listing 15.27: Iteration setup for a barycentric (edge align) rasterizer.

```
/** 2D barycentric evaluation w. perspective-correct attributes */
void rasterize3(Image& image, const Scene& scene,
       const Camera& camera){
  DepthBuffer depthBuffer(image.width(), image.height(), INFINITY);

  // For each triangle
  for (unsigned int t = 0; t < scene.triangleArray.size(); ++t) {
    const Triangle& T = scene.triangleArray[t];

    // Projected vertices
    Vector2 V[3];
    int x0, y0, x1, y1;
    computeBoundingBox(T, camera, image, V, x0, y0, x1, y1);

    // Vertex attributes, divided by -z
    float    vertexW[3];
    Vector3  vertexNw[3];
    Point3   vertexPw[3];
    for (int v = 0; v < 3; ++v) {
```

```
      const float w = -1.0f / T.vertex(v).z;
      vertexW[v] = w;
      vertexPw[v] = T.vertex(v) * w;
      vertexNw[v] = T.normal(v) * w;
    }

    // For each pixel
    for (int y = y0; y < y1; ++y) {
      for (int x = x0; x < x1; ++x) {
        // The pixel center
        const Point2 Q(x + 0.5f, y + 0.5f);
        ...

      }
    }
  }
}
```

Listing 15.28: Inner loop of a barycentric (edge align) rasterizer (see Listing 15.27 for the loop setup).

```
// For each pixel
for (int y = y0; y < y1; ++y) {
  for (int x = x0; x < x1; ++x) {
    // The pixel center
    const Point2 Q(x + 0.5f, y + 0.5f);

    // 2D Barycentric weights
    const float weight2D[3] =
      {bary2D(V[0], V[1], V[2], Q),
       bary2D(V[1], V[2], V[0], Q),
       bary2D(V[2], V[0], V[1], Q)};

    if ((weight2D[0]>0) && (weight2D[1]>0) && (weight2D[2]>0)) {
      // Interpolate depth
      float w = 0.0f;
      for (int v = 0; v < 3; ++v) {
        w += weight2D[v] * vertexW[v];
      }

      // Interpolate projective attributes, e.g., P', n'
      Point3 Pw;
      Vector3 nw;
      for (int v = 0; v < 3; ++v) {
        Pw += weight2D[v] * vertexPw[v];
        nw += weight2D[v] * vertexNw[v];
      }

      // Recover interpolated 3D attributes; e.g., P' -> P, n' -> n
      const Point3& P = Pw / w;
      const Vector3& n = nw / w;

      const float depth = P.length();
      // We could also use depth = z-axis distance: depth = -P.z

      // Depth test
      if (depth < depthBuffer.get(x, y)) {
        // Shade
        Radiance3  L_o;
        const Vector3& w_o = -P.direction();

        // Make the surface normal have unit length
        const Vector3& unitN = n.direction();
```

```
43          shade(scene, T, P, unitN, w_o, L_o);
44
45          depthBuffer.set(x, y, depth);
46          image.set(x, y, L_o);
47        }
48      }
49    }
50  }
```

To just test coverage, we don't need the magnitude of the barycentric weights. We only need to know that they are all positive. That is, that the current sample is on the positive side of every line bounding the triangle. To perform that test, we could use the distance from a point to a line instead of the full `bary2D` result. For this reason, this approach to rasterization is also referred to as testing the **edge aligns** at each sample. Since we need the barycentric weights for interpolation anyway, it makes sense to normalize the distances where they are computed. Our first instinct is to delay that normalization at least until after we know that the pixel is going to be shaded. However, even for performance, that is unnecessary—if we're going to optimize the inner loop, a much more significant optimization is available to us.

In general, barycentric weights vary linearly along any line through a triangle. The barycentric weight expressions are therefore linear in the loop variables `x` and `y`. You can see this by expanding `bary2D` in terms of the variables inside `lineDistance2D`, both from Listing 15.25. This becomes

$$\begin{aligned} \texttt{bary2D(A, B, C, Vector2(x, y))} &= \frac{(\texttt{n} \cdot (\texttt{x},\ \texttt{y}) + \texttt{d})/|\texttt{n}|}{(\texttt{n} \cdot \texttt{C} + \texttt{d})/|\texttt{n}|} \\ &= r \cdot \texttt{x} + s \cdot \texttt{y} + t, \end{aligned} \tag{15.9}$$

where the constants r, s, and t depend only on the triangle, and so are invariant across the triangle. We are particularly interested in properties invariant over horizontal and vertical lines, since those are our iteration directions.

For instance, `y` is invariant over the innermost loop along a scanline. Because the expressions inside the inner loop are constant in `y` (and all properties of `T`) and linear in `x`, we can compute them incrementally by accumulating derivatives with respect to `x`. That means that we can reduce all the computation inside the innermost loop and before the branch to three additions. Following the same argument for `y`, we can also reduce the computation that moves between rows to three additions. The only unavoidable operations are that for each sample that enters the branch for shading, we must perform three multiplications per scalar attribute; and we must perform a single division to compute $z = -1/w$, which is amortized over all attributes.

15.6.4.4 Precision for Incremental Interpolation

We need to think carefully about precision when incrementally accumulating derivatives rather than explicitly performing linear interpolation by the barycentric coordinates. To ensure that rasterization produces complementary pixel coverage for adjacent triangles with shared vertices ("watertight rasterization"), we must ensure that both triangles accumulate the same barycentric values at the shared edge as they iterate across their different bounding boxes. This means that we need an exact representation of the barycentric derivative. To accomplish this, we must

first round vertices to some imposed precision (say, one-quarter of a pixel width), and must then choose a representation and maximum screen size that provide exact storage.

The fundamental operation in the rasterizer is a 2D dot product to determine the side of the line on which a point lies. So we care about the precision of a multiplication and an addition. If our screen resolution is $w \times h$ and we want $k \times k$ subpixel positions for snapping or antialiasing, then we need $\lceil \log_2(k \cdot \max(w, h)) \rceil$ bits to store each scalar value. At 1920×1080 (i.e., effectively 2048×2048) with 4×4 subpixel precision, that's 14 bits. To store the product, we need twice as many bits. In our example, that's 28 bits. This is too large for the 23-bit mantissa portion of the IEEE 754 32-bit floating-point format, which means that we cannot implement the rasterizer using the single-precision `float` data type. We can use a 32-bit integer, representing a 24.4 fixed-point value. In fact, within that integer's space limitations we can increase screen resolution to 8192×8192 at 4×4 subpixel resolution. This is actually a fairly low-resolution subpixel grid, however. In contrast, DirectX 11 mandates eight bits of subpixel precision in each dimension. That is because under low subpixel precision, the aliasing pattern of a diagonal edge moving slowly across the screen appears to jump in discrete steps rather than evolve slowly with motion.

15.6.5 Rasterizing Shadows

Although we are now rasterizing primary visibility, our `shade` routine still determines the locations of shadows by casting rays. Shadowing from a local point source is equivalent to "visibility" from the perspective of that source. So we can apply rasterization to that visibility problem as well.

A **shadow map** [Wil78] is an auxiliary depth buffer rendered from a camera placed at the light's location. This contains the same distance information as obtained by casting rays from the light to selected points in the scene. The shadow map can be rendered in one pass over the scene geometry *before* the camera's view is rendered. Figure 15.14 shows a visualization of a shadow map, which is a common debugging aid.

When a shadowing computation arises during rendering from the camera's view, the renderer uses the shadow map to resolve it. For a rendered point to be unshadowed, it must be simultaneously visible to both the light and the camera. Recall that we are assuming a pinhole camera and a point light source, so the

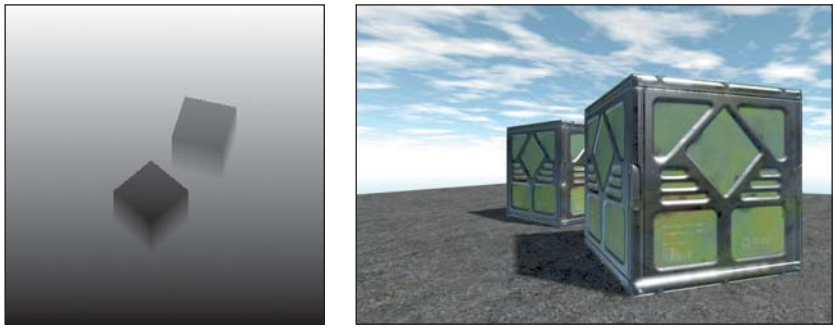


Figure 15.14: Left: A shadow map visualized with black = near the light and white = far from the light. Right: The camera's view of the scene with shadows.

paths from the point to each are defined by line segments of known length and orientation. Projecting the 3D point into the image space of the shadow map gives a 2D point. At that 2D point (or, more precisely, at a nearby one determined by rounding to the sampling grid for the shadow map) we previously stored the distance from the light to the first scene point, that is, the key information about the line segment. If that stored distance is equal to the distance from the 3D point to the 3D light source, then there must not have been any occluding surface and our point is lit. If the distance is less, then the point is in shadow because the light observes some other, shadow-casting, point first along the ray. This depth test must of course be conservative and approximate; we know there will be aliasing from both 2D discretization of the shadow map and its limited precision at each point.

Although we motivated shadow maps in the context of rasterization, they may be generated by or used to compute shadowing with both rasterization and ray casting renderers. There are often reasons to prefer to use the same visibility strategy throughout an application (e.g., the presence of efficient rasterization hardware), but there is no algorithmic constraint that we must do so.

When using a shadow map with triangle rasterization, we can amortize the cost of perspective projection into the shadow map over the triangle by performing most of the computational work at the vertices and then interpolating the results. The result must be interpolated in a perspective-correct fashion, of course. The key is that we want to be perspective-correct with respect to the matrix that maps points in world space to the shadow map, not to the viewport.

Recall the perspective-correct interpolation that we used for positions and texture coordinates (see previous sidebar, which essentially relied on linearly interpolating quantities of the form $\vec{u}/z$ and $w = -1/z$). If we multiply world-space vertices by the matrix that transforms them into 2D shadow map coordinates but do not perform the homogeneous division, then we have a value that varies linearly in the **homogeneous clip space** of the virtual camera at the light that produces the shadow map. In other words, we project each vertex into both the viewing camera's and the light camera's homogeneous clip space. We next perform the homogeneous division for the visible camera only and interpolate the four-component homogeneous vector representing the shadow map coordinate in a perspective-correct fashion in screen space. We next perform the perspective division for the shadow map coordinate at each pixel, paying only for the division and not the matrix product at each pixel. This allows us to transform to the light's projective view volume once per vertex and then interpolate those coordinates using the infrastructure already built for interpolating other elements. The reuse of a general interpolation mechanism and optimization of reducing transformations should naturally suggest that this approach is a good one for a hardware implementation of the graphics pipeline. Chapter 38 discusses how some of these ideas manifest in a particular graphics processor.

15.6.6 Beyond the Bounding Box

A triangle touching $O(n)$ pixels may have a bounding box containing $O(n^2)$ pixels. For triangles with all short edges, especially those with an area of about one pixel, rasterizing by iterating through all pixels in the bounding box is very efficient. Furthermore, the rasterization workload is very predictable for meshes of such triangles, since the number of tests to perform is immediately evident from the box bounds, and rectangular iteration is generally easier than triangular iteration.

For triangles with some large edges, iterating over the bounding box is a poor strategy because $n^2 \gg n$ for large n. In this case, other strategies can be more efficient. We now describe some of these briefly. Although we will not explore these strategies further, they make great projects for learning about hardware-aware algorithms and primary visibility computation.

15.6.6.1 Hierarchical Rasterization

Since the bounding-box rasterizer is efficient for small triangles and is easy to implement, a natural algorithmic approach is to recursively apply the bounding-box algorithm at increasingly fine resolution. This strategy is called **hierarchical rasterization** [Gre96].

Begin by dividing the entire image along a very coarse grid, such as into 16×16 macro-pixels that cover the entire screen. Apply a conservative variation of the bounding-box algorithm to these. Then subdivide the coarse grid and recursively apply the rasterization algorithm within all of the macro cells that overlapped the bounding box.

The algorithm could recur until the macro-pixels were actually a single pixel. However, at some point, we are able to perform a large number of tests either with Single Instruction Multiple Data (SIMD) operations or by using bitmasks packed into integers, so it may not always be a good idea to choose a single pixel as the base case. This is similar to the argument that you shouldn't quicksort all the way down to a length 1 array; for small problem sizes, the constant factors affect the performance more than the asymptotic bound.

For a given precision, one can precompute all the possible ways that a line passes through a tile of samples. These can be stored as bitmasks and indexed by the line's intercept points with the tile [FFR83, SW83]. For each line, using one bit to encode whether the sample is in the positive half-plane of the line allows an 8×8 pattern to fit in a single unsigned 64-bit integer. The bitwise AND of the patterns for the three line aligns defining the triangle gives the coverage mask for all 64 samples. One can use this trick to cull whole tiles efficiently, as well as avoiding per-sample visibility tests. (Kautz et al. [KLA04] extended this to a clever algorithm for rasterizing triangles onto hemispheres, which occurs frequently when sampling indirect illumination.) Furthermore, one can process multiple tiles simultaneously on a parallel processor. This is similar to the way that many GPUs rasterize today.

15.6.6.2 Chunking/Tiling Rasterization

A **chunking rasterizer,** a.k.a. a **tiling rasterizer,** subdivides the image into rectangular tiles, as if performing the first iteration of hierarchical rasterization. Instead of rasterizing a single triangle and performing recursive subdivision of the image, it takes *all* triangles in the scene and bins them according to which tiles they touch. A single triangle may appear in multiple bins.

The tiling rasterizer then uses some other method to rasterize within each tile. One good choice is to make the tiles 8×8 or some other size at which brute-force SIMD rasterization by a lookup table is feasible.

Working with small areas of the screen is a way to combine some of the best aspects of rasterization and ray casting. It maintains both triangle list and buffer memory coherence. It also allows triangle-level sorting so that visibility can be performed analytically instead of using a depth buffer. That allows both more

efficient visibility algorithms and the opportunity to handle translucent surfaces in more sophisticated ways.

15.6.6.3 Incremental Scanline Rasterization

For each row of pixels within the bounding box, there is some location that begins the span of pixels covered by the triangle and some location that ends the span. The bounding box contains the triangle vertically and triangles are convex, so there is exactly one span per row (although if the span is small, it may not actually cover the *center* of any pixels).

A scanline rasterizer divides the triangle into two triangles that meet at a horizontal line through the vertex with the median vertical ordinate of the original triangle (see Figure 15.15). One of these triangles may have zero area, since the original triangle may contain a horizontal edge.

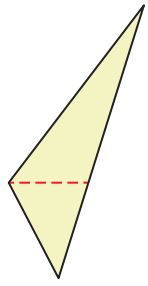


Figure 15.15: Dividing a triangle horizontally at its middle vertex.

The scanline rasterizer computes the rational slopes of the left and right edges of the top triangle. It then iterates down these in parallel (see Figure 15.16). Since these edges bound the beginning and end of the span across each scanline, no explicit per-pixel sample tests are needed: Every pixel center between the left and right edges at a given scanline is covered by the triangle. The rasterizer then iterates up the bottom triangle from the bottom vertex in the same fashion. Alternatively, it can iterate down the edges of the bottom triangle toward that vertex.

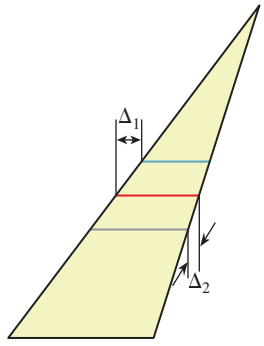


Figure 15.16: Each span's starting point shifts Δ_1 from that of the previous span, and its ending point shifts Δ_2.

The process of iterating along an edge is performed by a variant of either the **Digital Difference Analyzer (DDA)** or Bresenham line algorithm [Bre65], for which there are efficient floating-point and fixed-point implementations.

Pineda [Pin88] discusses several methods for altering the iteration pattern to maximize memory coherence. On current processor architectures this approach is generally eschewed in favor of tiled rasterization because it is hard to schedule for coherent parallel execution and frequently yields poor cache behavior.

15.6.6.4 Micropolygon Rasterization

Hierarchical rasterization recursively subdivided the *image* so that the triangle was always small relative to the number of macro-pixels in the image. An alternative is to maintain constant pixel size and instead subdivide the triangle. For example, each triangle can be divided into four similar triangles (see Figure 15.17). This is the rasterization strategy of the Reyes system [CCC87] used in one of the most popular film renderers, RenderMan. The subdivision process continues until the triangles cover about one pixel each. These triangles are called **micropolygons.** In addition to triangles, the algorithm is often applied to bilinear patches, that is, Bézier surfaces described by four control points (see Chapter 23).

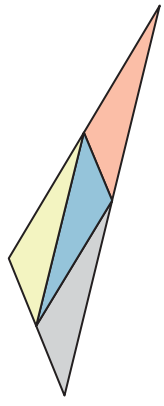


Figure 15.17: A triangle subdivided into four similar triangles.

Subdividing the geometry offers several advantages over subdividing the image. It allows additional geometric processing, such as displacement mapping, to be applied to the vertices after subdivision. This ensures that displacement is performed at (or slightly higher than) image resolution, effectively producing perfect level of detail. Shading can be performed at vertices of the micropolygons and interpolated to pixel centers. This means that the shading is "attached" to object-space locations instead of screen-space locations. This can cause shading features, such as highlights and edges, which move as the surface animates, to move more smoothly and with less aliasing than they do when we use screen-space shading. Finally, effects like motion blur and defocus can be applied by deforming the final shaded geometry before rasterization. This allows computation of shading at a rate

proportional to visible geometric complexity but independent of temporal and lens sampling.

15.7 Rendering with a Rasterization API

Rasterization has been encapsulated in APIs. We've seen that although the basic rasterization algorithm is very simple, the process of increasing its performance can rapidly introduce complexity. Very-high-performance rasterizers can be very complex. This complexity leads to a desire to separate out the parts of the rasterizer that we might wish to change between applications while encapsulating the parts that we would like to optimize once, abstract with an API, and then never change again. Of course, it is rare that one truly is willing to never alter an algorithm again, so this means that by building an API for part of the rasterizer we are trading performance and ease of use in some cases for flexibility in others. Hardware rasterizers are an extreme example of an optimized implementation, where flexibility is severely compromised in exchange for very high performance.

There have been several popular rasterization APIs. Today, OpenGL and DirectX are among the most popular hardware APIs for real-time applications. RenderMan is a popular software rasterization API for offline rendering. The space in between, of software rasterizers that run in real time on GPUs, is currently a popular research area with a few open source implementations available [LHLW10, LK11, Pan11].

In contrast to the relative standardization and popularity enjoyed among rasterizer APIs, several ray-casting systems have been built and several APIs have been proposed, although they have yet to reach the current level of standardization and acceptance of the rasterization APIs.

This section describes the OpenGL-DirectX abstraction in general terms. We prefer generalities because the exact entry points for these APIs change on a fairly regular basis. The details of the current versions can be found in their respective manuals. While important for implementation, those details obscure the important ideas.

15.7.1 The Graphics Pipeline

Consider the basic operations of any of our software rasterizer implementations:

1. (Vertex) Per-vertex transformation to screen space
2. (Rasterize) Per-triangle (clipping to the near plane and) iteration over pixels, with perspective-correct interpolation
3. (Pixel) Per-pixel shading
4. (Output Merge) Merging the output of shading with the current color and depth buffers (e.g., alpha blending)

These are the major stages of a rasterization API, and they form a sequence called the **graphics pipeline,** which was introduced in Chapter 1. Throughout the rest of this chapter, we refer to software that invokes API entry points as **host code** and software that is invoked as callbacks by the API as **device code.** In the context of a hardware-accelerated implementation, such as OpenGL on a GPU, this means that the C++ code running on the CPU is host code and the vertex and pixel shaders executing on the GPU are device code.

15.7.1.1 Rasterizing Stage

Most of the complexity that we would like such an API to abstract is in the rasterizing stage. Under current algorithms, rasterization is most efficient when implemented with only a few parameters, so this stage is usually implemented as a **fixed-function** unit. In hardware this may literally mean a specific circuit that can only compute rasterization. In software this may simply denote a module that accepts no parameterization.

15.7.1.2 Vertex and Pixel Stages

The per-vertex and per-pixel operations are ones for which a programmer using the API may need to perform a great deal of customization to produce the desired image. For example, an engineering application may require an orthographic projection of each vertex instead of a perspective one. We've already changed our per-pixel shading code three times, to support Lambertian, Blinn-Phong, and Blinn-Phong plus shadowing, so clearly customization of that stage is important. The performance impact of allowing nearly unlimited customization of vertex and pixel operations is relatively small compared to the benefits of that customization and the cost of rasterization and output merging. Most APIs enable customization of vertex and pixel stages by accepting callback functions that are executed for each vertex and each pixel. In this case, the stages are called **programmable** units.

A pipeline implementation with programmable units is sometimes called a **programmable pipeline.** Beware that in this context, the pipeline *order* is in fact fixed, and only the *units* within it are programmable. Truly programmable pipelines in which the order of stages can be altered have been proposed [SFB+09] but are not currently in common use.

For historical reasons, the callback functions are often called **shaders** or **programs.** Thus, a **pixel shader** or "pixel program" is a callback function that will be executed at the per-pixel stage. For triangle rasterization, the pixel stage is often referred to as the **fragment** stage. A fragment is the portion of a triangle that overlaps the bounds of a pixel. It is a matter of viewpoint whether one is computing the shade of the fragment and sampling that shade at the pixel, or directly computing the shade at the pixel. The distinction only becomes important when computing visibility independently from shading. **Multi-sample anti-aliasing** (**MSAA**) is an example of this. Under that rasterization strategy, many visibility samples (with corresponding depth buffer and radiance samples) are computed within each pixel, but a single shade is applied to all the samples that pass the depth and visibility test. In this case, one truly is shading a fragment and not a pixel.

15.7.1.3 Output Merging Stage

The output merging stage is one that we might like to customize as consumers of the API. For example, one might imagine simulating translucent surfaces by blending the current and previous radiance values in the frame buffer. However, the output merger is also a stage that requires synchronization between potentially parallel instances of the pixel shading units, since it writes to a shared frame buffer. As a result, most APIs provide only limited customization at the output merge stage. That allows lockless access to the underlying data structures, since the implementation may explicitly schedule pixel shading to avoid contention at the frame buffer. The limited customization options typically allow the programmer to choose the operator for the depth comparison. They also typically allow a choice of compositing operator for color limited to linear blending, minimum, and maximum operations on the color values.

There are of course more operations for which one might wish to provide an abstracted interface. These include per-object and per-mesh transformations, tessellation of curved patches into triangles, and per-triangle operations like silhouette detection or surface extrusion. Various APIs offer abstractions of these within a programming model similar to vertex and pixel shaders.

Chapter 38 discusses how GPUs are designed to execute this pipeline efficiently. Also refer to your API manual for a discussion of the additional stages (e.g., tessellate, geometry) that may be available.

15.7.2 Interface

The interface to a software rasterization API can be very simple. Because a software rasterizer uses the same memory space and execution model as the host program, one can pass the scene as a pointer and the callbacks as function pointers or classes with virtual methods. Rather than individual triangles, it is convenient to pass whole meshes to a software rasterizer to decrease the per-triangle overhead.

For a hardware rasterization API, the host machine (i.e., CPU) and graphics device (i.e., GPU) may have separate memory spaces and execution models. In this case, shared memory and function pointers no longer suffice. Hardware rasterization APIs therefore must impose an explicit memory boundary and narrow entry points for negotiating it. (This is also true of the fallback and reference software implementations of those APIs, such as Mesa and DXRefRast.) Such an API requires the following entry points, which are detailed in subsequent subsections.

1. Allocate device memory.
2. Copy data between host and device memory.
3. Free device memory.
4. Load (and compile) a shading program from source.
5. Configure the output merger and other fixed-function state.
6. Bind a shading program and set its arguments.
7. Launch a **draw call,** a set of device threads to render a triangle list.

15.7.2.1 Memory Principles

The memory management routines are conceptually straightforward. They correspond to `malloc`, `memcpy`, and `free`, and they are typically applied to large arrays, such as an array of vertex data. They are complicated by the details necessary to achieve high performance for the case where data must be transferred per rendered frame, rather than once per scene. This occurs when streaming geometry for a scene that is too large for the device memory; for example, in a world large enough that the viewer can only ever observe a small fraction at a time. It also occurs when a data stream from another device, such as a camera, is an input to the rendering algorithm. Furthermore, hybrid software-hardware rendering and physics algorithms perform some processing on each of the host and device and must communicate each frame.

One complicating factor for memory transfer is that it is often desirable to adjust the data layout and precision of arrays during the transfer. The data structure for 2D buffers such as images and depth buffers on the host often resembles the "linear," row-major ordering that we have used in this chapter. On a graphics processor, 2D buffers are often wrapped along Hilbert or Z-shaped (Morton)

curves, or at least grouped into small blocks that are themselves row-major (i.e., "block-linear"), to avoid the cache penalty of vertical iteration. The origin of a buffer may differ, and often additional padding is required to ensure that rows have specific memory alignments for wide vector operations and reduced pointer size.

Another complicating factor for memory transfer is that one would often like to overlap computation with memory operations to avoid stalling either the host or device. Asynchronous transfers are typically accomplished by semantically mapping device memory into the host address space. Regular host memory operations can then be performed as if both shared a memory space. In this case the programmer must manually synchronize both host and device programs to ensure that data is never read by one while being written by the other. Mapped memory is typically uncached and often has alignment considerations, so the programmer must furthermore be careful to control access patterns.

Note that memory transfers are intended for large data. For small values, such as scalars, 4×4 matrices, and even short arrays, it would be burdensome to explicitly allocate, copy, and free the values. For a shading program with twenty or so arguments, that would incur both runtime and software management overhead. So small values are often passed through a different API associated with shaders.

15.7.2.2 Memory Practice

Listing 15.30 shows part of an implementation of a triangle mesh class. Making rendering calls to transfer individual triangles from the host to the graphics device would be inefficient. So, the API forces us to load a large array of the geometry to the device once when the scene is created, and to encode that geometry as efficiently as possible.

Few programmers write directly to hardware graphics APIs. Those APIs reflect the fact that they are designed by committees and negotiated among vendors. They provide the necessary functionality but do so through awkward interfaces that obscure the underlying function of the calling code. Usage is error-prone because the code operates directly on pointers and uses manually managed memory.

For example, in OpenGL, the code to allocate a device array and bind it to a shader input looks something like Listing 15.29. Most programmers abstract these direct host calls into a vendor-independent, easier-to-use interface.

Listing 15.29: Host code for transferring an array of vertices to the device and binding it to a shader input.

```
1  // Allocate memory:
2  GLuint vbo;
3  glGenBuffers(1, &vbo);
4  glBindBuffer(GL_ARRAY_BUFFER, vbo);
5  glBufferData(GL_ARRAY_BUFFER, hostVertex.size() * 2 * sizeof(Vector3), NULL,GL_STATIC_DRAW);
6  GLvoid* deviceVertex = 0;
7  GLvoid* deviceNormal = hostVertex.size() * sizeof(Vector3);
8
9  // Copy memory:
10 glBufferSubData(GL_ARRAY_BUFFER, deviceVertex, hostVertex.size() *
       sizeof(Point3), &hostVertex[0]);
11
12 // Bind the array to a shader input:
13 int vertexIndex = glGetAttribLocation(shader, "vertex");
14 glEnableVertexAttribArray(vertexIndex);
15 glVertexAttribPointer(vertexIndex, 3, GL_FLOAT, GL_FALSE, 0, deviceVertex);
```

Most programmers wrap the underlying hardware API with their own layer that is easier to use and provides type safety and memory management. This also has the advantage of abstracting the renderer from the specific hardware API. Most console, OS, and mobile device vendors intentionally use equivalent but incompatible hardware rendering APIs. Abstracting the specific hardware API into a generic one makes it easier for a single code base to support multiple platforms, albeit at the cost of one additional level of function invocation.

For Listing 15.30, we wrote to one such platform abstraction instead of directly to a hardware API. In this code, the `VertexBuffer` class is a managed memory array in device RAM and `AttributeArray` and `IndexArray` are subsets of a `VertexBuffer`. The "vertex" in the name means that these classes store per-vertex data. It does not mean that they store only vertex positions—for example, the `m_normal` array is stored in an `AttributeArray`. This naming convention is a bit confusing, but it is inherited from OpenGL and DirectX. You can either translate this code to the hardware API of your choice, implement the `VertexBuffer` and `AttributeArray` classes yourself, or use a higher-level API such as G3D that provides these abstractions.

Listing 15.30: Host code for an indexed triangle mesh (equivalent to a set of `Triangle` *instances that share a* `BSDF`*).*

```
class Mesh {
private:
    AttributeArray    m_vertex;
    AttributeArray    m_normal;
    IndexStream       m_index;

    shared_ptr<BSDF>  m_bsdf;

public:

    Mesh() {}

    Mesh(const std::vector<Point3>& vertex,
        const std::vector<Vector3>& normal,
        const std::vector<int>& index, const shared_ptr<BSDF>& bsdf) : m_bsdf(bsdf) {

        shared_ptr<VertexBuffer> dataBuffer =
          VertexBuffer::create((vertex.size() + normal.size()) *
            sizeof(Vector3) + sizeof(int) * index.size());
        m_vertex = AttributeArray(&vertex[0], vertex.size(), dataBuffer);
        m_normal = AttributeArray(&normal[0], normal.size(), dataBuffer);

        m_index = IndexStream(&index[0], index.size(), dataBuffer);
    }

    ...
};

/** The rendering API pushes us towards a mesh representation
    because it would be inefficient to make per-triangle calls. */
class MeshScene {
public:
    std::vector<Light>    lightArray;
    std::vector<Mesh>     meshArray;
};
```

Listing 15.31 shows how this code is used to model the triangle-and-ground-plane scene. In it, the process of uploading the geometry to the graphics device is entirely abstracted within the `Mesh` class.

Listing 15.31: Host code to create indexed triangle meshes for the triangle-plus-ground scene.

```
void makeTrianglePlusGroundScene(MeshScene& s) {
    std::vector<Vector3> vertex, normal;
    std::vector<int> index;

    // Green triangle geometry
    vertex.push_back(Point3(0,1,-2)); vertex.push_back(Point3(-1.9f,-1,-2));
        vertex.push_back(Point3(1.6f,-0.5f,-2));
    normal.push_back(Vector3(0,0.6f,1).direction()); normal.
        push_back(Vector3(-0.4f,-0.4f, 1.0f).direction()); normal.
        push_back(Vector3(0.4f,-0.4f, 1.0f).direction());
    index.push_back(0); index.push_back(1); index.push_back(2);
    index.push_back(0); index.push_back(2); index.push_back(1);
    shared_ptr<BSDF> greenBSDF(new PhongBSDF(Color3::green() * 0.8f,
                                             Color3::white() * 0.2f, 100));

    s.meshArray.push_back(Mesh(vertex, normal, index, greenBSDF));
    vertex.clear(); normal.clear(); index.clear();

    ////////////////////////////////////////////////////////
    // Ground plane geometry
    const float groundY = -1.0f;
    vertex.push_back(Point3(-10, groundY, -10)); vertex.push_back(Point3(-10,
        groundY, -0.01f));
    vertex.push_back(Point3(10, groundY, -0.01f)); vertex.push_back(Point3(10,
        groundY, -10));

    normal.push_back(Vector3::unitY()); normal.push_back(Vector3::unitY());
    normal.push_back(Vector3::unitY()); normal.push_back(Vector3::unitY());

    index.push_back(0); index.push_back(1); index.push_back(2);
    index.push_back(0); index.push_back(2); index.push_back(3);

    const Color3 groundColor = Color3::white() * 0.8f;
    s.meshArray.push_back(Mesh(vertex, normal, index, groundColor));

    ////////////////////////////////////////////////////////
    // Light source
    s.lightArray.resize(1);
    s.lightArray[0].position = Vector3(1, 3, 1);
    s.lightArray[0].power = Color3::white() * 31.0f;
}
```

15.7.2.3 Creating Shaders

The vertex shader must transform the input vertex in global coordinates to a homogeneous point on the image plane. Listing 15.32 implements this transformation. We chose to use the OpenGL Shading Language (GLSL). GLSL is representative of other contemporary shading languages like HLSL, Cg, and RenderMan. All of these are similar to C++. However, there are some minor syntactic differences between GLSL and C++ that we call out here to aid your reading of this example. In GLSL,

- Arguments that are constant over all triangles are passed as global ("uniform") variables.
- Points, vectors, and colors are all stored in `vec3` type.
- `const` has different semantics (compile-time constant).
- `in`, `out`, and `inout` are used in place of C++ reference syntax.
- `length`, `dot`, etc. are functions instead of methods on vector classes.

Listing 15.32: Vertex shader for projecting vertices. The output is in homogeneous space before the division operation. This corresponds to the `perspectiveProject` *function from Listing 15.24.*

```
#version 130

// Triangle vertices
in vec3 vertex;
in vec3 normal;

// Camera and screen parameters
uniform float fieldOfViewX;
uniform float zNear;
uniform float zFar;
uniform float width;
uniform float height;

// Position to be interpolated
out vec3 Pinterp;

// Normal to be interpolated
out vec3 ninterp;

vec4 perspectiveProject(in vec3 P) {
    // Compute the side of a square at z = -1 based on our
    // horizontal left-edge-to-right-edge field of view .
    float s = -2.0f * tan(fieldOfViewX * 0.5f);
    float aspect = height / width;

    // Project onto z = -1
    vec4 Q;
    Q.x = 2.0 * -Q.x / s;
    Q.y = 2.0 * -Q.y / (s * aspect);
    Q.z = 1.0;
    Q.w = -P.z;

    return Q;
}

void main() {
    Pinterp = vertex;
    ninterp = normal;

    gl_Position = perspectiveProject(Pinterp);
}
```

None of these affect the expressiveness or performance of the basic language. The specifics of shading-language syntax change frequently as new versions are released, so don't focus too much on the details. The point of this example is how the overall form of our original program is preserved but adjusted to the conventions of the hardware API.

Under the OpenGL API, the outputs of a vertex shader are a set of attributes and a vertex of the form $(x, y, a, -z)$. That is, a homogeneous point for which the perspective division has not yet been performed. The value $a/-z$ will be used for the depth test. We choose $a = 1$ so that the depth test is performed on $-1/z$, which is a positive value for the negative z locations that will be visible to the camera. We previously saw that any function that provides a consistent depth ordering can be used for the depth test. We mentioned that distance along the eye ray, $-z$, and $-1/z$ are common choices. Typically one scales the a value such that $-a/z$ is in the range $[0, 1]$ or $[-1, 1]$, but for simplicity we'll omit that here. See Chapter 13 for the derivation of that transformation.

Note that we did not scale the output vertex to the dimensions of the image, negate the y-axis, or translate the origin to the upper left in screen space, as we did for the software renderer. That is because by convention, OpenGL considers the upper-left corner of the screen to be at $(-1, 1)$ and the lower-right corner at $(1, -1)$.

We choose the 3D position of the vertex and its normal as our attributes. The hardware rasterizer will automatically interpolate these across the surface of the triangle in a perspective-correct manner. We need to treat the vertex as an attribute because OpenGL does not expose the 3D coordinates of the point being shaded.

Listings 15.33 and 15.34 give the pixel shader code for the `shade` routine, which corresponds to the shade function from Listing 15.17, and helper functions that correspond to the `visible` and `BSDF::evaluateFiniteScatteringDensity` routines from the ray tracer and software rasterizer. The output of the shader is in homogeneous space before the division operation. This corresponds to the `perspectiveProject` function from Listing 15.24. The interpolated attributes enter the shader as global variables `Pinterp` and `ninterp`. We then perform shading in exactly the same manner as for the software renderers.

Listing 15.33: Pixel shader for computing the radiance scattered toward the camera from one triangle illuminated by one light.

```
#version 130
// BSDF
uniform vec3    lambertian;
uniform vec3    glossy;
uniform float   glossySharpness;

// Light
uniform vec3    lightPosition;
uniform vec3    lightPower;

// Pre-rendered depth map from the light's position
uniform sampler2DShadow shadowMap;

// Point being shaded. OpenGL has automatically performed
// homogeneous division and perspective-correct interpolation for us.
in vec3            Pinterp;
in vec3            ninterp;

// Value we are computing
out vec3           radiance;

// Normalize the interpolated normal; OpenGL does not automatically
// renormalize for us.
vec3 n = normalize(ninterp);

```

```
vec3 shade(const in vec3 P, const in vec3 n) {
vec3 radiance              = vec3(0.0);

  // Assume only one light
  vec3 offset           = lightPosition - P;
  float distanceToLight = length(offset);
  vec3 w_i              = offset / distanceToLight;
  vec3 w_o              = -normalize(P);

  if (visible(P, w_i, distanceToLight, shadowMap)) {
      vec3 L_i = lightPower / (4 * PI * distanceToLight * distanceToLight);

      // Scatter the light.
      radiance +=
          L_i *
          evaluateFiniteScatteringDensity(w_i, w_o) *
          max(0.0, dot(w_i, n));
  }

  return radiance;
}

void main() {
     vec3 P = Pinterp;


     radiance = shade(P, n);
}
```

Listing 15.34: Helper functions for the pixel shader.

```
#define PI 3.1415927

bool visible(const in vec3 P, const in vec3 w_i, const in float distanceToLight,
    sampler2DShadow shadowMap) {
    return true;
}

/** Returns f(wi, wo). Same as BSDF::evaluateFiniteScatteringDensity
    from the ray tracer. */
vec3 evaluateFiniteScatteringDensity(const in vec3 w_i, const in vec3 w_o) {
     vec3 w_h = normalize(w_i + w_o);

     return (k_L +
             k_G * ((s + 8.0) * pow(max(0.0, dot(w_h, n)), s) / 8.0)) / PI;
}
```

However, there is one exception. The software renderers iterated over all the lights in the scene for each point to be shaded. The pixel shader is hardcoded to accept a single light source. That is because processing a variable number of arguments is challenging at the hardware level. For performance, the inputs to shaders are typically passed through registers, not heap memory. Register allocation is generally a major factor in optimization. Therefore, most shading compilers require the number of registers consumed to be known at compile time, which precludes passing variable length arrays. Programmers have developed three **forward-rendering** design patterns for working within this limitation. These use a single framebuffer and thus limit the total space required by the algorithm. A fourth and currently popular **deferred-rendering** method requires additional space.

1. Multipass Rendering: Make one **pass** per light over all geometry, summing the individual results. This works because light combines by superposition. However, one has to be careful to resolve visibility correctly on the first pass and then never alter the depth buffer. This is the simplest and most elegant solution. It is also the slowest because the overhead of launching a pixel shader may be significant, so launching it multiple times to shade the same point is inefficient.

2. Übershader: Bound the total number of lights, write a shader for that maximum number, and set the unused lights to have zero power. This is one of the most common solutions. If the overhead of launching the pixel shader is high and there is significant work involved in reading the BSDF parameters, the added cost of including a few unused lights may be low. This is a fairly straightforward modification to the base shader and is a good compromise between performance and code clarity.

3. Code Generation: Generate a set of shading programs, one for each number of lights. These are typically produced by writing another program that automatically generates the shader code. Load *all* of these shaders at runtime and bind whichever one matches the number of lights affecting a particular object. This achieves high performance if the shader only needs to be swapped a few times per frame, and is potentially the fastest method. However, it requires significant infrastructure for managing both the source code and the compiled versions of all the shaders, and may actually be slower than the conservative solution if changing shaders is an expensive operation.

If there are different BSDF terms for different surfaces, then we have to deal with all the permutations of the number of lights and the BSDF variations. We again choose between the above three options. This combinatorial explosion is one of the primary drawbacks of current shading languages, and it arises directly from the requirement that the shading compiler produce efficient code. It is not hard to design more flexible languages and to write compilers for them. But our motivation for moving to a hardware API was largely to achieve increased performance, so we are unlikely to accept a more general shading language if it significantly degrades performance.

4. Deferred Lighting: A deferred approach that addresses these problems but requires more memory is to separate the computation of *which* point will color each pixel from illumination computation. An initial rendering pass renders many parallel buffers that encode the shading coefficients, surface normal, and location of each point (often, assuming an übershader). Subsequent passes then iterate over the screen-space area conservatively affected by each light, computing and summing illumination. Two common structures for those lighting passes are multiple lights applied to large screen-space tiles and ellipsoids for individual lights that cover the volume within which their contribution is non-negligible.

For the single-light case, moving from our own software rasterizer to a hardware API did not change our `perspectiveProject` and `shade` functions substantially.

However, our shade function was not particularly powerful. Although we did not choose to do so, in our software rasterizer, we could have executed arbitrary code inside the shade function. For example, we could have written to locations other than the current pixel in the frame buffer, or cast rays for shadows or reflections. Such operations are typically disallowed in a hardware API. That is because they interfere with the implementation's ability to efficiently schedule parallel instances of the shading programs in the absence of explicit (inefficient) memory locks.

This leaves us with two choices when designing an algorithm with more significant processing, especially at the pixel level. The first choice is to build a hybrid renderer that performs some of the processing on a more general processor, such as the host, or perhaps on a general computation API (e.g., CUDA, Direct Compute, OpenCL, OpenGL Compute). Hybrid renderers typically incur the cost of additional memory operations and the associated synchronization complexity.

The second choice is to frame the algorithm purely in terms of rasterization operations, and make multiple rasterization passes. For example, we can't conveniently cast shadow rays in most hardware rendering APIs today. But we can sample from a previously rendered shadow map.

Similar methods exist for implementing reflection, refraction, and indirect illumination purely in terms of rasterization. These avoid much of the performance overhead of hybrid rendering and leverage the high performance of hardware rasterization. However, they may not be the most natural way of expressing an algorithm, and that may lead to a net inefficiency and certainly to additional software complexity. Recall that changing the order of iteration from ray casting to rasterization increased the space demands of rendering by requiring a depth buffer to store intermediate results. In general, converting an arbitrary algorithm to a rasterization-based one often has this effect. The space demands might grow larger than is practical in cases where those intermediate results are themselves large.

Shading languages are almost always compiled into executable code at runtime, inside the API. That is because even within products from one vendor the underlying micro-architecture may vary significantly. This creates a tension within the compiler between optimizing the target code and producing the executable quickly. Most implementations err on the side of optimization, since shaders are often loaded once per scene. Beware that if you synthesize or stream shaders throughout the rendering process there may be substantial overhead.

Some languages (e.g., HLSL and CUDA) offer an initial compilation step to an intermediate representation. This eliminates the runtime cost of parsing and some trivial compilation operations while maintaining flexibility to optimize for a specific device. It also allows software developers to distribute their graphics applications without revealing the shading programs to the end-user in a human-readable form on the file system. For closed systems with fixed specifications, such as game consoles, it is possible to compile shading programs down to true machine code. That is because on those systems the exact runtime device is known at host-program compile time. However, doing so would reveal some details of the proprietary micro-architecture, so even in this case vendors do not always choose to have their APIs perform a complete compilation step.

15.7.2.4 Executing Draw Calls

To invoke the shaders we issue `draw` calls. These occur on the host side. One typically clears the framebuffer, and then, for each mesh, performs the following operations.

1. Set fixed function state.
2. Bind a shader.
3. Set shader arguments.
4. Issue the draw call.

These are followed by a call to send the framebuffer to the display, which is often called a **buffer swap.** An abstracted implementation of this process might look like Listing 15.35. This is called from a main rendering loop, such as Listing 15.36.

Listing 15.35: Host code to set fixed-function state and shader arguments, and to launch a draw call under an abstracted hardware API.

```
void loopBody(RenderDevice* gpu) {
    gpu->setColorClearValue(Color3::cyan() * 0.1f);
    gpu->clear();

    const Light& light = scene.lightArray[0];

    for (unsigned int m = 0; m < scene.meshArray.size(); ++m) {
        Args args;
        const Mesh& mesh = scene.meshArray[m];
        const shared_ptr<BSDF>& bsdf = mesh.bsdf();

        args.setUniform("fieldOfViewX",      camera.fieldOfViewX);
        args.setUniform("zNear",             camera.zNear);
        args.setUniform("zFar",              camera.zFar);

        args.setUniform("lambertian",        bsdf->lambertian);
        args.setUniform("glossy",            bsdf->glossy);
        args.setUniform("glossySharpness",   bsdf->glossySharpness);

        args.setUniform("lightPosition",     light.position);
        args.setUniform("lightPower",        light.power);

        args.setUniform("shadowMap",         shadowMap);

        args.setUniform("width",             gpu->width());
        args.setUniform("height",            gpu->height());

        gpu->setShader(shader);

        mesh.sendGeometry(gpu, args);
    }
    gpu->swapBuffers();
}
```

Listing 15.36: Host code to set up the main hardware rendering loop.

```
OSWindow::Settings osWindowSettings;
RenderDevice* gpu = new RenderDevice();
gpu->init(osWindowSettings);

// Load the vertex and pixel programs
shader = Shader::fromFiles("project.vrt", "shade.pix");

shadowMap = Texture::createEmpty("Shadow map", 1024, 1024,
    ImageFormat::DEPTH24(), Texture::DIM_2D_NPOT, Texture::Settings::shadow());
makeTrianglePlusGroundScene(scene);

```

```
// The depth test will run directly on the interpolated value in
// Q.z/Q.w, which is going to be smallest at the far plane
gpu->setDepthTest(RenderDevice::DEPTH_GREATER);
gpu->setDepthClearValue(0.0);

while (! done) {
    loopBody(gpu);
    processUserInput();
}

...
```

15.8 Performance and Optimization

We'll now consider several examples of optimization in hardware-based rendering. This is by no means an exhaustive list, but rather a set of model techniques from which you can draw ideas to generate your own optimizations when you need them.

15.8.1 Abstraction Considerations

Many performance optimizations will come at the price of significantly complicating the implementation. Weigh the performance advantage of an optimization against the additional cost of debugging and code maintenance. High-level algorithmic optimizations may require significant thought and restructuring of code, but they tend to yield the best tradeoff of performance for code complexity. For example, simply dividing the screen in half and asynchronously rendering each side on a separate processor nearly doubles performance at the cost of perhaps 50 additional lines of code that do not interact with the inner loop of the renderer.

In contrast, consider some low-level optimizations that we intentionally passed over. These include reducing common subexpressions (e.g., mapping all of those repeated divisions to multiplications by an inverse that is computed once) and lifting constants outside loops. Performing those destroys the clarity of the algorithm, but will probably gain only a 50% throughput improvement.

This is not to say that low-level optimizations are not worthwhile. But they are primarily worthwhile when you have completed your high-level optimizations; at that point you are more willing to complicate your code and its maintenance because you are done adding features.

15.8.2 Architectural Considerations

The primary difference between the simple rasterizer and ray caster described in this chapter is that the "for each pixel" and "for each triangle" loops have the opposite nesting. This is a trivial change and the body of the inner loop is largely similar in each case. But the trivial change has profound implications for memory access patterns and how we can algorithmically optimize each.

Scene triangles are typically stored in the heap. They may be in a flat 1D array, or arranged in a more sophisticated data structure. If they are in a simple data structure such as an array, then we can ensure reasonable memory coherence by iterating through them in the same order that they appear in memory. That produces efficient cache behavior. However, that iteration also requires substantial

bandwidth because the entire scene will be processed for each pixel. If we use a more sophisticated data structure, then we likely will reduce bandwidth but also reduce memory coherence. Furthermore, adjacent pixels likely sample the same triangle, but by the time we have iterated through to testing that triangle again it is likely to have been flushed from the cache. A popular low-level optimization for a ray tracer is to trace a bundle of rays called a **ray packet** through adjacent pixels. These rays likely traverse the scene data structure in a similar way, which increases memory coherence. On a SIMD processor a single thread can trace an entire packet simultaneously. However, packet tracing suffers from computational coherence problems. Sometimes different rays in the same packet progress to different parts of the scene data structure or branch different ways in the ray intersection test. In these cases, processing multiple rays simultaneously on a thread gives no advantage because memory coherence is lost or both sides of the branch must be taken. As a result, fast ray tracers are often designed to trace packets through very sophisticated data structures. They are typically limited not by computation but by memory performance problems arising from resultant cache inefficiency.

Because frame buffer storage per pixel is often much smaller than scene structure per triangle, the rasterizer has an inherent memory performance advantage over the ray tracer. A rasterizer reads each triangle into memory and then processes it to completion, iterating over many pixels. Those pixels must be adjacent to each other in space. For a row-major image, if we iterate along rows, then the pixels covered by the triangle are also adjacent in memory and we will have excellent coherence and fairly low memory bandwidth in the inner loop. Furthermore, we can process multiple adjacent pixels, either horizontally or vertically, simultaneously on a SIMD architecture. These will be highly memory and branch coherent because we're stepping along a single triangle. There are many variations on ray casting and rasterization that improve their asymptotic behavior. However, these algorithms have historically been applied to only millions of triangles and pixels. At those sizes, constant factors like coherence still drive the performance of the algorithms, and rasterization's superior coherence properties have made it preferred for high-performance rendering. The cost of this coherence is that after even the few optimizations needed to get real-time performance from a rasterizer, the code becomes so littered with bit-manipulation tricks and highly derived terms that the elegance of a simple ray cast seems very attractive from a software engineering perspective. This difference is only magnified when we make the rendering algorithm more sophisticated. The conventional wisdom is that ray-tracing algorithms are elegant and easy to extend but are hard to optimize, and rasterization algorithms are very efficient but are awkward and hard to augment with new features. Of course, one can always make a ray tracer fast and ugly (which packet tracing succeeds at admirably) and a rasterizer extensible but slow (e.g., Pixar's RenderMan, which was used extensively in film rendering over the past two decades).

15.8.3 Early-Depth-Test Example

One simple optimization that can significantly improve performance, yet only minimally affects clarity, is an early depth test. Both the rasterizer and the ray-tracer structures sometimes shaded a point, only to later find that some other point was closer to the surface. As an optimization, we might first find the closest point before doing any shading, then go back and shade the point that was closest. In ray

tracing, each pixel is processed to completion before moving to the next, so this involves running the entire visibility loop for one pixel, maintaining the shading inputs for the closest-known intersection at each iteration, and then shading after that loop terminates. In rasterization, pixels are processed many times, so we have to make a complete first pass to determine visibility and then a second pass to do shading. This is called an **early-depth pass** [HW96] if it primes `depthBuffer` so that only the surface that shades will pass the inner test. The process is called **deferred shading** if it also accumulates the shading parameters so that they do not need to be recomputed. This style of rendering was first introduced by Whitted and Weimer [WW82] to compute shading independent from visibility at a time when primary visibility computation was considered expensive. Within a decade it was considered a method to accelerate complex rendering toward real-time rendering (and the "deferred" term was coined) [MEP92], and today its use is widespread as a further optimization on hardware platforms that already achieve real time for complex scenes.

For a scene that has high **depth complexity** (i.e., in which many triangles project to the same point in the image) and an expensive shading routine, the performance benefit of an early depth test is significant. The cost of rendering a pixel without an early depth test is $\mathrm{O}(tv+ts)$, where t is the number of triangles, v is the time for a visibility test, and s is the time for shading. This is an upper bound. When we are lucky and always encounter the closest triangle first, the performance matches the lower bound of $\Omega(tv+s)$ since we only shade once. The early-depth optimization ensures that we are always in this lower-bound case. We have seen how rasterization can drive the cost of v very low—it can be reduced to a few additions per pixel—at which point the challenge becomes reducing the number of triangles tested at each pixel. Unfortunately, that is not as simple. Strategies exist for obtaining expected $\mathrm{O}(v\log t+s)$ rendering times for scenes with certain properties, but they significantly increase code complexity.

15.8.4 When Early Optimization Is Good

The domain of graphics raises two time-based exceptions to the general rule of thumb to avoid premature optimization. The more significant of these exceptions is that when low-level optimizations can accelerate a rendering algorithm just enough to make it run at interactive rates, it might be worth making those optimizations early in the development process. It is much easier to debug an interactive rendering system than an offline one. Interaction allows you to quickly experiment with new viewpoints and scene variations, effectively giving you a true 3D perception of your data instead of a 2D slice. If that lets you debug faster, then the optimization has increased your ability to work with the code despite the added complexity. The other exception applies when the render time is just at the threshold of your patience. Most programmers are willing to wait for 30 seconds for an image to render, but they will likely leave the computer or switch tasks if the render time is, say, more than two minutes. Every time you switch tasks or leave the computer you're amplifying the time cost of debugging, because on your return you have to recall what you were doing before you left and get back into the development flow. If you can reduce the render time to something you are willing to wait for, then you have cut your debugging time and made the process sufficiently more pleasant that your productivity will again rise despite increased code complexity. We enshrine these ideas in a principle:

✓ **THE EARLY OPTIMIZATION PRINCIPLE:** It's worth optimizing early if it makes the difference between an interactive program and one that takes several minutes to execute. Shortening the debugging cycle and supporting interactive testing are worth the extra effort.

15.8.5 Improving the Asymptotic Bound

To scale to truly large scenes, no linear-time rendering algorithm suffices. We must somehow eliminate whole parts of the scene without actually touching their data even once. Data structures for this are a classic area of computer graphics that continues to be a hot research topic. The basic idea behind most of these is the same as behind using tree and bucket data structures for search and sort problems. Visibility testing is primarily a search operation, where we are searching for the closest ray intersection with the scene. If we precompute a treelike data structure that orders the scene primitives in some way that allows conservatively culling a constant fraction of the primitives at each layer, we will approach $O(\log n)$-time visibility testing for the entire scene, instead of $O(n)$ in the number of primitives. When the cost of traversing tree nodes is sufficiently low, this strategy scales well for arbitrarily constructed scenes and allows an exponential increase in the number of primitives we can render in a fixed time. For scenes with specific kinds of structure we may be able to do even better. For example, say that we could find an indexing scheme or hash function that can divide our scene into $O(n)$ buckets that allow conservative culling with $O(1)$ primitives per bucket. This would approach $O(d)$-time visibility testing in the distance d to the first intersection. When that distance is small (e.g., in twisty corridors), the runtime of this scheme for static scenes becomes independent of the number of primitives and we can theoretically render arbitrarily large scenes. See Chapter 37 for a detailed discussion of algorithms based on these ideas.

15.9 Discussion

Our goal in this chapter was not to say, "You can build either a ray tracer or a rasterizer," but rather that rendering involves sampling of light sources, objects, and rays, and that there are broad algorithmic strategies you can use for accumulating samples and interpolating among them. This provides a stage for all future rendering, where we try to select samples efficiently and with good statistical characteristics.

For sampling the scene along eye rays through pixel centers, we saw that three tests—explicit 3D ray-triangle tests, 2D ray-triangle through incremental barycentric tests, and 2D ray-triangle through incremental edge equation tests—were mathematically equivalent. We also discussed how to implement them so that the mathematical equivalence was preserved even in the context of bounded-precision arithmetic. In each case we computed some value directly related to the barycentric weights and then tested whether the weights corresponded to a point on the interior of the triangle. It is essential that these are mathematically equivalent tests. Were they not, we would not expect all methods to produce the same image! Algorithmically, these approaches led to very different strategies. That is

because they allowed amortization in different ways and provoked different memory access patterns.

Sampling is the core of physically based rendering. The kinds of design choices you faced in this chapter echo throughout all aspects of rendering. In fact, they are significant for all high-performance computing, spreading into fields as diverse as biology, finance, and weather simulation. That is because many interesting problems do not admit analytic solutions and must be solved by taking discrete samples. One frequently wants to take many of those samples in parallel to reduce computation latency. So considerations about how to sample over a complex domain, which in our case was the set product of triangles and eye rays, are fundamental to science well beyond image synthesis.

The ray tracer in this chapter is a stripped-down, no-frills ray tracer. But it still works pretty well. Ten years ago you would have had to wait an hour for the teapot to render. It will probably take at most a few seconds on your computer today. This performance increase allows you to more freely experiment with the algorithms in this chapter than people have been able to in the past. It also allows you to exercise clearer software design and to quickly explore more sophisticated algorithms, since you need not spend significant time on low-level optimization to obtain reasonable rendering rates.

Despite the relatively high performance of modern machines, we still considered design choices and compromises related to the tension between abstraction and performance. That is because there are few places where that tension is felt as keenly in computer graphics as at the primary visibility level, and without at least *some* care our renderers would still have been unacceptably slow. This is largely because primary visibility is driven by large constants—scene complexity and the number of pixels—and because primary visibility is effectively the tail end of the graphics pipeline.

Someday, machines may be fast enough that we don't have to make as many compromises to achieve acceptable rendering rates as we do today. For example, it would be desirable to operate at a purely algorithmic level without exposing the internal memory layout of our `Image` class. Whether this day arrives soon depends on both algorithmic and hardware advances. Previous hardware performance increases have in part been due to faster clock speeds and increased duplication of parallel processing and memory units. But today's semiconductor-based processors are incapable of running at greater clock speeds because they have hit the limits of voltage leakage and inductive capacitance. So future speedups will not come from higher clock rates due to better manufacturing processes on the same substrates. Furthermore, the individual wires within today's processors are close to one molecule in thickness, so we are near the limits of miniaturization for circuits. Many graphics algorithms are today limited by communication between parallel processing units and between memory and processors. That means that simply increasing the number of ALUs, lanes, or processing cores will not increase performance. In fact, increased parallelism can even decrease performance when runtime is dominated by communication. So we require radically new algorithms or hardware architectures, or much more sophisticated compilers, if we want today's performance with better abstraction.

There are of course design considerations beyond sample statistics and raw efficiency. For example, we saw that if you're sampling really small triangles, then micropolygons or tile rasterization seems like a good rendering strategy. However, what if you're sampling shapes that aren't triangles and can't easily be subdivided?

Shapes as simple as a sphere fall into this category. In that case, ray casting seems like a very good strategy because you can simply replace ray-triangle intersection with ray-sphere intersection. Any micro-optimization of a rasterizer must be evaluated compared to the question, "What if we could render one nontriangular shape, instead of thousands of small triangles?" At some point, the constants make working with more abstract models like spheres and spline surfaces more preferable than working with many triangles.

When we consider sampling visibility in not just space, but also exposure time and lens position, individual triangles become six-dimensional, nonpolyhedral shapes. While algorithms for rasterizing these have recently been developed, they are certainly more complicated than ray-sampling strategies. We've seen that small changes, such as inverting the order of two nested loops, can yield significant algorithmic implications. There are many such changes that one can make to visibility sampling strategies, and many that have been made previously. It is probably best to begin a renderer by considering the desired balance of performance and code manageability, the size of the triangles and target image, and the sampling patterns desired. One can then begin with the simplest visibility algorithm appropriate for those goals, and subsequently experiment with variations.

Many of these variations have already been tried and are discussed in the literature. Only a few of these are cited here. Appel presented the first significant 3D visibility solution of ray casting in 1968. Nearly half a century later, new sampling algorithms appear regularly in top publication venues and the industry is hard at work designing new hardware for visibility sampling. This means that the best strategies may still await discovery, so some of the variations you try should be of your own design!

15.10 Exercises

Exercise 15.1: Generalize the `Image` and `DepthBuffer` implementations into different instances of a single, templated buffer class.

Exercise 15.2: Use the equations from Section 7.8.2 to extend your ray tracer to also intersect spheres. A sphere does not define a barycentric coordinate frame or vertex normals. How will you compute the normal to the sphere?

Exercise 15.3: Expand the barycentric weight computation that is abstracted in the `bary2D` function so that it appears explicitly within the per-pixel loop. Then lift the computation of expressions that are constant along a row or column outside the corresponding loop. Your resultant code should contain a single division operation within the inner loop.

Exercise 15.4: Characterize the asymptotic performance of each algorithm described in Section 15.6. Under what circumstances would each algorithm be preferred, according to this analysis?

Exercise 15.5: Consider the "1D rasterization" problem of coloring the pixel centers (say, at integer locations) covered by a line segment lying on the real number line.

1. What is the longest a segment can be while covering no pixel centers? Draw the number line and it should be obvious.
2. If we rasterize by snapping vertices at real locations to the nearest integer locations, how does that affect your answer to the previous question?

(Hint: Nothing wider than 0.5 pixels can now hide between two pixel centers.)

3. If we rasterize in fixed point with 8-bit subpixel precision and snap vertices to that grid before rasterization, how does that affect your answer? (Hint: Pixel centers are now spaced every 256 units.)

Exercise 15.6: Say that we transform the final scene for our ray tracer by moving the teapot 10 cm and ground to the right by adding 10 cm to the x-ordinate of each vertex. We could also accomplish this by leaving the teapot in the original position and instead transforming the ray origins to the left by 10 cm. This is the Coordinate-System/Basis principle. Now, consider the case where we wish to render 1000 teapots with identical geometry but different positions and orientations. Describe how to modify your ray tracer to represent this scene without explicitly storing 1000 copies of the teapot geometry, and how to trace that scene representation. (This idea is called **instancing.**)

Exercise 15.7: One way to model scenes is with **constructive solid geometry** or **CSG:** building solid primitives like balls, filled cubes, etc., transformed and combined by boolean operations. For instance, one might take two unit spheres, one at the origin and one translated to $(1.7, 0, 0)$, and declare their intersection to be a "lens" shape, or their union to be a representation of a hydrogen molecule. If the shapes are defined by meshes representing their boundaries, finding a mesh representation of the union, intersection, or difference (everything in shape A that's *npt* in shape B) can be complex and costly. For ray casting, things are simpler.

(a) Show that if a and b are the intervals where the ray R intersects objects A and B, then *acupb* is where R intersects $A \cap B$; show similar statements for the intersection and difference.

(b) Suppose a CSG representation of a scene is described by a tree (where edges are transformations, leaves are primitives, and nodes are CSG operations like union, intersection, and difference); sketch a ray-intersect-CSG-tree algorithm. Note: Despite the simplicity of the ideas suggested in this exercise, the speedups offered by bounding-volume hierarchies described in Chapter 37 favor their use, at least for complex scenes.

Chapter 16

Survey of Real-Time 3D Graphics Platforms

16.1 Introduction

Now that you've seen the core ideas that let us use computer graphics to make pictures, we're going to describe to you the variety of approaches that have been developed to encapsulate this knowledge. Such approaches have the benefit of isolating the programmer from the details of the graphics hardware, which helps with maintainability of programs. They also let application developers concentrate on things specific to their application domain, rather than the way that images are presented to the user. The variety of approaches available is due, in part, to the pattern of hardware development, which is where we'll begin our survey.

As impressive as the rate of improvement in commodity CPUs has been during the past four decades, the evolution of graphics hardware has been even more remarkable. Hardware-based graphics acceleration—removing the burden of executing the 3D pipeline from the primary processor by offloading it onto a peripheral—was first commercialized for vector displays in the late 1960s and for raster displays in the 1980s. Rendering pipelines moved from software to raster graphics hardware via the development of geometry- and pixel-processing chips that were integrated into graphics workstations built by high-end real-time 3D graphics vendors such as SGI and Evans & Sutherland, and mid-level-performance raster graphics was provided by workstation vendors including Apollo and Sun Microsystems. These devices were expensive, affordable only to academic and corporate institutions. The maturation of this technology into a true commodity, available on personal computers, took place in the mid-1990s in the form of graphics cards featuring cheap but powerful GPUs (described in Chapter 38).

Since each brand/model of GPU has its own native instruction set and interface, a standard API providing hardware independence is essential. The two dominant APIs providing this important abstraction layer are Microsoft's proprietary

Direct3D[1] (on Windows platforms for desktop/laptop, smartphone, and gaming hardware such as Xbox 360) and the open source cross-platform **OpenGL.**

The goal of these low-level platforms is to provide access to the graphics hardware functions in a hardware-independent manner with minimal resource cost; they are thin layers above the graphics hardware device drivers. A key characteristic is that they do not retain the scene; instead, the application must respecify the scene to the platform in order to perform any update of the display. These **immediate-mode (IM)** platforms thus act as conduits to the graphics hardware, translating a device-independent stream of graphics instructions and data into the proprietary instruction set of the underlying GPU.

You as a developer have the choice of directly using a low-level IM API—which places the application close to the hardware, allowing maximum control—or using a **retained-mode (RM)** middleware platform (such as WPF) that offers the convenient abstraction of a scene graph. RM platforms—described in greater detail in Section 16.4.2—create opportunities for automated performance optimization, and simplify many development tasks by making it easier to express complex constructions. However, when you use an RM platform, you lose the potential for peak performance and may experience delays in access to the latest hardware features when waiting for the next release of the middleware. It is usually a good idea to work at the highest level practical for your application, and use a limited amount of lower-level code for performance-critical features.

It is important to note that the architecture of graphics hardware is in flux, as GPUs become more powerful and more general-purpose. The GPU is rapidly morphing into what might be called a Highly Parallel Processing Unit, which has already brought ray tracing onto commodity graphics hardware and into the realm of real-time rendering. Keep in mind that our focus here is on platforms built using current GPU polygon-rendering architectures.

16.1.1 Evolution from Fixed-Function to Programmable Rendering Pipeline

Graphics hardware and IM APIs have been co-evolving for several decades, each influencing the other. New features on the hardware side have, of course, required API enhancements for access. Simultaneously, developer identification of bottlenecks and limitations in the IM layer and the underlying hardware produces feedback that leads to innovations in graphics hardware. The co-evolution has, over time, caused a major paradigm shift in the IM layer's functionality, as exhibited in Figure 16.1, whose focus is on the evolution of the two most pervasive IM platforms on commodity hardware.

16.1.1.1 The Fixed-Function Era

Commodity graphics acceleration hardware in the early/mid-1990s implemented **fixed-function (FF)** pipelines (similar to WPF's) using industry-standard non-global lighting and shading models (Phong or Blinn-Phong lighting, Gouraud or Phong shading), and depth-buffer visible surface determination as described in

1. Direct3D is the 3D graphics portion of the umbrella suite of multimedia APIs that is known as DirectX. Note that some 3D-related publications will use the two names interchangeably, but the proper way to refer to the 3D functionality is to use the term "Direct3D" or the abbreviation "D3D."

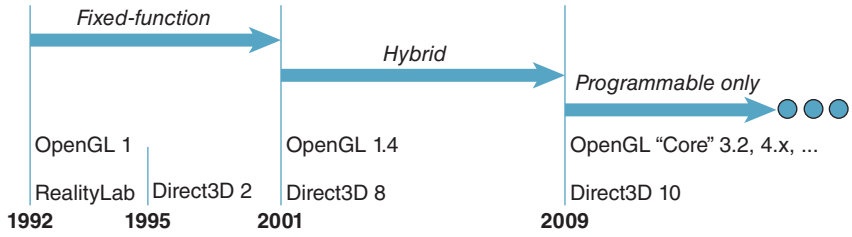


Figure 16.1: Evolution of two important commercial real-time 3D graphics platforms: OpenGL and Direct3D.

Section 36.3. The modules in the pipeline were configurable via parameters, but their algorithms were hardwired and could not be customized or replaced. Thus, the IM platforms in that era were focused on providing access to the FF features, and as new features were introduced in successive hardware releases, the IM APIs expanded to provide access.

16.1.1.2 Extensibility via Shaders: The "Hybrid Pipeline" Era

As the CG community demanded ever-higher levels of control over the rendering process and greater access to GPU capabilities, the popularity of **shaders** surged in the early 2000s—almost a full two decades after their introduction in a seminal 1984 paper by Rob Cook and pioneering implementation as part of Pixar's RenderMan software developed by Hanrahan et al. The term "shader" is misleading, as it would seem to restrict its utility to just surface-color determination—in actuality, the technology encompasses many stages of the rendering pipeline and the term thus refers to any programmable module that can be dynamically installed into the 3D rendering pipeline.

For many years, shader programming had a steep learning curve due to its assembly-language specification; however, in the 2003–2004 time frame, shader programming became more accessible with the development of high-level languages (similar to C) like HLSL/Cg (from a Microsoft/NVIDIA collaboration) and GLSL (introduced in OpenGL 2.0, designed by the OpenGL Architecture Review Board).

The IM layer initially treated shader support as an add-on to the FF pipeline, and for many years, fixed and programmable features co-existed, with applications using the FF pipeline when appropriate and installing supplementary shaders as needed. For example, an animation house working on a movie could use the hybrid pipeline to facilitate real-time tests of scenes before moving on to expensive ray-traced renderings; the availability of shaders could be used to allow the real-time rendering to have at least some special effects such as a water-surface effect that cannot be achieved with the FF pipeline.

16.1.1.3 The Programmable Pipeline

As would be expected, reliance on the FF pipeline has decreased as the expectations of movie audiences and video gamers regarding imaging quality have surged, setting off a race among application programmers and GPU designers to provide the next "cool effect." As a result, in the middle of this century's first decade, both OpenGL and Direct3D began the process of deprecating the FF pipeline. Starting with OpenGL 3.2, the fixed functions have been moved to OpenGL's

Compatibility Profile, no longer considered a mainstream part of the API. Similarly, starting with Direct3D 10, the FF pipeline is no longer available. Thus, the modern IM API is leaner, with far fewer graphics-related entry points, as described in Section 16.3.

16.2 The Programmer's Model: OpenGL Compatibility (Fixed-Function) Profile

In this section, we illustrate the techniques involved in the use of a fixed-function IM platform; to serve as the example platform, we have chosen OpenGL[2] in light of its OS- and language-independence.

OpenGL uses a client/server model in which the application acts as the client (with the CPU being its processing resource and main-memory RAM being the memory resource), and the graphics hardware is the server (with its resources being the GPU and its associated high-performance RAM used for storing mesh geometry, textures, etc.).

The API provides a very thin layer that translates API calls into instructions pushed from client to server. In this section, we focus on the fixed-function API, which is now part of OpenGL's Compatibility Profile. Its fixed-function IM platform operates as a **state machine.** For the most part, each API call either sets a global **state variable** (e.g., the current color) or launches an operation that uses the global state to determine how it should operate.

State variables are used to store all information that affects how a geometric primitive is to be placed/viewed (e.g., modeling transformations, camera characteristics) and how it should appear (e.g., materials). State variables also help you control the behavior of the graphics pipeline by enabling or disabling certain rendering features (e.g., fog).

As an example, consider this pseudocode illustrating state-machine-based generation of three 2D primitives:

```
1 SetState (LineStyle, DASHED);
2 SetState (LineColor, RED);
3 DrawLine (  PtStart = (x1,y1), PtEnd = (x2,y2) ); // Dashed, red
4 SetState (LineColor, BLUE);
5 DrawLine (  PtStart = (x2,y2), PtEnd = (x3,y3) ); // Dashed, blue
6 SetState (LineStyle, SOLID);
7 DrawLine (  PtStart = (x3,y3), PtEnd = (x4,y4) ); // Solid, blue
```

This strategy contrasts with that of an object-oriented system such as WPF, which binds the primitive and its attributes together as illustrated in this pseudocode:

```
1 BundleDASHR =
2    AttributeBundle( LineStyle = DASHED, LineColor = RED );
3 BundleDASHB =
4    AttributeBundle( LineStyle = DASHED, LineColor = BLUE );
5 BundleSOLIDB =
6    AttributeBundle( LineStyle = SOLID, LineColor = BLUE );
7 DrawLine ( Appearance=BundleDASHR,
8             PtStart = (x1,y1), PtEnd = (x2,y2) );
```

2. At a high level, Direct3D's programmer's model is similar, although its API is not.

```
DrawLine ( Appearance=BundleDASHB,
           PtStart = (x2,y2), PtEnd = (x3,y3) );
DrawLine ( Appearance=BundleSOLIDB,
           PtStart = (x3,y3), PtEnd = (x4,y4) );
```

The use of the state-machine strategy is natural for IM platforms, since the goal is to represent the underlying graphics hardware as closely as possible. This strategy has both pros and cons. The advantages include being more concise and supporting control over subordinate modules. Consider a function that draws a dashed triangle:

```
function DrawDashedTriangle (pt1,pt2,p3)
{
   SetState( LineStyle, DASHED );
   DrawLine( PtStart=pt1, PtStart=p2 );
   DrawLine( PtStart=pt2, PtStart=p3 );
   DrawLine( PtStart=pt3, PtStart=p1 );
}
```

What color will the generated triangle be? Since the function controls only the line style, the color is unspecified and depends on the state when control is passed to the function. This has advantages (the caller can control the subordinate's behavior, and the subordinate can produce a greater variety of effects by allowing this control). However, it also has disadvantages: The effect of the function is not fully defined, and debugging unexpected output is difficult because the programmer has to trace backward through the execution flow to the most recent settings of the relevant attributes.

This uncertainty of behavior is actually bidirectional, since subordinate functions are not isolated and can inadvertently produce side effects that damage the caller's behavior. Our function `DrawDashedTriangle` changes the line-style state variable, and thus can have an impact on the caller's behavior and on logic that executes subsequently. The effect will persist until the next explicit setting of the line style. To avoid side effects, each function that changes state should bear the responsibility of restoring state before it returns, as illustrated here in pseudocode:

```
function DrawDashedTriangle (pt1,pt2,p3)
{
   PushAttributeState();
   SetState( LineStyle, DASHED );
   ...
   PopAttributeState();
}
```

Clearly, unless constructed with such protocols to reduce/eliminate side effects, an application built on a state-based platform can produce unintended behaviors that can be difficult to diagnose, so programmer discipline is crucial.

16.2.1 OpenGL Program Structure

In a typical OpenGL application, the program's main function will start by initializing the pipeline, specifying the screen/window location of the **viewport** (as in WPF, the rectangular area on the output device in which the scene will be rendered), setting up camera and lighting characteristics, loading or calculating meshes and textures, setting up event handlers, and finally passing control to

an event-polling loop—at which point the application's role becomes limited to responding to events.

OpenGL itself is window-system-independent and thus has no support for creating and managing windows or handling events. These types of activities, which require highly OS-specific techniques, are typically made available to application programmers via 3rd-party libraries. There are many such libraries, and for this example we've chosen GLUT (OpenGL Utility Toolkit), which has been very popular in OpenGL development for decades. In addition, we use the popular GLU (OpenGL Utility) library for its matrix utilities.

GLUT supports many event types, of which these are the most fundamental.

- **Display:** GLUT calls the registered Display-event handler when it is time for the application to draw the initial image (i.e., when control has just been transferred to GLUT's event-polling loop) or whenever the viewport needs to be refreshed (e.g., to perform "damage repair" as described in Section 1.11).
- **Mouse/keyboard/etc.:** GLUT calls registered interaction handlers to let the application know of the user's attempts to interact with the application through input devices such as the keyboard and mouse.
- **Idle:** To continuously draw new frames as fast as the graphics system can handle them, an application registers an "idle" handler invoked when the graphics pipeline is empty and awaiting new commands. This technique is of value for other purposes as well, such as polling external entities or performing time-consuming operations.

16.2.2 Initialization and the Main Loop

In Figure 16.2, we show the high-level call-graph structure of a typical OpenGL application, with yellow boxes representing modules found in the application and gray boxes representing functions provided by OpenGL and related utilities. Let's examine this call graph in detail here.

OpenGL library bindings are available for a large variety of languages, but in this discussion we'll use C/C++ as our example language. Our program's `main()` function uses GLUT to perform many of the initialization activities, starting with an obligatory call to `glutInit`:

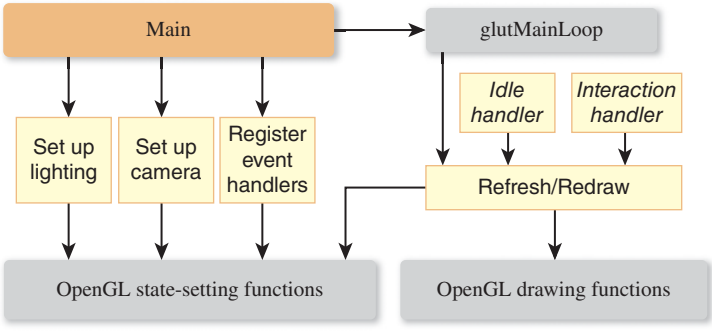


Figure 16.2: Structure of a simple OpenGL application.

```
int main( int argc, char** argv )
{
   glutInit( &argc, argv ); // Boilerplate initialization
```

Next we request color support and depth buffering (also known as Z-buffering, used for hidden-surface removal as described in Chapter 36):

```
   glutInitDisplayMode( GLUT_RGB | GLUT_DEPTH );
```

Then, we call a sequence of three GLUT functions to create the window that will be assigned to our application. The first two function calls allow us to specify the optimal initial size and position for the new window.

```
   // Specify window position (from top corner)
   glutInitWindowPosition( 50, 50 );
   // Specify window size in pixels (width, height)
   glutInitWindowSize( 640, 480 );
   // Create window and specify its title
   glutCreateWindow( "OpenGL Example" );
```

The result is a window whose client area (see Section 2.2) has the specified position and size. The application is free to further divide the client area into different regions; for example, to include user-interface controls. In the following call to `glViewport`, we reserve the entire client area for use as the 3D viewport:

```
   glViewport(
               /* lower-left corner of the viewport */   0, 0,
               /* width, height of the viewport */    640, 480 );
```

OpenGL provides a variety of rendering effects, and with the function calls shown below, we specify that the front side of each triangle be filled using Gouraud's smooth shading. (Alternatively, we can make the pipeline act as a point plotter or as a wireframe renderer, for example.)

```
   // Specify Gouraud shading
   glShadeModel( GL_SMOOTH );

   // Specify solid (not wireframe) rendering
   glPolygonMode( GL_FRONT, GL_FILL );
```

Initialization continues with calls to initialization routines that we will show later:

```
   setupCamera();
   setupLighting();
```

The `main` function is near its end, and still nothing has been drawn. It is time to register our application's display handler to ensure GLUT will know how to trigger the generation of the initial image:

```
   glutDisplayFunc(
        drawEntireScene // the name of our display-event handler
   );
```

The final act in the `main` function is the transfer of control to GLUT:

```
    // Start the main loop.
    // Pass control to GLUT for the remainder of execution.
    glutMainLoop(); // This function call does not return!
}
```

Once GLUT has control, it will invoke the registered display function to trigger the generation of the program's first rendered frame.

16.2.3 Lighting and Materials

OpenGL's fixed-function lighting/materials model is somewhat different from that of WPF as described in Chapter 6, but any effect achievable in one system can effectively be emulated in the other quite readily. Because of this similarity, we omit this portion of the code, but the web materials for this chapter include it.

16.2.4 Geometry Processing

A simplified view[3] of the OpenGL fixed-function rendering pipeline is shown in Figure 16.3; it might be useful to review Section 1.6.2 if some of the terms (e.g., "fragment") are unfamiliar to you. In a fixed-function pipeline, the application's focus is on configuring and feeding data to the per-vertex stage, as is exhibited by the locations of the two boxes representing application input and their varied

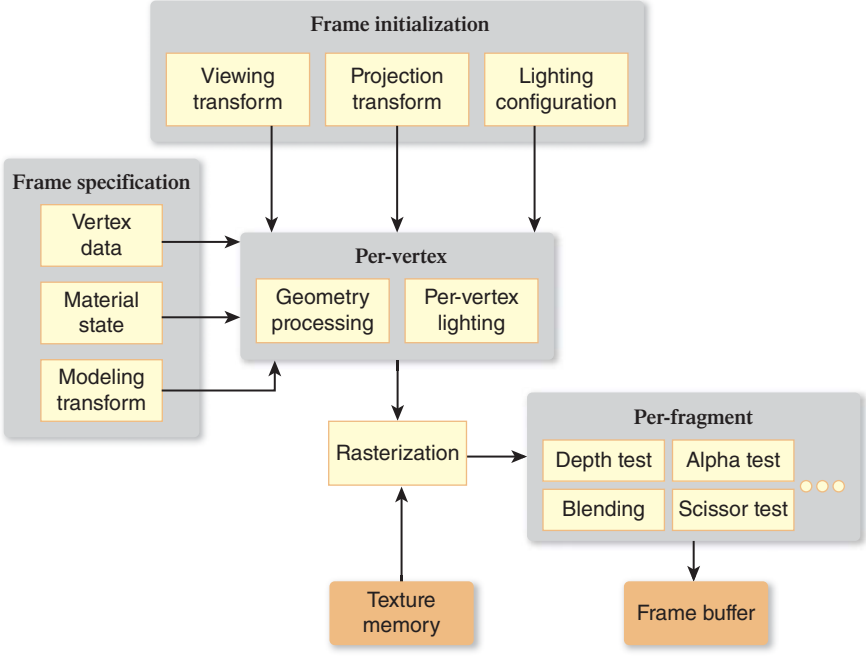


Figure 16.3: Simplified view of the fundamental components of the OpenGL fixed-function pipeline.

3. This simplified view omits pixel data and operations such as bitmaps, images, texture setup, and direct framebuffer access.

contents toward the front of the pipeline. The rest of the pipeline, including rasterization and the many per-fragment operations at the end of the pipeline, use hardwired algorithms controlled via a number of configuration parameters.

As explained in Section 1.6, every 3D graphics system includes **geometry processing** that controls the conversion of geometric data (e.g., mesh vertices) successively from the modeling coordinate system ("object coordinates" in OpenGL nomenclature) to the world coordinate system, continuing on to the camera coordinate system (depicted in Figure 1.15, known in OpenGL as the "eye coordinate system" or "eye space"), and ultimately to some physical "device" coordinate system.

Coordinate-system transformations are performed via matrix arithmetic, as described in Chapters 7 and 11. Matrices are set up by the application using an abstraction provided by the immediate-mode API; we describe the OpenGL fixed-function abstraction below. Internally, the IM layer manipulates the matrices to prepare them for transmission to the GPU. The GPU itself may perform further manipulations to maximize the speed of computations and it must extend the pipeline's scope further to produce physical/screen-pixel coordinates. Our discussion here focuses solely on the IM-level abstraction.

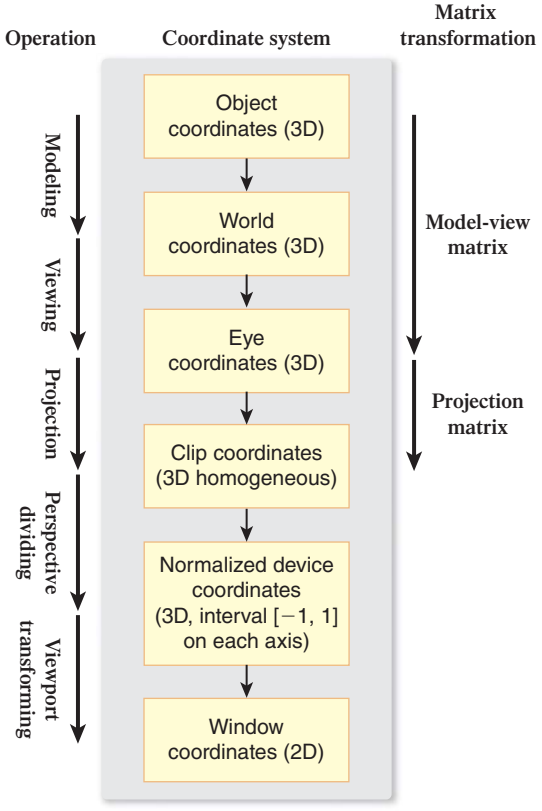


Figure 16.4: OpenGL's geometry pipeline: a sequence of coordinate systems through which each 3D vertex of the original model progresses, via transformations, into its corresponding 2D display-device position.

In fixed-function OpenGL, the geometry processing is performed by what is called the **transformation pipeline,** which has the stages depicted in Figure 16.4.

The **modeling stage** brings the individual components' coordinates from their original "raw" local coordinate systems into a single unified world coordinate system, and we examine how the modeling stage is used to implement hierarchical modeling in the online resources for Section 16.2.9. Then, the **viewing stage** transforms the coordinates into eye space, after which the camera is positioned at the origin and oriented in the canonical way described in Section 1.8.1.

Next, the **projection stage** works on normalizing the view volume's shape and dimensions to OpenGL's version of the standard perspective view volume described in Chapter 13, transforming the coordinates into 3D clip coordinates. The transformation pipeline includes two more stages that eventually yield the window coordinates that are sent to the rasterization module.

In OpenGL, the first three stages of the pipeline are controlled by the application via specification of two matrices, $M_{MODELVIEW}$ and $M_{PROJECTION}$.

The MODELVIEW matrix handles the first two stages; the application has the responsibility of setting it (using utilities described in the next section) according to this equation:

$$M_{MODELVIEW} = M_{view} \cdot M_{model}$$

The order in which the two matrixes are combined ensures the modeling transform is performed on the incoming vertex (V) before the viewing transform is performed:

$$M_{MODELVIEW} \cdot V = M_{view} \cdot M_{model} \cdot V$$

The application separately sets (again using utilities described later) the PROJECTION matrix to define either a perspective or parallel view volume and its corresponding projection.

There is a good reason why OpenGL's designers combined modeling and viewing, but kept projection separate. Typically, the camera view volume's shape (defined by $M_{PROJECTION}$) is quite static, persistent for an entire scene or even for the entire application's lifetime. However, the camera's location and orientation (M_{view}) is typically as dynamic, if not more so, than the scene objects themselves. By isolating the specification of camera shape/projection from the specification of camera position/orientation, OpenGL's design makes it natural to specify these in different parts of the program structure/flow. Normally, setting the projection matrix will be more of an "initialization" operation, whereas specifying the viewing matrix is a part of each new animation frame's computation, typically done just before scene construction commences.

16.2.5 Camera Setup

As described above, the camera specification is divided into two matrix specifications: viewing and projection.

To assist in specifying the latter, GLU provides the convenience function `gluPerspective` to compute the matrix for the common perspective-projection case of a symmetric frustum.

For example, to set the projection matrix (typically as part of an initialization sequence) to match the perspective camera we had specified in Section 6.2 for our pyramid scene, we perform this sequence:

```
// Prepare to specify the PROJECTION matrix.
glMatrixMode( GL_PROJECTION );

// Reset the PROJ matrix to ignore any previous value.
glLoadIdentity();

// Generate a perspective-proj matrix,
// append the result to the PROJ matrix.
gluPerspective(           45, // y-axis field of view
               (640.0/480.0), // ratio of FOV(x) to FOV(y)
                        0.02, // distance to near clip plane
                        1000 );// distance to far clip plane
```

Note that each OpenGL or GLU matrix-calculation function performs two actions: (1) calculates the matrix that meets the specification given by the parameters, and (2) "appends" (via matrix multiplication) it to the current value of the matrix that is currently being specified. This is why the call to `glLoadIdentity` is important; without it, the calculated perspective matrix would be combined with, instead of replacing, the current value of the projection matrix.

Now, let's set up the viewing transformation, typically one of the first operations in the preparation for rendering each frame of the animation. We describe the camera's position and orientation using a GLU convenience function, providing parameters equivalent to those used for WPF camera specification:

```
// Prepare to specify the MODELVIEW matrix.
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();

// Generate a viewing matrix and append result
// to the MODELVIEW matrix.
gluLookAt( 57,41,247, /* camera position in world coordinates */
           0,0,0,     /* the point at which camera is aimed */
           0,1,0 );   /* the "up vector" */
```

16.2.6 Drawing Primitives

OpenGL provides several mesh-specification strategies, including the efficient triangle-strip and triangle-fan techniques described in Chapter 14.

The application is responsible for representing curved surfaces via tessellation into triangles; additionally, the application is expected to provide the vertex normal for each vertex. Since complex models are typically either computed via mathematics (in which case the normal is easily derived as part of that algorithm) or imported by loading a model (which often includes precomputed normal values), this requirement is rarely inconvenient.

The application can choose from several strategies for transmitting the mesh specification to the platform, including techniques for managing and using GPU hardware RAM (e.g., vertex buffers or VBOs, described in Section 15.7.2). In this example, for simplicity, we use the Compatibility Profile's per-vertex function calls which are inefficient (and no longer present in the Core API) but popular in demos and "hello world" programs. To use this strategy as demonstrated in the code below, set the current material, initiate mesh-specification mode, enumerate each vertex one at a time (interleaved with control of the current-vertex-normal state variable), then end the specification mode. Let's specify the same solid-yellow pyramid that we constructed in Section 6.2:

```
// Specify the material before specifying primitives.
GLfloat yellow[] = {1.0f, 1.0f, 0.0f, 1.0f};
glMaterialfv( GL_FRONT, GL_AMBIENT, yellow );
glMaterialfv( GL_FRONT, GL_DIFFUSE, yellow );

glBegin( GL_TRIANGLES );

 The platform is now in a mode in which each trio of calls to
 glVertex3f adds one triangle to the mesh.

// Set the current normal vector for the next three vertices.
// Send normalized (unit-length) normal vectors.
glNormal3f( ... );

// Specify the first face's three vertices.
// Specify vertices of the front side of the face
// in counter-clockwise order, thus explicitly
// identifying which side is front versus back.
glVertex3f(   0.f, 75.f,  0.f );
glVertex3f( -50.f,  0.f, 50.f );
glVertex3f(  50.f,  0.f, 50.f );

// Set the current normal vector for the next three vertices
glNormal3f( ... );

glVertex3f(  0.f, 75.f,   0.f ); // Specify three vertices
glVertex3f( 50.f,  0.f,  50.f );
glVertex3f( 50.f,  0.f, -50.f );

  ... and so on for the next two faces ...

glEnd(); // Exit the mesh-specification mode

 The mesh is now queued for rendering.
```

16.2.7 Putting It All Together—Part 1: Static Frame

In contrast to WPF, which automatically updates the display to represent the current status of the scene graph, immediate-mode packages place the burden of display refresh on the application. First let's look at a static-image generator, and then we'll add dynamics.

Consider the case of a program that is to generate a single static image; in such a case, the rendering activity is only needed

- As part of initialization, to generate the first rendering
- Whenever the window manager reports to GLUT that the image has been damaged and needs repair (e.g., upon closing of a window that was partially covering the OpenGL window)

In this situation, the display function typically looks like this:

```
void drawEntireScene()
{
    Set up projection transform, as shown above.
    Set up viewing transform, as shown above.
    Draw the scene, via an ordered sequence of actions, such as:
         Set the material state.
         Specify primitive.
```

```
8           ...
9
10    // Final action: force display of the newly-generated image
11    glFlush();
12  }
```

16.2.8 Putting It All Together—Part 2: Dynamics

Let's extend the scenario to include dynamics; let's have our simple pyramid model spin around the *y*-axis as though on an invisible turntable, the visualization technique used in the lab software for Chapter 6 to demonstrate the effect of directional lighting.

In an object-oriented system, each node in the hierarchical scene specification supports attached transformation properties, so you would perform this animation by attaching a rotation transformation to the pyramid primitive and tasking an `Animator` element to dynamically modify the amount of rotation. This technique is identical to the clock-rotation technique we used in Chapter 2.

In an immediate-mode platform, we perform modeling by direct manipulation of the MODELVIEW matrix. We've already learned how to initialize the MODELVIEW matrix with the viewing transform. To achieve the spinning dynamics, we must change our scene-generation function slightly, by appending a rotation transformation to the MODELVIEW matrix just before drawing the scene.

We elaborate on this technique, and provide source-code examples, in the online material for this chapter.

16.2.9 Hierarchical Modeling

Previously, we presented two examples of hierarchical modeling: a 2D clock in Chapter 2, and a 3D camel in Section 6.6. In those examples, we learned how to create a scene using a retained-mode platform such as WPF: The application sets up the scene by creating a hierarchy of component nodes (attaching instance transforms to specify initial placement), and animates the scene by adjusting the values of the joint transformations attached to the nodes. This is an intuitive way to work, since the scene graph's structure exactly mirrors the physical structure of the scene being modeled.

Now, reconsider the camel hierarchy shown in Figure 6.41. How can we construct this model using an immediate-mode platform?

The challenges here are twofold.

- The IM platform has no facility for storing the model. The application is wholly responsible for representing the model's hierarchy, and for computing and storing the values of all transforms that control position and orientation.
- Section 10.11 describes the graph-traversal and matrix-stack techniques necessary to compute the **composite transformation matrix** to properly position and orient a particular leaf component in the hierarchy. A retained-mode platform performs this calculation automatically, but this burden rests on the application when using an IM platform.

To satisfy the need for model representation,[4] there are two approaches.

- Create a custom scene-graph module, duplicating functionality found in a typical RM layer (see Section 16.4), in terms of both storage (such as a hierarchical component graph, and a database of transform values) and processing (traversal and generation of the IM instructions).
- Use the program's call-graph hierarchy to represent the model's structure, by writing a distinct function for each type of node (grouping or primitive), with the functions for higher-level nodes calling the functions for lower-level ones.

In the online resources for this chapter, we examine the latter approach in detail, including source code for a complete working example.

16.2.10 Pick Correlation

One of the advantages of an RM platform is its support for **pick correlation,** the determination of the primitive that is the target of a user-initiated mouse click or other equivalent device action. For example, WPF converts a given 2D viewport pick point's coordinates into a **pick path** that identifies the entire path from the root of the scene graph down to the leaf primitive.

Of course, an IM platform cannot automate such functionality since it does not retain the scene. Thus, an IM application often uses custom correlation logic, running an algorithm such as ray casting (Chapter 15) while traversing through the application's scene data store. OpenGL offers an alternative technique that uses the name stack to efficiently automate hierarchical pick correlation. This technique is described in the online materials for this chapter.

16.3 The Programmer's Model: OpenGL Programmable Pipeline

At its core, real-time graphics is still done using lights, meshes, materials, transformations for viewing and modeling, etc. The progression from fixed-function to programmable has not changed its essence. But where these types of objects reside in the application source code has shifted to programs written in shader languages and installed in the GPU, as described in detail in Chapter 33.

16.3.1 Abstract View of a Programmable Pipeline

Let's examine an abstract view of a programmable pipeline, shown in Figure 16.5 and explained in the next few paragraphs. To keep this model simple, we omit texture data/operations, show only vertex and fragment shaders, and omit feedback loops. We use OpenGL terminology, but this is also applicable to Direct3D.

If this diagram seems to be a bit incomplete, you're on the right track! Where are the lights, materials, and camera? They are all present, but only "in spirit":

4. Here, we are concerned with a model that has only the geometric data necessary to generate the target image, but the focus of a typical real-world application is an "application model" (see Section 16.4) that encapsulates many different data types, graphical and nongraphical.

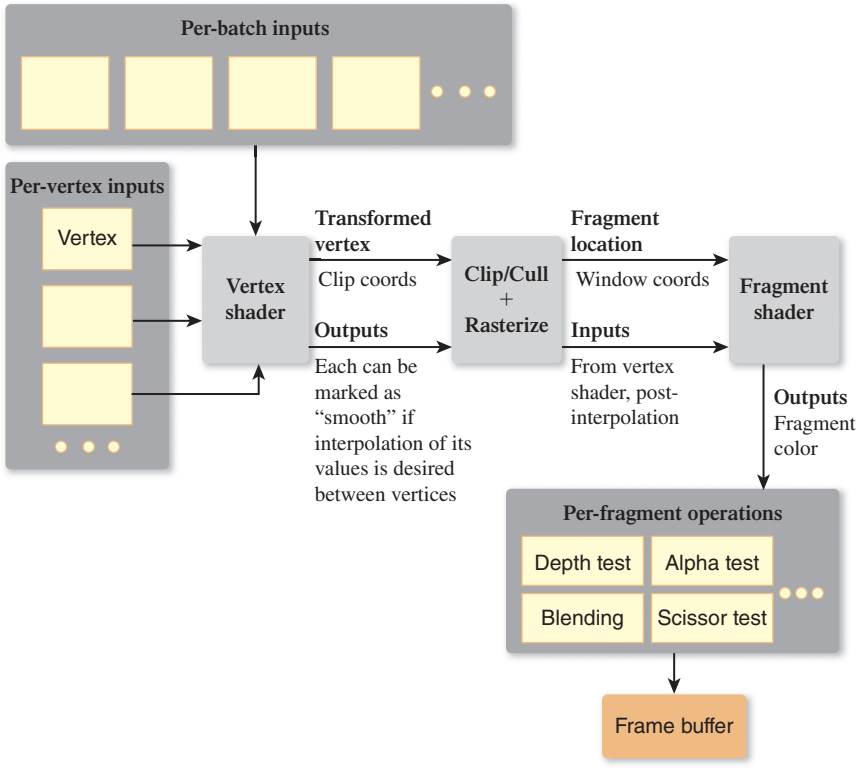


Figure 16.5: Simplified view of the fundamental components of the OpenGL programmable pipeline.

in the empty input-related yellow boxes and in the two shader boxes. Whereas the fixed-function pipeline can be considered a configurable *appliance,* the programmable pipeline is a computer on which you install an application of your own construction. Let's first examine the pipeline's semantics in the absence of a particular application, and then we'll look at how this pipeline might appear when loaded with an actual program.

The application sends batches of vertices through this pipeline, each batch representing a mesh for which certain characteristics (e.g., material or lighting) are constant across its vertices. The **vertex shader** is a function that is called once for each vertex in the batch, receiving as input the vertex and associated attributes such as the vertex normal or the texture coordinate. The vertex shader also has access to any number of per-batch "uniform" inputs providing information that is constant for the entire batch (e.g., camera characteristics).

What the vertex shader does is up to the programmer. At the least, its output must include the vertex transformed to clip coordinates (i.e., already passed through modeling, viewing, and projection transformations), but it also may include any amount of other output data values to "tag along" with this vertex to the next stage of the pipeline. Typically, a calculated vertex color is one of the outputs, but there is no limit to the number of outputs or their semantics. Each output is typically marked as "smooth," which tells the rasterizer stage in the pipeline to interpolate that particular output's values for the pixels that lie between the vertices. (The alternative is "flat," which disables this interpolation.)

What lies between the vertex and fragment shaders is a mini-pipeline of several modules,[5] shown in Figure 16.5 as a single box. They work together to complete the transformation pipeline to convert to window coordinates, to perform clipping/culling against the view frustum, and to perform rasterization to produce a sequence of output fragments.

The **fragment shader** (whose equivalent in Direct3D is called the "pixel shader") is called once for each fragment and receives the fragment's location (in window coordinates) and the already-interpolated varying outputs from the vertex shader. In very simple cases, the fragment shader often does nothing but pass the color unchanged through to the next stage, but the fragment shader is essential for lighting algorithms for which simple linear interpolation between vertices is insufficient, and is also instrumental in special effects such as blurring.

At the end of the pipeline lie the per-fragment operations, similar to those found in the OpenGL FF pipeline.

16.3.2 The Nature of the Core API

The OpenGL Core API retains only a fraction of the entry points that were present to support the FF pipeline. The techniques for specifying and transmitting textures and meshes have not changed significantly, and end-of-pipeline activities like blending and double-buffering are similar. However, all other information and operations now lie inside in the uniforms, attributes, and shader code. The leaner OpenGL Core API is mostly concerned with activities of the following types:

- Buffer object management—control over all data stored on the GPU, including allocation/deallocation and data transmission
- Drawing commands—sending meshes down the pipeline
- Shader-program management—downloading, compiling, activating, and setting up uniforms and attributes
- Texturing management—installation and management of texture data structures for use by vertex and fragment shaders
- Per-fragment operations—control over per-fragment operations at the end of the pipeline, such as blending and dithering
- And framebuffer direct access—pixel-level read/write access

16.4 Architectures of Graphics Applications

We now discuss the general structure of a typical graphics application, some approaches to speed up certain parts of this structure, and various kinds of software designed to offload that work from the typical designer.

16.4.1 The Application Model

A typical 3D application includes an **application model (AM)**—a collection of data, resident in a database or in data structures, whose application-domain

5. On some hardware platforms, this part of the pipeline is also programmable, through use of a third type of shader, known as a "geometry shader."

semantics go beyond a mere rendered image, but for which an image is one possible view of the AM.

In our simple WPF-2D clock application, the AM contained only the current time of day, but it could be extended to include alarm-related data (date, choice of alarm sound, enabling of "snooze control," etc.) or support for multiple time zones. Note that the AM need not include any inherent geometry; for example, there's nothing geometric about the time of day, and the rotation-based analog display of clock time is simply one way to display that data.

Most applications have a heterogeneous AM containing both nongeometric and geometric data, and the latter can be further subdivided into **abstract geometric** (not in a form ready for the IM layer) versus **ready for rendering** (in a form ready for the IM layer, e.g., geometry in the form of a triangle mesh).

Consider the breakdown of the AM of a chess application.

- Nongeometric data would include
  - Current board location of each piece (i.e., the square on which it resides)
  - Record of each move since the game started, to facilitate export of a game "transcript"
  - Chess strategy data used by the game to plan its moves
  - Duration of the game in progress, the player whose turn it is, the amount of time left for her move, etc.
- Abstract geometric data might include
  - Mathematically defined shapes of the pieces (which must be converted into meshes in order to be made ready for rendering)
  - Motion paths, specified as cubic Bézier curves, to support animation of the movement of pieces from square to square
- The ready-for-IM geometric data might include
  - Geometry and materials for rendering the chessboard itself
  - Camera definitions for several points of view (if the UI allows the user to choose from several POVs, e.g., directly overhead, POV from seated avatar, etc.)
  - Modeling transforms for the pieces (e.g., if the user is able to control the 3D positions and orientations of the pieces beyond their abstract locations on specific squares)

Now consider the highly complex AM for a CAD/CAM representation of a jet airliner, consisting of millions of components, each including geometric, spatial layout, and connectivity/joint data; behavioral data used in aircraft-operation simulation; part numbers, costs, and supplier IDs used in procurement; maintenance/repair instructions or cautions; and much more. In addition, each component "lives" in several organizational systems for the purpose of searching and filtering; for example, a spatial organizational system might separate components into regions (e.g., cockpit, main cabin), but a functional one might separate components into distinct systems such as electrical or hydraulic.

These databases thus act as a confluence of many types of data, used and manipulated by a large variety of different systems and applications, of which only a fraction are "computer graphics" programs.

For the purposes of this text, we are concerned only with the data that is either intrinsically geometric, or can be represented geometrically for the purpose of rendering.

16.4.2 The Application-Model-to-IM-Platform Pipeline (AMIP)

We now consider how an IM-based application drives an IM platform, which in turn drives the GPU. Every such graphics application must implement—in addition to its own special semantics/logic—a multistage process that we call the "Application-Model-to-IM-Platform pipeline," or **AMIP.** The AMIP is the front part of the client (CPU) side of the complete rendering pipeline depicted conceptually as a sequence of stages, each executing a designated task, in Figure 16.6.

At its most basic, the AMIP is composed of a traversal of the AM to do the following.

- **Determine the scene to be rendered,** including all geometry, materials, lighting/special effects, and camera configuration. The application traverses the application model to extract the data relevant to the scene, transforming any nongeometric data into a geometric representation for inclusion in the scene. This is analogous to the act of generating a view of a database, an action requiring both selection (extraction based on query criteria) and transformation (arbitrary computation on or reformatting of extracted data fields).
- **Calculate the sequence of API calls** needed to drive the IM layer to produce the image of the scene.

Figure 16.6 depicts the complete rendering pipeline from a functional point of view. Another way to describe a graphics application is from a software-engineering point of view: enumerating the layered **software stack** of components, with the custom application code at the top of the stack, the graphics hardware driver at the bottom, and intermediate platforms/libraries in between.

The graphics-related stack for a typical 3D graphics application is made up of at least three layers (see Figure 16.7), and may contain four layers (see Figure 16.8) if a retained-mode middleware platform is used to assist with AMIP duties.

How the AMIP's tasks are sequenced within the pipeline and divided between layers of the software stack is volatile, as technologies evolve; moreover, AMIP tasks that currently typically live on the CPU side are subject to movement to the graphics hardware as GPU programmability becomes more exploited. We will

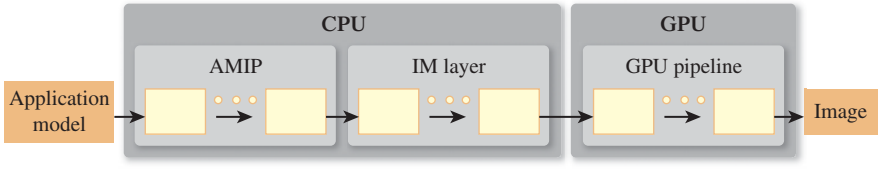


Figure 16.6: Abstract view of the typical application pipeline transforming the application model into a scene delivered to the immediate-mode platform for rendering.

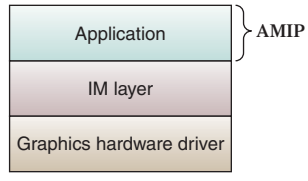


Figure 16.7: Software stack for an application that describes the scene directly to the immediate-mode platform.

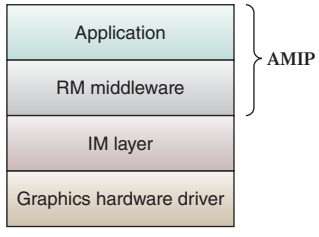


Figure 16.8: Software stack for an application that constructs the scene using retained-mode middleware.

address this "division of labor" later; first let's focus on the kinds of tasks present in the AMIP.

In all but the most trivial applications, it is necessary for the AMIP to work in a highly optimized way in order to ensure **scalability**—that is, to ensure adequate performance (especially for real-time animation) even as the size and complexity of the scene grows. The term "Large-Model Visualization" (often abbreviated LMV) is typically used to describe applications or platforms that handle scenes of extremely high complexity such as a CAD model of a jetliner or cruise ship.

The primary goal of the AMIP's optimization tasks is to reduce consumption of resources such as the following:

- Bandwidth between the CPU and the GPU, by minimizing data transmission to the GPU
- GPU memory consumption, for example by reducing the size of geometry data cached on the GPU
- And GPU processing cycles, by generating an efficient sequence of IM-layer instructions to the GPU to render the scene

CPU-side resources are used to perform these optimization tasks, so there is a tradeoff here: The benefit of the reduced consumption later in the pipeline (especially regarding the GPU) comes at the cost of extra work being done earlier in the pipeline.

In general, the AMIP is composed of stages through which scene data flows. These stages operate sequentially, or partially in parallel, performing optimization duties in three categories:

1. Reducing scene complexity
2. Generating an efficient stream of instructions to the IM layer
3. And avoiding redundant computation activities through appropriate caching

16.4.2.1 Reducing Scene Complexity

There are three categories of tasks that help reduce the amount of graphical data sent to and processed by the underlying IM layer, which we now describe.

16.4.2.1(a) Extracting the Scene from the "Universe"

In large applications, the application model may contain or represent a "universe" of graphical objects that are not necessarily all visible simultaneously. The AMIP thus extracts the relevant subset of the universe, based on application data that specifies the desired rendering goals. For example, in an airplane CAD application, the universe is the entire airplane's specification, and the subset to be rendered might be determined by the user's selection of the subsystems (such as electrical, HVAC, or hydraulic) of current interest.

As another example, consider a multilevel game, in which each level is a completely different subworld—the application need only extract the current subworld to determine the scene to be rendered.

16.4.2.1(b) Reducing the Scene to the Minimal Potentially Visible Set of Primitives

The AMIP's focus here is on high-level culling—elimination of entire primitives or (better yet) large portions of the scene that need not be rendered. This is in

contrast to hidden-surface removal activities performed by the GPU—such as back-face culling as described in Section 36.6, or occlusion culling on a per-pixel level through the depth buffer as described in Section 36.3.

Together, the high-level (CPU-side) and low-level (GPU-side) culling activities work together toward a common goal of reducing scene complexity and thus GPU workload.

It is tempting to think of pre-GPU culling as unnecessary, that as GPUs become more powerful and bandwidth increases, there is less justification for throwing CPU resources at the problem of determining the potentially visible set. However, the GPU-side visible surface determination has a cost that is linear in the number of primitives. Thus, when one considers a Boeing 777 model—which has more than 100,000 unique parts and several million fastener parts—it becomes obvious that there continues to be a need for optimizing the sequence of commands and data sent to the hardware rendering pipeline.

Below is an unordered list of modules of this category, many of which require spatial data structures as discussed below in point 3(b):

View-Frustum Culling: As explained in Chapter 13, the camera's location and viewing parameters determine the geometry of the view frustum, and only geometry lying inside the frustum is visible. This culling stage seeks to identify and eliminate portions of the scene that lie wholly outside the frustum. Implementation is typically performed via arrangement of the scene's contents in a Bounding Volume Hierarchy (BVH), as described in Section 36.7; however, other data structures (e.g., BSP trees discussed in Section 36.2.1) have been used for certain situations (e.g., static scenes).

Sector-Based Culling: In many applications, the scene's environment is architectural, that is, located in the interior of a building, with walls segmenting space into "sectors" and windows/doors creating "portals" that connect adjacent sectors. A number of algorithms, described in Section 36.8 and sometimes called **portal culling** techniques, are available to cull objects in these kinds of environments.

Occlusion Culling: Consider a scene modeling midtown Manhattan, seen from the point of view of a pedestrian at just one intersection. If the depth of the view frustum covers many city blocks, each visible surface, especially those close to the viewer, is occluding a very large number of objects. In these types of environments, there can be great advantage in removing these occluded objects.

Contribution Culling/Detail Culling: A visible primitive, or an entire subportion of the scene, may be too small and/or too far away to make an impact on the rendering. This culling step is designed to detect and dismiss such content. Some applications might choose to use this type of culling only when the viewer is in motion, since the absence of small objects will very likely go unnoticed during dynamics but may be detectable when the camera is at rest.

16.4.2.1(c) Reducing the Transmission/Rendering Cost of Geometric Shapes

In this set of activities, complex geometric shapes specified via meshes either are encoded to reduce the GPU-side rendering cost or the size of the data buffers needed to transfer the specification to the GPU, or are simplified by reducing the mesh's complexity (e.g., reducing the number of triangles and vertices).

Reencoding is the act of converting the mesh's specification to one that is more quickly processed by the graphics hardware. For example, converting to

triangle strips [EMX02] is common since many hardware pipelines are highly optimized for that succinct encoding type.

Simplification is a form of compression similar to the image compression of JPEG or audio compression of MP3. The compression is **lossy** in that the resultant geometry is not as accurate as the original. But just as MP3 compression produces music files that are "good enough" for many purposes, geometric simplification also can be tuned to be satisfactory for specific applications.

For example, if an object or the viewpoint is in motion, it may be possible to simplify the object without the viewer being aware. Or, if the object's "importance" (measured in terms of how many pixels its projected image takes up on the viewport) is lower than some threshold, a certain amount of simplification may be possible without damage to the object's legibility.

There are numerous types of geometric simplification, including the following.

- **Continuous Level of Detail / Multiresolution Geometry / Geomorphing / Selective Refinement/Progressive Meshes/Hierarchical Dynamic Simplification**

  A number of algorithms, known by a variety of names, can be used for automatic simplification of a mesh in a manner that fine-tunes the amount of simplification based on the importance (as described above) of the object's image. These algorithms vary widely in their strategies and usage scenarios, and they should be studied by any developer needing "just enough but not too much" simplification.

  As an example, a progressive mesh (described in detail in Section 25.4.1) provides for storage of a mesh at many resolutions in a single data structure. The structure can be thought of as a sequence: The coarsest (least-expensive, lowest-quality) mesh is stored as the "core," followed by a sequence of "reconstruction records" describing how to incrementally restore the higher-resolution information. The renderer of this mesh sequence can choose to stop reconstruction at any point in the sequence; the more the processor continues to execute the reconstruction records, the closer the resultant mesh is to the original resolution.

  Note that opportunities for implementation of some of these algorithms lie both in the AMIP and directly on the graphics hardware.

- **Discrete Level of Detail**
  When the application needs full control over the simplified geometry, this technique can be used instead of fully automated mesh simplification. Key objects are specified via multiple mesh definitions (e.g., high-, medium-, and low-detail versions), and the application uses the one appropriate for the object's current importance (as described above).

- **Simulating Complex Geometry**
  For certain applications (e.g., the bumpy surface of an orange, or rocky terrain seen from afar), instead of representing the geometry in the actual mesh and bearing the cost of lighting/shading calculation, an application can use tricks to fool the eye into seeing complex geometry that is not actually part of the mesh. Texture mapping, as described in Section 1.6.1, is a primitive technique that wraps the mesh with an image in order to provide color and translucency variation. More sophisticated algorithms, described in Chapter 20, include normal mapping, displacement mapping, bump mapping, and procedural texturing. As an extreme case of cost reduction,

consider the "billboard" technique described in Section 14.6.2, in which far-away complexity is simulated via a texture-mapped planar polygon.

- **Subdivision Surfaces / GPU Tessellation**
  In contrast to simplification techniques, this technique has the opposite goal—it uses an iterative subdivision algorithm (see Section 14.5.3 and Chapters 22 and 23) to *add* complexity to a coarse "base mesh" to provide a smoother appearance. This is an optimization technique due to its use of the GPU to perform the tessellation; the CPU side deals only with the coarse base mesh, thus reducing use of CPU/GPU bandwidth. Since the process is iterative, the GPU-based tessellation can adjust the amount of smoothing work based on the primitive's distance from the viewer, thus providing a kind of variable level-of-detail control.

16.4.2.2 Generating an Efficient Sequence of IM-Layer Instructions to Render the Simplified Scene

The graphics hardware pipeline is a complex combination of functional units, and achieving maximum throughput requires expert knowledge of the pipeline's idiosyncrasies and potential bottlenecks. Certain types of operation sequences can cause "pipeline stalls" that radically undermine performance. As the pipeline has adapted to relieve these bottlenecks over the years, new bottlenecks have arisen, presenting opportunities for further adaptation.

In particular, state changes (e.g., a change in the current-material state variable, or a switch to a different vertex shader) disrupt pipeline throughput and should be minimized by careful ordering of the primitive-drawing sequence. The actual cost varies by API, hardware platform, driver software, and type of state variable being modified. Nevertheless, as a rule, each state change should be followed by the generation of as many primitives as possible; thus, as part of the AMIP, logic should analyze the potentially visible set for the purpose of reordering the primitives so as to draw in a batch all primitives that require the same state configuration. In a chess application, one might model all the black pieces as obsidian and the white pieces as onyx. It then makes sense to render all the black pieces before all the white ones, or vice versa.

The modification of specification order generally does not have an impact on the final rendered image, but it should be noted that the use of translucent materials presents an exception and does complicate this optimization task (and others, e.g., occlusion culling).

16.4.2.3 Using Caching to Avoid Redundant Computations in Performance of Tasks in Categories 1 and 2

The CPU and memory resources necessary to perform the activities described above for task categories 16.4.2.1(a–c) and 16.4.2.2 can be substantial. But many of these activities, when executed to produce frame i, produce results that remain useful for frame i+1, if the difference between the two frames meets certain requirements. Thus, caching is of value in reducing the CPU cost of such activities.

Caches used for this purpose are called **acceleration data structures** and are used in two distinct ways.

- **Cache the *result* of computations.**
  - For example, consider 16.4.2.2. The generated IM-layer instruction sequence associated with static portions of the scene can be cached and

reused in successive frames; moreover, this cache can even be downloaded to the graphics hardware as a **display list** to eliminate its being redundantly sent across the CPU/GPU boundary.

  - As another example, consider 16.4.2.1(b). Simplified meshes computed through algorithms such as those listed above should also be cached, and, here again, there is the opportunity for either client-side (CPU) or server-side (GPU) storage.

- **Cache the *data that is used* in the performance of these computations.**
  - For example, consider 16.4.2.1(a). View-frustum culling is typically performed by organizing the candidate primitives into a BVH (see Chapter 37), cached and reused across frames; moreover, the longevity of this cached data structure can be extended by ensuring that it is selectively updated as needed when changes are made to the scene's geometry.
  - As another example, consider 16.4.2.1(b). The decision-making logic for the various types of conditional simplification is typically performed through computation of the **cost** (a measurement of the geometry's complexity and thus of the cost associated with rendering it) and **value** (a measurement of how much screen real estate its image will take up) for each graphics object. The value:cost ratio is then used by the algorithm that determines how much simplification of a particular object is tolerable. The value/cost information can be cached, of course, keeping in mind that the "value" parts of the cache will need to be invalidated as POV changes occur.

The software that maintains an acceleration data structure is often nontrivial in design; we address this topic further in Section 16.4.3.

As you were reading the above list of common AMIP tasks, you may have been creating a mental image of a well-defined sequential pipeline such as that shown in Figure 16.9.

It's important to note again that this is a highly conceptual view of the complete AM-to-rendered-image graphics pipeline. Real-world implementations vary from this abstract view in several ways.

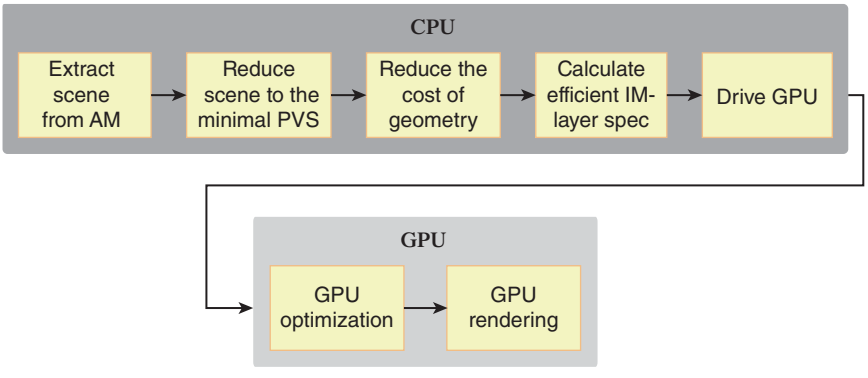


Figure 16.9: Sample sequence of components in a typical AMIP.

- The tasks are not necessarily sequential; the use of parallel computation may be possible for some tasks.
- The order in which we show these tasks is conceptual and not necessarily adhered to by actual software/hardware.
- Some types of tasks may be split across several software/hardware modules, and (for extremely high scalability) across multiple CPUs and GPUs.
- We've not discussed GPU internals here; consult Chapter 38 for a discussion of modern graphics hardware.

With this in mind, let's next turn our attention to the CPU-side software stack, in particular the four-level stack, which includes a retained-mode middleware layer designed to handle a large proportion of the AMIP's duties.

16.4.3 Scene-Graph Middleware

You have now been introduced to two abstraction levels for 3D graphics specification: immediate mode (IM) and the higher-level retained mode (RM). You already know that two key responsibilities of the RM layer are to provide for storage of a scene graph, and to provide functionality to traverse the scene graph to generate the instructions to be sent to the IM layer to produce the image.

Additionally, in Section 16.2.9, we noted that an application built directly atop the IM layer typically necessarily contains custom-built modules duplicating basic RM functionality, simply because scene storage (and converting such into an image) is so commonly needed in graphics applications.

Let's take a more complete look at the functionality of RM middleware. An RM-based application extracts information from the application model, and uses the RM API to construct the scene graph that resides in the middleware layer. Some of the programmer conveniences gained by the middleware's maintenance of the scene description include

- Object-oriented representation of key 3D graphics concepts (primitives, transforms, materials, textures, camera, lights), providing programmers with an intuitive API for setting up scenes
- Support for hierarchical modeling
- Support for dynamics through incremental editing of the scene description
- Support for hierarchical pick correlation

These conveniences are found in virtually all RM middleware; however, in addition to these features, RM middleware designed for optimal performance optionally can perform any of the post-application-model AMIP optimization tasks listed in Section 16.4.2, as shown in Figure 16.10. Many RM layers perform very little optimization, some are designed for optimal scalability and performance, and some are simply conveniences offering no optimization at all. (For specifics, visit the online materials for this chapter, which contain a curated list of scene-graph platforms.)

16.4.3.1 Optimization via Acceleration Data Structures

To achieve performance gains, some RM implementations build acceleration data structures (described in Section 16.4.2.3) to store information useful for optimization. Designing the software logic that maintains these data structures—illustrated in Figure 16.11 and covered throughout Chapters 36 and 37—is a nontrivial task,

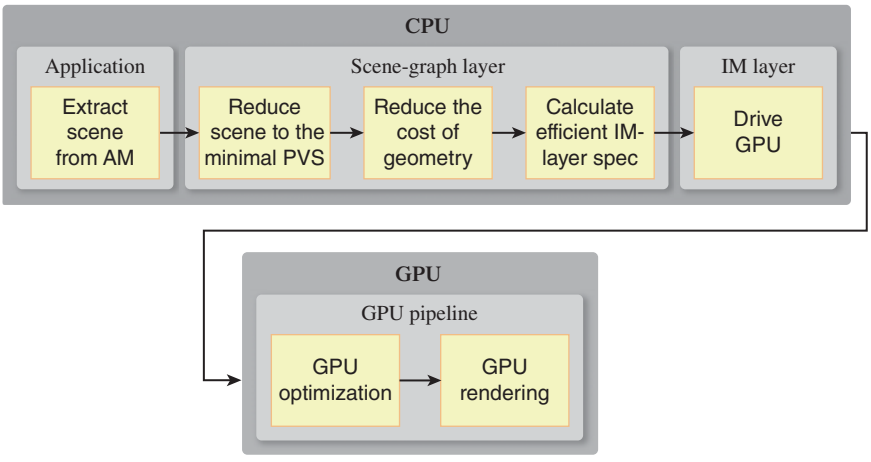


Figure 16.10: Sample distribution of AMIP responsibilities in an application using retained-mode middleware.

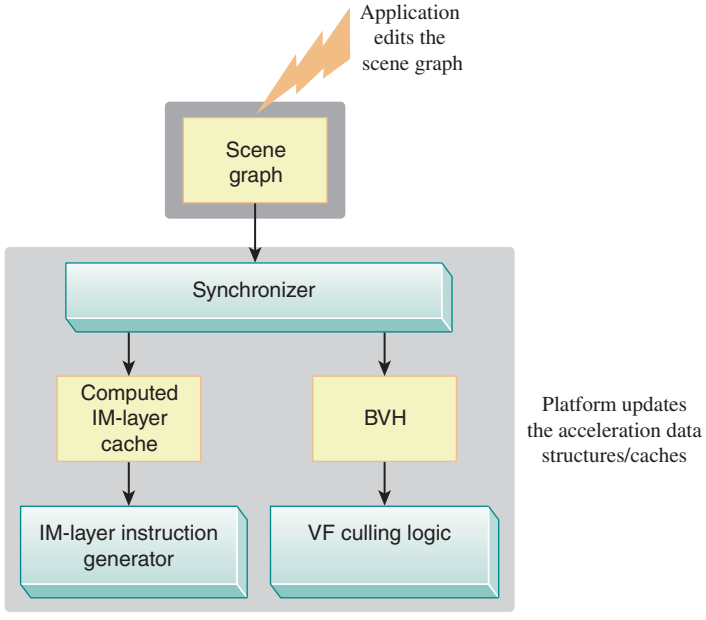


Figure 16.11: Abstract depiction of an RM layer providing two types of optimization discussed in Section 16.4.2 (view-frustum culling and IM-instruction reuse), showing the synchronization logic that ensures the acceleration data structures (BVH and IM-instruction cache, in this example) are updated when the scene graph is modified.

because the structures must be maintained as the scene graph is modified, requiring sophisticated logic to avoid unnecessary invalidations (i.e., premature discarding) of cached information, and to avoid the expense of wholesale regeneration of the structures.

In addition to the CPU time allocated to maintaining these structures, there is also a nontrivial CPU memory cost as well. At the end of this section, we'll visit the issue of the runtime cost of scene-graph middleware and discuss the cost/benefit tradeoffs.

16.4.3.2 Optimization of Static Scene Portions

An initially counterintuitive but then "obvious" maxim about scene graphs, proffered by Henry Sowizral, the designer of Java3D (a pioneering RM platform in terms of optimization support), is simply: *Traversal of a scene graph is expensive and should be avoided to the fullest possible extent.*

In particular, during real-time animation sequences, it is untenable for the scene graph to be retraversed for each individual frame of the animation. A scalable scene-graph platform thus minimizes traversal by performing tasks of the type mentioned in Section 16.4.2.3:

- Identifying—either automatically or via application-provided "hints" such as those in Java3D—graph nodes with static content
- Generating and caching acceleration structures for such "subgraphs"
- And not allowing dynamic components in the scene to affect these cached acceleration structures (which would lead to a high frequency of unnecessary cache invalidations)

When using an RM layer, you can assist the middleware with this task by isolating static from dynamic components in the scene graph. For example, if the root node's first child is a node *X* containing all static parts of the scene, the subgraph rooted by *X* need be traversed only once, and that traversal will prepare the acceleration data structures that provide for efficient generation of that part of the scene. The remaining children of the root would be marked as dynamic and not allowed to alter the acceleration structures for the static portion of the scene.

In an application that allows the viewer to travel through a static scene the entire scene graph can be handled in this way—that is, traversed exactly once to set up the acceleration structures that will efficiently generate each frame of the animation.

It is interesting to note that the generation and caching of the acceleration structures often involves "flattening" the scene-graph hierarchy, to eliminate traversal dependencies. Why? Well, remember Sowizral's admonition: *Avoid traversal!* The goal of the acceleration structures is to give the optimization algorithms rapid access to ready-to-use data needed for their operation; to require the optimization logic to traverse the scene-graph hierarchy in order to interpret the acceleration data would be a serious slowdown.

16.4.3.3 Costs and Disadvantages of Retained-Mode Middleware

The benefits provided by an RM layer come at a cost. Development-related costs include the learning curve associated with the package's API, and the time needed to gain the experience required to efficiently diagnose and repair bugs during development. (Of course, there is also a financial cost for commercial middleware products.)

Runtime costs include both CPU-side memory and processor usage, primarily in these two categories.

1. The scene graph itself is stored on the CPU side and its memory requirements can be nontrivial for highly complex scenes. Moreover, if the AM is predominantly geometric itself, the scene graph might be considered quite redundant with the AM.

2. The acceleration structures/caches used internally by the middleware's optimization modules use CPU-side memory and can be nontrivial in size for highly complex scenes.

It is interesting to reexamine our camel-modeling application, and compare our OpenGL function-based approach with a scene-graph approach such as that presented in Chapter 6. Consider a desert scene with 100 camels moving as a caravan. What are the CPU-side memory requirements for each of these strategies?

In Section 6.6.4, we showed how reuse of the composite components at different levels of the hierarchy affected the number of nodes in the scene graph. We saw that component reuse resulted in resource savings at the cost of loss of detail in the animation control. Ultimately, if our target animation quality requires individual control over every joint in the scene, reuse of composite components is not possible and the cost of the scene-graph storage is at its highest. (Of course, the mesh data associated with the atomic components can be shared without any loss of control over joint animation.)

By contrast, it would at first glance appear that the function-based approach of Section 16.2.9 is highly scalable. The cost of the representation of the hierarchy's *design* is indeed constant, since it lives in the compiled executable and is unrelated to camel count. Indeed, if the goal was to render 100 camels in random locations for a still-frame rendering, one could write a program calling the `Camel()` function 100 times with random nonretained instance and joint transforms. The CPU RAM cost would be truly a constant, unrelated to camel count. However, if animation of the caravan is required, at the very least the application model must include per-camel location/orientation status. And if complete control over each joint is required for high-quality animation, this AM-resident storage of camel information (with location, orientation, and status at all joints) starts looking more and more like a scene graph. The higher the requirement for control over the scene's details, the more the AM will start having many of the qualities of scene graph, and the more time the development team will spend building what is essentially a custom scene graph and custom AMIP.

A development team choosing between constructing a custom AMIP and using a middleware platform to offload a lot of responsibility should take these costs into account. As noted, some of the costs (e.g., acceleration data structures) are unavoidable; however, a custom AMIP does offer the possibility of avoiding duplication of AM data. The redundancy of a middleware-resident copy of AM data, and its CPU-side cost in terms of both memory and processing cycles, is seen by many as a fundamental problem with general-purpose, domain-independent scene-graph middleware technology.

Yet, the conveniences of a hierarchical scene-graph and the automation of AMIP optimization are highly beneficial, and thus a more efficient platform architecture (described in the next section), which merges the AM and scene-graph data, has become popular in some specialized domains.

16.4.4 Graphics Application Platforms

In a few key application domains with large developer communities, the need for the convenience and optimizations of a scene-description database, coupled with the need for solutions to other problems common to that domain, have led to the development of what we call **graphics application platforms.**

The most prominent domain of this kind is game development. Consider the list of tasks associated with producing a 3D interactive game:

- Highly optimized AMIP
- Statistics/history recording/"score keeping"
- Audio (background music, synchronized sound effects)
- Physics for realistic dynamics
- Networking (for multiplayer games, LAN-based or Internet-based)
- Artificial Intelligence (for character/object autonomous behavior)
- Input-device handling

To provide a feature-rich foundation for game development, a number of **game application platforms** (often called **game engines**) have been made available as commercial products or open source projects. Many have evolved as a side effect of a team's creation of a specific game product or series. Internal to the runtime module of a game application platform is a set of databases that can together be considered a "super scene graph" that stores the scene information interleaved with the application model. For example, in an auto racing game, a template representing an instantiable car might contain not only the expected geometric information (e.g., a scene-graph template representing the car's geometry and appearance), but also its maneuverability characteristics (e.g., acceleration limitations, handling characteristics in sharp turns, etc.). Moreover, an instance of that template, representing an actual car involved in an ongoing race, would carry additional game-related information (e.g., current velocity, current angular momentum) in addition to the geometric current-location/orientation information that would be present in a normal scene graph.

Game development involves much more than just implementing the executable runtime, and thus mature game-development systems also often include utilities/IDEs that assist in design-time activities. For example, the popular Unreal game-development environment includes tools for the following:

- Three-dimensional model construction/editing, including facilities for loading existing models from a variety of file formats
- Character skinning/rigging for designing humanoid/animal figures that move realistically
- Art direction, including background painting, material design, and lighting design

Of course, graphics application platforms are available in other disciplines as well. For example, in the CAD/CAM domain, Autodesk's AutoCAD system has evolved from an application to a platform with a high amount of configurability, a "super scene graph" database, and a rich API. Instead of trying a one-size-fits-all approach, AutoCAD offers distinct environments with targeted AM semantics for several subdisciplines, such as mechanical, architecture, factory-floor layout, etc.

16.5 3D on Other Platforms

The computational requirements of 3D applications have in the past kept them as standalone applications running on desktop/laptop PCs or on specialized gaming platforms such as Xbox 360, PlayStation, Wii, etc. However, progress in both

hardware capacity and software development is poised to increase the viability and use of "embedded" 3D.

16.5.1 3D on Mobile Devices

The typical high-end smartphone or tablet includes a hardware-accelerated, programmable pipeline that includes vertex and pixel shaders. To provide a consistent hardware-independent API for these devices, a lightweight version of the OpenGL API has been developed, named **OpenGL ES** (Embedded Systems). This API is currently pervasive in the mobile space, with the notable exception of Windows-based phones and tablets driven via DirectX 9 APIs.

The design of the ES variant primarily involved adjusting to the limitations of mobile devices with regard to processing capability, memory availability, memory bandwidth, battery life, etc. For example, precision qualifiers were added to the shading language to allow applications to choose lower numeric precision to reduce use of the processor. Some features that place a large burden on the processor, such as pseudorandom noise computation, were eliminated. Additionally, the ES variant promotes the strategy of downloading precompiled (binary) shaders, since compilation of shader code is computationally expensive.

16.5.2 3D in Browsers

Efforts to define a text-file format for 3D scene specification, suitable for Internet delivery of 3D content to web browsers (as well as for cross-application transfer of scene/model specifications), date back to 1994's first version of **VRML** (Virtual Reality Modeling Language). Now extensively evolved and renamed **X3D,** this ISO standard maintained by the Web3D Consortium provides XML declarative specification of 3D scene graphs, supporting the fixed-function pipeline and shader extensions. Special scripting and interaction/animation nodes provide some dynamics, and navigation nodes provide for setting up walkthrough/flythrough navigation, making it more than just a generator of static images. However, the lack of native support for X3D in popular web browsers has slowed adoption, and the format has not gained traction with website authors outside of academia.

The potential for widespread use of 3D content on websites is far higher with **WebGL,** a JavaScript API native to most prominent browser brands, supporting immediate-mode 3D rendering into the HTML5 canvas. Based on OpenGL ES, it has no fixed-function pipeline and requires the use of shaders for all appearance control. Thus, programmers wanting a fixed-function model and/or a retained-mode scene graph will rely on middleware platforms, of which several are currently in development.

For information on this rapidly evolving topic, access the online materials for this chapter.

16.6 Discussion

This chapter has provided a brief introduction to graphics platforms with differing design goals and levels of abstraction. No one model has been or is likely to become dominant any more than one programming model or language has become dominant. Developers will be able to choose how much control they want to have over the underlying GPU hardware, much as they have the choice of whether to

program in assembly language, C, or a higher-level procedural, object-oriented, or functional language. Developers of 3D graphics will have the choice of programming at a "low" level by writing shader programs to take advantage of all the latest algorithms and tricks of the rapidly developing art and science of rendering (both photorealistic physics and cartoon physics), typically in OpenGL in its various forms or in Microsoft's Direct3D. Alternatively, they can sacrifice that kind of direct control of the GPU by programming at a higher level of abstraction, one that has a much less steep learning curve. At that higher level, they can program in immediate mode and maintain their own data structures to drive the GPU, or they can take advantage of retained mode, which offers convenient functionality especially for displaying hierarchical models. The more the package aids the developer, the greater the chance that some performance is sacrificed, just as it is with the use of higher-level languages with many features. At the time of this writing, shader programming clearly is the dominant programming model, but this may well change. The one trend we can comfortably predict is that mobile computing, taking advantage of both rapidly increasing device performance and better cloud services, will make available on smartphones and tablets the amazing real-time graphics provided today by high-end graphics cards.

Chapter 17

Image Representation and Manipulation

17.1 Introduction

Digital imagery appears in all forms of media today. Although most of these images are digital photos or other types of 2D pictures that have been loaded or scanned into a computer, an increasing number of them are generated in 3D using sophisticated modeling and rendering software. Accompanying this trend is a large number of image formats, most of which are interconvertible (albeit with some loss of fidelity). Images in each format have limitations, especially in their ability to represent wide ranges of intensity; as a result, new formats for **high dynamic range (HDR) images** have also evolved. Because most images come from digital cameras, it's natural to think of each pixel as storing a red, a green, and a blue value (an **RGB** format), and then using the values to drive the red, green, and blue colors of a screen pixel when the image is displayed. But in practice, especially with digital images, each pixel is likely to contain considerably more information. The pixel may also contain a **depth** value representing distance from the virtual camera, an **alpha** value representing a kind of transparency, and even values like an identifying constant that tells what object is visible in this pixel.

In this chapter and the one that follows, we discuss how images are typically stored, and some techniques for manipulating them, including compositing. Then we examine the content of images more carefully, determining how much data an image can hold and what this says about the operations we can perform on it reliably. Finally, with this richer view of images, we discuss different forms of image transformation and take a look at their benefits and limitations.

17.2 What *Is* an Image?

We'll start with a definition, which we'll later refine somewhat: An **image** is a rectangular array of values, called **pixel values,** all of which have the same type. These pixel values may be real numbers representing levels of gray (a **grayscale** image), or they may be triples of numbers representing mixtures of red, green, and blue (an **RGB image**),[1] or they may contain, at each pixel, other information in addition to color or grayscale data; a rich example is so-called ***z*-data,** indicating at each pixel the distance from the viewpoint from which the image was captured or produced.

A rectangular array of numbers can be interpreted in many ways. For instance, it's possible to display a z-data or depth image in grayscale, in which case the parts of the image that are near the viewer are displayed in lighter shades of gray than the parts that are far away. A priori, the numbers in the array have no particular significance. But for practical matters, when we take a digital photograph we'd like to know whether the pixels store red-green-blue triples or green-blue-red triples, since any confusion could cause very peculiar pictures to be displayed or printed. Thus, image data is typically stored in certain standard file formats, where the meaning of the data associated to each pixel is standardized. Some formats, notably TIFF (**Tagged Image File Format**), allow you to associate a description to each datum. For instance, the description of a TIFF file might be "Each pixel has five values associated to it: a red, green, and blue value represented by an integer ranging from 0 to 255, a z-value represented by an IEEE floating-point number, and an object identifier represented by a 16-bit unsigned integer." With this in mind, we begin our discussion of images with the mundane and practical issue of how conventional file formats store and represent rectangular arrays of data.

How these rectangular arrays of values actually represent light intensities (or other physical phenomena) and how *well* they do so is also important. Following our discussion of image file formats, we move on to discuss the content of images.

17.2.1 The Information Stored in an Image

When we have a typical image file format, storing an $n \times k$ array of grayscale values or RGB triples, it's natural to think about operations like adding together two images, pixel by pixel (or averaging them, pixel by pixel), to create effects like a cross-fade. To do such things requires a notion of addition of images and of multiplication by constants, which we take from the operations on each pixel (i.e., to add two images, we add corresponding pixel values). For grayscale images, what we have, in effect, is a correspondence between the set of $n \times k$ images and the elements of $\mathbf{R}^{nk}$, given by enumerating the pixel values in some fixed order. Thus, the set of all images forms a subset of an nk-dimensional space.

Inline Exercise 17.1: Each element of the standard basis for $\mathbf{R}^{nk}$ consists of $nk - 1$ zeroes and a single one. What does the corresponding image look like? Can you see how you could represent every image as a sum of scalar multiples of such "basis images"?

1. The precise meanings of the red, green, and blue values may be quite vague; we'll discuss this thoroughly later.

Contrast this description of an $n \times k$ image with the scene you witness as you look through a window: At every point of the window, you perceive a color with some amount of lightness. That is to say, we could summarize your percept as a function from points of the window to real numbers representing lightness (measured in some way) at the points. The set of all real-valued functions on a rectangle constitutes an *infinite*-dimensional vector space. We choose, however, to represent such images with $n \times k$ representative numbers, that is, an element of a *finite*-dimensional vector space. There is necessarily some loss in the conversion from the former to the latter. The exact nature of this loss depends on how the finite image was created; we'll see that the choices made during image formation (whether via a camera or via a software renderer) can have far-reaching impact.

17.3 Image File Formats

Images are stored in many formats; typically the storage format bears a close resemblance to the display format. That is, an $n \times k$ image may be stored as nk triples of RGB values, with each R value representing the red part of a pixel, stored in a sequence of some fixed number of bits, and similarly for G and B. But some formats have more complex representations. For example, we might, considering the red values only, reading across a row of an image, store the value of the first pixel, and the *difference* of the second from the first, and then the difference of the third from the second etc. Because these differences will tend to be smaller numbers, we might hope to store them with fewer bits. This would give a **losslessly compressed** image: one in which the data occupies less space, but from which the original RGB image can be reconstructed.

On the other hand, sometimes we can use **lossy compression**—a method of compressing an image so that some of the original data is lost, but not enough to matter for the intended use of the image. A simple lossy compression scheme would be to store only a checkerboard pattern of alternate pixels and then, at display time, interpolate missing pixel values from the known neighboring values. This generates a two-to-one savings in storage, but at a cost of substantial image-quality loss in many cases. More sophisticated compression schemes use the known statistics of natural images and known information about the human visual system (e.g., we're sensitive to sharp edges, but less sensitive to slowly changing colors) to choose which data in the image to keep and omit. JPEG compression, for instance, divides the image into small blocks and compresses the data stored in each one; it's easy to see the blocks if you zoom in on a displayed JPEG image.

Some formats also store **metadata** (information about when the image was produced, what device or program produced it, etc.) and, in some cases, information about the contents, which is typically described in terms of **channels.** The red values for all pixels constitute one channel, called a **color channel;** there are corresponding blue and green channels. The colors stored in one color channel may be represented by small integers with some number of bits, so we speak of an "8-bit red channel" or a "6-bit blue channel." The image metadata gives information like the "bit depth" of each color channel. If the image also contains a depth value at each pixel, we speak of a "depth channel"; the metadata describes such noncolor channels as well.

17.3.1 Choosing an Image Format

Most digital cameras produce JPEG images; because of this, JPEG has become a de facto standard that is especially appropriate for natural images containing gray values or RGB values. On the other hand, the format is lossy, which makes it difficult to use when comparing images because it is impossible to know whether the images are really different from each other or whether minor underlying differences caused the JPEG compression algorithm to make different choices.

When image storage requirements were critical, and scanners and digital cameras were rare, a common format was **GIF (Graphics Interchange Format)**, in which each pixel stored a number from 0 to 255, which was an index into a table of 256 colors. To create a GIF image, one had to decide which 256 colors to use, adjust each pixel to be one of these 256 colors, and then build the array of indices into the table. In images with just a few colors (some corporate logos, diagrams produced with simple drawing and painting tools, icons like arrows, etc.), the GIF format works beautifully; for natural images, it works rather poorly in general. Because one is only allowed 256 colors, the GIF representation is usually lossy.

As mentioned above, TIFF images store multiple channels, each with a description of its contents. For image editing and compositing tools, in which multiple layers of images are blended or laid atop one another, a TIFF image provides an ideal representation for intermediate (or final) results.

The PPM format, which you already encountered in Chapter 15, is very closely related to the organization of image data. In the text-based version of the format, one gives a "magic code" (namely `P3`), and then the width and height of the image (a pair of ASCII representations of positive integers w and h), the maximum color value (an integer no greater than 65,536), and then $3wh$ color values, representing the red, green, and blue components of the image pixels, in left-to-right, top-to-bottom order (so the first $3w$ numbers represent the colors in the top row of the image). Each color value must be no greater than the specified maximum color value, and is stored as an ASCII representation of the value. All values (including the width and height) are separated by whitespace. There's also a binary version of the format (with magic code `P5`) in which the pixel data is stored in a binary representation, and there are also variants for storing grayscale images in text and binary formats.

One particular advantage of PPM is that the *meaning* of each pixel is, to a large degree, specified by the format. To quote the description:

> [Pixel values] are proportional to the intensity of the CIE Rec. 709 red, green, and blue in the pixel, adjusted by the CIE Rec. 709 gamma transfer function. (That transfer function specifies a gamma number of 2.2 and has a linear section for small intensities). A value of Maxval for all three samples represents CIE D65 white and the most intense color in the color universe of which the image is part (the color universe is all the colors in all images to which this image might be compared) [Net09].

The *CIE* referred to in this description is the standards committee for color descriptions, discussed in detail in Chapter 28.

In recent years, the **Portable Network Graphics** or **PNG** format has become popular, in part because of patent issues with the GIF format. It is generally more compact than the naive PPM format, but it is equally easy to use.

For programs that manipulate images, the choice of image format is almost always irrelevant: You almost certainly want to represent an image as an array of double-precision floating-point numbers (or one such array per channel, or perhaps a single three-index array where the third index selects the channel). The reason to favor floating-point representations is that we often perform operations in which adjacent pixel values are averaged; averaging integer or fixed-point values, especially when it's done repeatedly, may result in unacceptable accumulated roundoff errors.

There are two exceptions to the "use floating point" rule.

- If the data associated to each pixel is of a type for which averaging makes no sense (e.g., an object identifier telling which object is visible at that pixel—a so-called **object ID channel**), then it is better to store the value in a form for which arithmetic operations are undefined (such as enumerated types), as a preventive measure against silly programming errors.
- If the pixel data will be used in a search procedure, then a fixed-point representation may make more sense. If, for example, one is going to look through the image for all pixels whose neighborhoods "look like" the neighborhood of a given pixel, integer equality tests may make more sense than floating-point equality tests, which must almost always be implemented as "near-equality" tests (i.e., "Is the difference less than some small value ϵ?").

17.4 Image Compositing

Movie directors often want to film a scene in which actors are in some type of interesting situation (e.g., a remote country, a spaceship, an exploding building, etc.). In many cases, it's not practical to have the actors actually be in these situations (e.g., for insurance reasons it's impossible to arrange for top-paid actors to stand inside exploding buildings). Hollywood uses a technique called **blue screening** (see Figure 17.1) to address this. With this technique, the actors are recorded in a featureless room in which the back wall is of some known color (originally blue, now often green; we'll use green in our description). From the resultant digital images, any pixel that's all green is determined to be part of the background; pixels that have no green are "actor" pixels and those that are a mix of green and some other color are "part actor, part background" pixels. Then the interesting situation (e.g., the exploding building) is also recorded. Finally, the image of the actors is **composited** atop the images of the interesting situation: Every green pixel in the actor image is replaced by the color of the situation-image pixel; every nongreen pixel remains. And the partially green pixels are replaced by a combination of a color extracted from the actor image and the situation image (see Figure 17.2). The resultant composite appears to show the actor in front of the exploding building.

There are some limitations to this approach: The lighting on the actors does not come from the lighting in the situation (or it must be carefully choreographed to approximate it), and things like shadows present real difficulties. Furthermore, at the part actor, part background pixels, we have to estimate the color that's to be associated to the actors, and the fraction of coverage. The result is a **foreground image** and a **mask,** whose pixel values indicate what fraction of the pixel is covered by foreground content: A pixel containing an actor has mask value 1; a pixel

Figure 17.1: An actor, photographed in front of a green screen, is to be composited into a scene. (Jackson Lee/Splash News/Corbis)

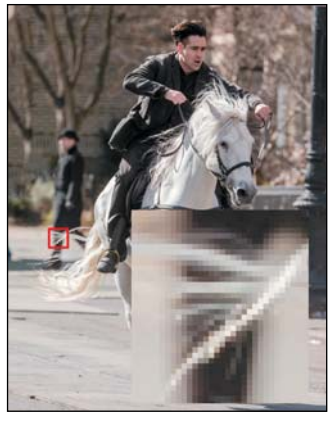


Figure 17.2: The actor, composited atop an outdoor scene. The detail shows how the horse's tail obscures part of the background, while some background shows through. (Jackson Lee/Splash News/Corbis)

showing the background has mask value 0, and a pixel at the edge (e.g., in the actor's hair) has some intermediate value.

In computer graphics, we often perform similar operations: We generate a rendering of some scene, and we want to place other objects (which we also render) into the scene after the fact. Porter and Duff [PD84] gave the first published description of the details of these operations, but they credit the ideas to prior work at the New York Institute of Technology. Fortunately, in computer graphics, as we render these foreground objects we can usually compute the mask value at the same time, rather than having to estimate it; after all, we know the exact geometry of our object and the virtual camera. The mask value in computer graphics is typically denoted by the letter α so that pixels are represented by a 4-tuple (R, G, B, α). Images with pixels of this form are referred to as *RGBA* images and as *RGB*α images.

Porter and Duff [PD84] describe a wide collection of image composition operations; we'll follow their development after first concentrating on the single operation described above: If *U* and *V* are images, the image "*U* over *V*" corresponds to "actor over (i.e., in front of) situation."

17.4.1 The Meaning of a Pixel During Image Compositing

The value α represents the opacity of a single pixel of the image. If we regard the image as being composed of tiny squares, then $\alpha = 0.75$ for some square tells us that the square is three-quarters covered by some object (i.e., 3/4 opaque) but 1/4 uncovered (i.e., 1/4 transparent). Thus, if our rendering is of an object consisting of a single pure-red triangle whose interior covers three-quarters of some pixel, the α-value for that pixel would be 0.75, while the R value would indicate the intensity of the red light from the object, and G and B would be zero.

With a single number, α, we cannot indicate anything more than the opacity; we cannot, for instance, indicate whether it is the left or the right half of the pixel that is most covered, or whether it's covered in a striped or polka-dot pattern. We therefore make the *assumption* that the coverage is uniformly distributed across the pixel: If you picked a point at random in the pixel, the probability that it is opaque rather than transparent is α. We make the further assumption that there is no correlation between these probabilities in the two images; that is, if $\alpha_U = 0.5$ and $\alpha_V = 0.75$, then the probability that a random point is opaque in both images is $0.5 \cdot 0.75 = 0.375$, and the probability that it is transparent in both is 0.125.

The red, green, and blue values represent the intensity of light that *would* arise from the pixel if it were fully opaque, that is, if $\alpha = 1$.

17.4.2 Computing *U* over *V*

Because compositing is performed one pixel at a time, we can illustrate our computation with a single pixel. Figure 17.3 shows an example in which $\alpha_U = 0.4$ and $\alpha_V = 0.3$. The fraction of the image covered by both is $0.4 \cdot 0.3 = 0.12$, while the fraction covered by *V* but not *U* is $0.6 \cdot 0.3 = 0.18$.

To compute *U* over *V*, we must assign both an α-value and a color to the resultant pixel. The coverage, α, will be $0.4 + 0.18$, representing that all the opaque parts of *U* persist in the composite, as do the parts of *V* not obscured by *U*. In more generality, we have

Figure 17.3: (a) A pixel from an image U, 40% covered. Properly speaking, the covered area should be shown scattered randomly about the pixel square. (b) A pixel from the image V, 30% covered. (c) The two pixels, drawn in a single square; the overlap area is 12% of the pixel. (d) The compositing result for U over *V: All of the opaque part of U shows (covering 40% of the pixel), and the nonhidden opaque part of V shows (covering 18% of the pixel).*

$$\alpha = \alpha_U + (1 - \alpha_U)\alpha_V = \alpha_U + \alpha_V - \alpha_U\alpha_V. \tag{17.1}$$

What about the color of the resultant pixel (i.e., the intensity of red, green, and blue light)? Well, the light contributed by the U portion of the pixel (i.e., the fraction α_U containing opaque bits from U) is $\alpha_U \cdot (R_U, G_U, B_U)$, where the subscript indicates that these are the RGB values from the U pixel. The light contributed by the V part of the pixel is $(1 - \alpha_U)\alpha_V \cdot (R_V, G_V, B_V)$. Thus, the total light is

$$\alpha_U \cdot (R_U, G_U, B_U) + (1 - \alpha_U)\alpha_V \cdot (R_V, G_V, B_V), \tag{17.2}$$

while the total opacity is $\alpha = \alpha_U + (1 - \alpha_U)\alpha_V$. If the pixel were totally opaque, the resultant light would be brighter by a factor of α; to avoid this brightness change, we must divide by α, so the RGB values for the pixel are

$$\frac{\alpha_U \cdot (R_U, G_U, B_U) + (1 - \alpha_U)\alpha_V \cdot (R_V, G_V, B_V)}{\alpha_U + (1 - \alpha_U)\alpha_V}. \tag{17.3}$$

These compositing equations tell us how to associate an opacity or coverage value and a color to each pixel of the U over V composite.

17.4.3 Simplifying Compositing

Porter and Duff [PD84] observe that in these equations, the color of U always appears multiplied by α_U, and similarly for V. Thus, if instead of storing the values (R, G, B, α) at each pixel, we stored $(\alpha R, \alpha G, \alpha B, \alpha)$, the computations would simplify. Denoting these by (r, g, b, α) (so that r denotes $R\alpha$, for instance), the compositing equations become

$$\begin{aligned} \alpha &= 1 \cdot \alpha_U + (1 - \alpha_U) \cdot \alpha_V \text{ and} \\ (r, g, b) &= 1 \cdot (r_U, g_U, b_U) + (1 - \alpha_U) \cdot (r_V, g_V, b_V), \end{aligned}$$

where the fraction has disappeared because the new (r, g, b) values must include the premultiplied α-value.

The form of these two equations is identical: The data for U are multiplied by 1, and the data for V are multiplied by $(1 - \alpha_U)$. Calling these F_U and F_V, the "over" compositing rule becomes

$$(r, g, b, \alpha) = F_U \cdot (r_U, g_U, b_U, \alpha_U) + F_V \cdot (r_V, g_V, b_V, \alpha_V). \tag{17.4}$$

17.4.4 Other Compositing Operations

Porter and Duff define other compositing operations as well; almost all have the same form as Equation 17.4, with the values F_U and F_V varying. One can think of each point of the pixel as being in the opaque part of neither U nor V, the opaque part of just U, of just V, or of both. For each, we can think of taking the color from U, from V, or from neither, but to use the color of V on a point where only U is opaque seems nonsensical, and similarly for the points that are transparent in both. Writing a quadruple to describe the chosen color, we have choices like $(0, U, V, U)$ representing U over V and $(0, U, V, 0)$ representing U xor V (i.e., show the part of the image that's in either U or V but not both). Figure 17.4, following Porter and Duff, lists the possible operations, the associated quadruples, and the multipliers F_A and F_B associated to each. The table in the figure omits symmetric operations (i.e., we show U over V, but not V over U).

Finally, there are other compositing operations that do not follow the blending-by-Fs rule. One of these is the **darken** operation, which makes the opaque part of an image darker without changing the coverage:

$$darken(U, s) = (sr_U, sg_U, sb_U, \alpha_U). \tag{17.5}$$

Closely related is the **dissolve** operation, in which the pixel retains its color, but the coverage is gradually reduced:

$$dissolve(U, s) = (sr_U, sg_U, sb_U, s\alpha_U). \tag{17.6}$$

Operation	Quadruple	Diagram	F_U	F_V
Clear	$(0, 0, 0, 0)$		0	0
U	$(0, U, 0, U)$		1	0
U over V	$(0, U, V, U)$		1	$1 - \alpha_U$
U in V	$(0, 0, 0, U)$		α_V	0
U out V	$(0, U, 0, 0)$		$1 - \alpha_V$	0
U atop V	$(0, 0, V, U)$		α_V	$1 - \alpha_U$
U xor V	$(0, U, V, 0)$		$1 - \alpha_V$	$1 - \alpha_U$

Figure 17.4: Compositing operations, and the multipliers for each, to be used with colors premultiplied by α (following Porter and Duff).

Inline Exercise 17.2: Explain why, in the dissolve operation, we had to multiply the "rgb" values by s, even though we were merely altering the opacity of the pixel.

The dissolve operation can be used to create a transition from one image to another:

$$blend(U, V, s) = dissolve(U, (1 - s)) + dissolve(V, s), \tag{17.7}$$

where component-by-component addition is indicated by the $+$ sign, and the parameter s varies from 0 (a pure-U result) to 1 (a pure-V result).

Inline Exercise 17.3: Explain why, if α_U and α_V are both between zero and one, the resultant α-value will be as well so that the resultant pixel is meaningful.

Image operations like these, and their generalizations, are the foundation of image editing programs like Adobe Photoshop [Wik].

17.4.4.1 Problems with Premultiplied Alpha

Suppose you wrote a compositing program that converted ordinary *RGBA* images into premultiplied-α images internally, performed compositing operations, and then wrote out the images after conversion back to unpremultiplied-α images. What would happen if someone used your program to operate on an *RGBA* image in which the α-values were *already* premultiplied? In places where $\alpha = 0$, it would make very little difference; the same goes for $\alpha = 1$. But in partially opaque places, the opacity would be reduced. That would make background objects show through to the foreground more clearly. In practice, this happens fairly frequently; it's a tribute to our visual system's tolerance that it does not tend to confound us.

17.4.5 Physical Units and Compositing

We've spoken about blending light "intensity" using α-values. This has really been a proxy for the idea of blending radiance values (discussed in Chapter 26), which are the values that represent the measurement of light in terms of energy. If, instead, our pixels' red values are simply numbers between 0 and 255 that represent a range from "no red at all" to "as much red as we can represent," then combining them with linear operations is meaningless. Worse still, if they do not correspond to radiance, but to some power of radiance (e.g., its square root), then linear combinations definitely produce the wrong results. Nonetheless, image composition using pixel values directly, whatever they might mean, was done for many years; once again, it's a testament to the visual system's adaptivity that we found the results so convincing. When people began to composite real-world imagery and computer-generated imagery together, however, some problems became apparent; the industry standard is now to do compositing "in linear space," that is, representing color values by something that's a scalar multiple of a physically meaningful and linearly combinable unit [Rob].

As indicated by the description of the PPM image format, pixel values in standard formats are often nonlinearly related to physical values; we'll encounter this again when we discuss gamma correction in Section 28.12.

17.5 Other Image Types

The rectangular array of values that represents an image can contain more than just red, green, and blue values, as we've already seen with images that record opacity (α) as well. What else can be stored in the pixels of an image? Almost anything! A good example is *depth*. There are now cameras that can record a depth image as well as a color image, where the depth value at each pixel represents the distance from the camera to the item shown in the pixel. And during rendering, we typically compute depth values in the course of determining other information about a pixel (such as "What object is visible here?"), so we can get a depth image at no additional cost.

With this additional information, we can consider compositing an actor into a scene in which there are objects between the actor and the camera, and others that are behind the actor. The compositing rule becomes "If the actor pixel is nearer than the scene pixel, composite the actor pixel over the scene; if it's farther away, composite the scene pixel over the actor pixel." But how should we associate a depth value to the new pixel? It's clear that blending depths is not the correct answer. Indeed, for a blended pixel, there's evidently no single correct answer; blending of colors works properly because when we see light of multiple colors, we perceive it as blended. But when we see multiple depths in an area, we don't perceive the area as having a depth that's a blend of these depths. Probably the best solution is to say that when you composite two images that have depths associated to each other, the composite does not have depths, although using the minimum of the two depths is also a fairly safe approach. An alternative is to say that if you want to do multiple composites of depth images, you should do them all at once so that the relative depths are all available during the composition process. Duff [Duf85] addresses these and related questions.

Depths are just one instance of the notion of adding new channels to an image. **Image maps** are often used in web browsers as interface elements: An image is displayed, and a user click on some portion of the image invokes some particular action. For example, an international corporation might display a world map; when you click on your country you are taken to a country-specific website. In an image map, each pixel not only has RGB values, but also has an "action" value (typically a small integer) associated to each pixel. When you click on pixel $(42, 17)$ the action value associated to that pixel is looked up in the image and is used to dispatch an associated action.

Many surfaces created during rendering involve texture maps (see Chapter 20), where every point of the surface has not only x-, y-, and z-coordinates, but also additional texture coordinates, often called u and v. We can make an image in which these u- and v-coordinates are also recorded for each pixel (with some special value for places in the image where there's no object, hence no uv-coordinates).

There are also images that contain, at each pixel, an object identifier telling which object is visible at this pixel; such object IDs are often meaningful only in the context of the program that creates the image, but we often (especially in

expressive rendering) generate images that serve as auxiliary data for creating a final rendering. If we take an object ID image, for instance, and identify points at which the object ID changes and color those black, while coloring other points white, we get a picture of the boundaries between entities in the scene, a kind of condensed representation of the relationships among entities in the image.

17.5.1 Nomenclature

The term "image" is reserved by some for arrays of color or grayscale values; they prefer to call something that contains an object ID at every point a **map** instead, in analogy with things like topographical maps, which contain, at each location, information about the height or roughness of some terrain. Unfortunately, the term "map" is already used in mathematics to mean a (usually continuous and one-to-one) function between two spaces. To the degree that a graphics "map" associates to each point of the plane some value (like an object ID or a transparency), the map is a particular instance of the more general mathematical notion. Further confusion arises when we examine texture mapping, in which we must associate to each point of a surface a pair of texture coordinates, and then use these coordinates to index into some image; the color from the image is used as the color for the surface. Both the image itself and the assignment of texture coordinates to surface points are part of the texture-mapping process. Is the image a texture map? (This usage is common.) Is the assignment of coordinates actually "texture mapping"? (This usage is less common, but it more closely matches the mathematical notion of mapping.) You'll see "map" and "image" used, in the literature, almost interchangeably in many places. Fortunately, the meaning is usually fairly clear from context.

17.6 MIP Maps

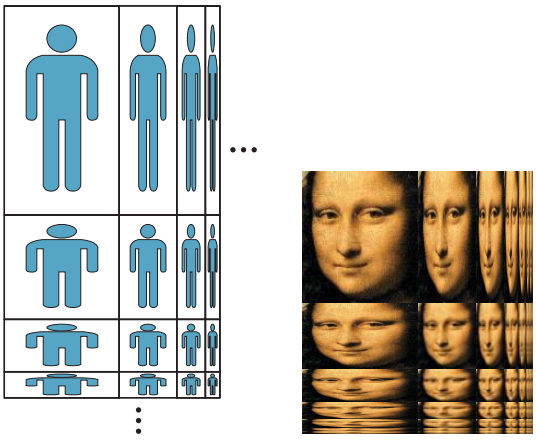
As we'll see when we discuss texture mapping in Chapter 20, it's often important to have multiple representations of the color image that's used in texturing. Lance Williams developed **MIP maps** ("MIP" stands for "multum in parvo," Latin for "many in small") for this very reason. In a MIP map (see Figure 17.5) we store not only an image, but also copies of the image shrunk by varying amounts along the two axes.

The "shrinking" process used to reduce the number of columns by a factor of two is very simple; pairs of adjacent columns are averaged, as shown in Listing 17.1. Analogous code is used to halve the number of rows. The process is repeated, for both rows and columns, until we reach a 1×1 image as shown in Figure 17.5.

Listing 17.1: One stage of column reduction for MIP mapping.

```
1 foreach row of image {
2    for(int c = 0; c < number of columns/2; c++){
3       output[row,c] = (input[row, 2*c] + input[row, 2*c+1])/2;
4    }
5 }
```

In Chapter 19 you'll learn the techniques necessary to analyze the MIP-mapping process and determine its limitations. But for now, let's simply consider the problem of how to MIP-map an image with more than color data. Suppose



(a) (b)

Figure 17.5: (a) A MIP map, schematically. An $n \times k$ image is stored in the upper-left corner; to its right is an $n \times k/2$ version of the image, then an $n \times k/4$ version, etc.; below it is an $n/2 \times k$ image, then an $n/4 \times k$ image etc. The remaining quadrant is filled in with versions of the image that are condensed both in row size and column size. The recursion stops when the image is reduced to a single pixel. (b) A MIP map for a real image.

we have an image, I, storing R-, G-, B-, and α-values (the color values have *not* been premultiplied by α). The recipe for MIP mapping tells us to average adjacent pairs of colors, but is that the right thing to do when α-values are present as well? Suppose, for instance, that we have a red pixel, with opacity 0, next to a blue pixel of opacity 1. Surely the correct "combined" pixel is blue, with opacity .5. In fact, considering the adjacent pixels as contributing to a single "wide pixel," we can suppose that we have a left subpixel with colors (R_L, G_L, B_L) and opacity α_L, and a right subpixel with subscripts "R" on each item. The left subpixel's opacity can contribute at most 50% opacity to the wide pixel; the same goes for the right one. Hence the opacity for the wide pixel should be

$$\alpha = \frac{1}{2}(\alpha_L + \alpha_R). \tag{17.8}$$

What about the color of the wide pixel, though? As the red and blue example shows, opacity must be taken into account in the blending process. In fact, the resultant color should be

$$\frac{\frac{1}{2}(\alpha_L(R_L, G_L, B_L) + \alpha_R(R_R, G_R, B_R))}{\alpha_L + \alpha_R}. \tag{17.9}$$

Thus, we see that even for MIP mapping, it's natural to use premultiplied α-values for the colors. For further detail on the relationship between MIP mapping and α-values, see McGuire and Stone [MS97].

MIP mapping of other image characteristics such as depth or object ID is more problematic; there's no clear correct answer.

17.7 Discussion and Further Reading

While our use of images in graphics is primarily in rendering, images themselves have long been a subject of study in their own right, in the field known as **image**

processing. With the advent of image-based rendering (the synthesis of new views of a scene from one or more photographs or renderings of previous views), certain problems arose, such as "What pixel values should I fill in for the parts of the scene that weren't visible in the previous view, but are in this one?" If it's a matter of just a pixel or two, filling in with colors from neighboring pixels is good enough to fool the eye, but for larger regions, hole filling is a serious (although obviously underdetermined) problem. Problems of hole filling, combining multiple blurred images to create an unblurred image, compositing multiple images when no a priori masks are known, etc., are at the heart of the emerging field of **computational photography.** Other aspects of computational photography are the development of cameras with **coded apertures** (complex masks inside the lens assembly of a camera), and of computational cameras, in which processing built into the camera can adjust the image-acquisition process. Information on Laplace image fill that we provide on this book's website gives just a slight notion of the power of these techniques.

There's no clear dividing line between "images" and "rectangular arrays of values." Organizing graphics-related data in rectangular arrays is powerful because once it's in this form, any kind of per-cell operation can be applied. But there are even more general things that fit into a broad definition of "image," and you should open your mind to further possibilities. For instance, we often store *samples,* many per pixel area, which are then used to compute a pixel value. We'll see this when we discuss rendering, where we often shoot multiple rays near a pixel center and average the results to get a pixel value. These multiple values are, for practical reasons, often taken at fixed locations around the pixel center, making it easy to compare them, but they need not be. It's essential, of course, to record the semantics of the samples, just as we earlier suggested recording the semantics of the pixel values. Images containing these multiple values are generally not meant for display—instead, they provide a spatial organization of information that can be converted to a form useful for display or other reuse of the data.

Arrays of samples which are to be combined into values for display require that the combination process must itself be made explicit. In rendering, the "measurement equation," discussed in Section 29.4.1, makes this explicit.

The notion of coverage, or alpha value, as developed by Porter and Duff, has become nearly universal. At the same time, it has been extended somewhat. Adobe's PDF format [Ado08], for instance, defines for each object both an "opacity" and a "shape" property for each point. A shape value of 0.0 means that the point is outside the object, and 1.0 means it's inside. A value like 0.5 is used to indicate that the point is on the edge of a "soft-edged" object. The product of the shape and opacity values, on the other hand, corresponds to the alpha value we've described in this chapter. These two values can then be used to define quite complex compositing operations.

17.8 Exercises

Exercise 17.1: The blend operation can be described by what happens to a point in a pixel that's in neither the opaque part of U nor the opaque part of V, in just U, in just V, or in both. Give such a description. Is the U-and-V part of the composition consistent with our assumptions about the distribution of the opaque parts of each individual pixel?

Exercise 17.2: Suppose you needed to store images that contained many large regions of constant color. Think of a lossless way to compress such images for more compact storage.

Exercise 17.3: Implement the checkerboard-selection lossy compression scheme described in this chapter; try it on several images and describe the artifacts that you notice in the redisplayed images.

Exercise 17.4: Our description of MIP maps was informal. Suppose that M is a MIP map of some image, I. It's easy to label subparts of M: We let I_{pq} be the subimage that is I, shrunk by 2^p in rows and by 2^q in columns. Thus, the upper-left corner of M, which is a copy of the original image, is I_{00}; the half-as-wide image to its right is I_{01}; the half-as-tall image below I_{00} is I_{10}, etc. If you consider the subimage of I consisting of all parts I_{pq} where $p \geq 1$, it's evidently a MIP map for I_{10}; a similar statement holds for the set of parts where $q \geq 1$: It's the MIP map for I_{01}. Use this idea to formulate a recursive definition of the MIP map of an image I. You may assume that I has a width and height that are powers of two.

Exercise 17.5: MIP mapping is often performed as a preprocess on an image, with the MIP map itself being used many times. The preprocessing cost is therefore relatively unimportant. Nonetheless, for large images, especially those so large that they cannot fit in memory, it can be worth being efficient. Assume that the source image I and the MIP map M are both too large to fit in memory, and that they are stored in row-major order (i.e., that $I[0,0], I[0,1], I[0,2], \ldots$ are adjacent in memory), and that each time you access a new row it incurs a substantial cost, while accessing elements in a single row is relatively inexpensive. How would you generate a MIP map efficiently under these conditions?

Chapter 18

Images and Signal Processing

18.1 Introduction

This chapter introduces the mathematics needed to understand what happens when we perform various operations on images like scaling, rotating, blurring, sharpening, etc., and how to avoid certain unpleasant artifacts when we do these operations. It's a long chapter with lots of mathematics; we've done our best to keep it to a minimum without telling any lies. We begin with a very concise summary of the chapter, and gradually expand on the themes presented there.

The entire chapter can be regarded as an application of the Coordinate-System/Basis principle: Always use a basis that is well suited to your work. In this case, the objects we're working with are not geometric shapes, as in Chapter 2, but images, or more accurately, real-valued functions on a rectangle or a line segment.

18.1.1 A Broad Overview

Even with the goal of minimal mathematics with no lies, it can be difficult to see the forest for the trees, so in this section we present an informal description of the keys ideas of this chapter. *Much of what we say in this section is deliberately wrong.* Usually there's a corresponding true statement, which unfortunately has so many preconditions that it's difficult to see the essential ideas. You should therefore consider this as a high-level guide to the remainder of the chapter.

We'll be looking at the light arriving at one row of an image sensor, because almost all the interesting questions arise when we look at a single row. We'll say that the amount of light arriving at position x is $S(x)$. If we're ray tracing, we might determine the value $S(x)$ by tracing a ray starting at location x. If we're using a real-world camera, the value $S(x)$ is provided by nature. In either case, S is a real-valued function on an interval, and we'll assume it's continuous. So we'll begin by looking at continuous functions on an interval.

We can build a continuous function on $[0, 2\pi]$ by summing up several periodic functions on that interval, as shown in Figure 18.1. Surprisingly, we can also (with the help of some integrals) start with a continuous function f on an interval and break it into a possibly infinite sum of periodic functions, in the reverse of the process shown in Figure 18.1. This decomposition is analogous to breaking a vector in $\mathbf{R}^3$ into three component vectors along the x-, y-, and z-axes. The coefficients of the component periodic functions completely determine f; the coefficients, listed in order of frequency, are called the "Fourier transform" of the function f. So, if $f(x) = 2.1\cos(x) - 3.5\cos(2x) - 8\cos(3x)$, then its Fourier transform, denoted $\hat{f}$, is the function with $\hat{f}(1) = 2.1, \hat{f}(2) = 3.5$, and $\hat{f}(3) = -8$. (Actually, decomposing the function f may involve both cosines *and* sines, but we're going to ignore this.)

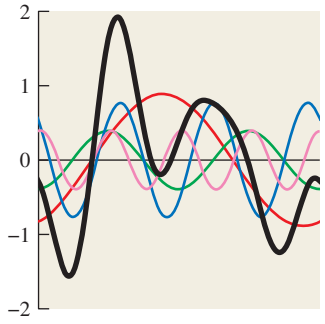


Figure 18.1: Summing up periodic functions (in color) of different frequencies leads to more complicated functions, in this case the black (thickest) one.

The same idea works for functions on the real line: We can take a function $f : \mathbf{R} \to \mathbf{R}$ and compute (with a lot of integration) a different function $\hat{f} : \mathbf{R} \to \mathbf{R}$ with the property that $\hat{f}(\omega)$ tells us "how much f looks like a cosine of frequency ω," just as the x-coordinate of a vector $\mathbf{v}$ in $\mathbf{R}^3$ tells us how much $\mathbf{v}$ "looks like" the unit vector along the x-axis. And if you tell me $\hat{f}$, I can recover f from it. So the $f \iff \hat{f}$ correspondence gives us two different ways to look at any function: The first ("the value representation") tells the value of a function at each point; the second ("the frequency representation") tells "how much it looks like a periodic function of frequency ω" for any ω.

In the course of ray tracing, using one ray per pixel, traced from the pixel center, we're taking the function S, defined on $\mathbf{R}$, and evaluating it at the pixel centers, which we'll assume are the integer points; that is, we're looking at $S(0), S(1), S(2)$, etc. It's quite possible for two different functions S and T to have the same integer samples (see Figure 18.2). That's reason for concern: The samples we're collecting don't uniquely determine the arriving light! Even so, when we display those samples on a (one-dimensional) LCD screen, we get a piecewise constant function (see Figure 18.3) that's not *very* different from either S or T. And anyhow, our eyes tend to smooth out the transitions between adjacent display pixels, making the approximation even better.

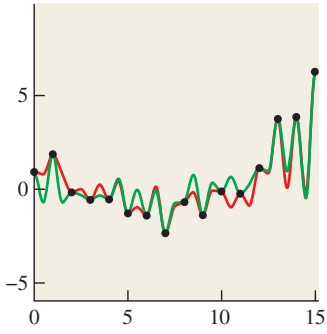


Figure 18.2: The functions S (red) and T (green) have the same values at each integer point (black dots).

How serious a problem is the nonuniqueness of the preceding paragraph? Very. It's what makes sloping straight lines on a black-and-white display look jagged, what makes wagon wheels appear to rotate backward in old movies, what makes scaled-down images look bad, and a whole host of other problems. It's got a name: aliasing. We'll now see where that name came from by looking at the samples of some very simple functions: the periodic functions from which all other functions can be created.

Let's look at the interval $[0, 2\pi]$, and consider ten equally spaced samples of the function $x \mapsto \sin(x)$ on this interval, shown in Figure 18.4. From these samples, it's pretty easy to reconstruct the original sine function. We can come very close to reconstructing it by just "connecting the dots," for instance.

The same thing is true for the samples of $x \mapsto \sin(2x)$ or (barely) $x \mapsto \sin(3x)$. But by the time we look at $x \mapsto \sin(5x)$ (see Figure 18.5), something odd happens: The samples are all zeroes. By looking at only the samples, we can't tell the difference between $\sin(5x)$ and $\sin(0x)$. As Figure 18.6 shows, the same is true for $\sin(1x)$ and $\sin(11x)$.

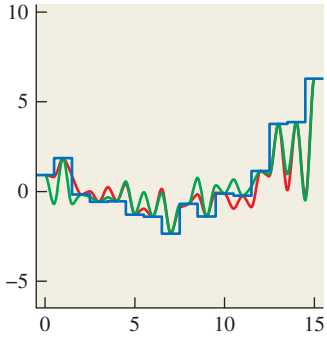


Figure 18.3: The light resulting from displaying samples of either the red or the green function on a 1D LCD screen, plotted in blue.

The means that if our arriving-light function S happened to be $x \mapsto \sin(11x)$, we might think, from looking at the recorded samples, that it was $\sin(x)$ instead: The frequency-11 sine is masquerading as a frequency-1 sinusoid. It's the fact

that the sample values correspond to various different sinusoids that leads to the name "aliasing." In general, if we take $2N$ equispaced samples, then sines of frequency k and $k + 2N$ and $k + 4N$, etc., will all have identical samples. But if we restrict our function S to contain only frequencies strictly between $-N$ and N, then the samples uniquely determine the function. The same idea works for functions defined on $\mathbf{R}$ rather than on an interval: If $\hat{f}(\omega) = 0$ for $\omega \geq \omega_0$, then f can be reconstructed from its values at any infinite sequence of points with spacing π/ω_0.

We can't actually constrain the arriving-light function to not have high frequencies, however. If we photograph a picket fence from far away, the bright pickets against the dark grass can occur with arbitrarily high frequencies. The shorthand description of this situation is that "If your scene has high frequencies in it, then you'll get aliasing when you render it." The solution is to apply various tricks to remove the high frequencies before we take the samples (or in the course of sampling). Doing so in a way that's computationally efficient and yet effective requires the deeper understanding that the rest of this chapter will give you.

Here's one example of a trick to remove high frequencies, just to give you a taste. Consider the $\sin(x)$ versus $\sin(11x)$ example we looked at earlier. We sampled these two functions at certain values of x. Let's say one of them is x_0. What would happen if you took your input signal S and instead of computing $S(x_0)$, you computed $\frac{1}{3}(S(x_0) + S(x_0 + r_1) + S(x_0 - r_2))$, where r_1 and r_2 are small random numbers, on the order of half the sample spacing of $2\pi/10$? If the input signal S were $S(x) = \sin(x)$, the three values you'd average would all be very close to $\sin(x_0)$; that is, the randomization would have little effect (see Figure 18.7). On the other hand, if the original signal was $S(x) = \sin(11x)$, then the three values you'd average would tend to be very different, and their average (see Figure 18.8) would generally be closer to zero than $S(x_0)$. In short, this method tends to *attenuate* high-frequency parts of the input, while retaining low-frequency parts.

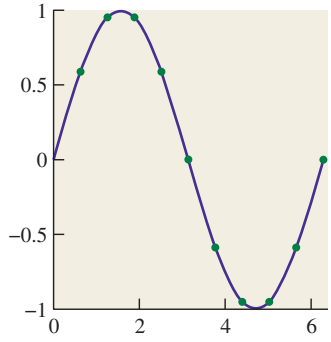


Figure 18.4: Ten evenly spaced samples of $y = \sin(x)$ on the interval $[0, 2\pi]$.

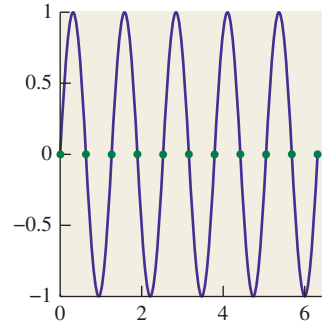


Figure 18.5: The ten samples of $y = \sin(5x)$ all turn out to be zero!

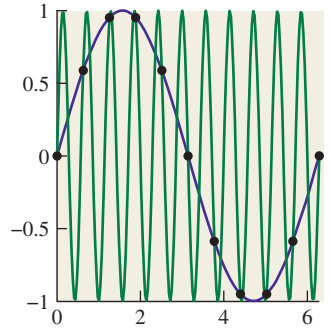


Figure 18.6: The functions $x \mapsto \sin(x)$ and $x \mapsto \sin(11x)$ have identical values at our ten sample points.

18.1.2 Important Terms, Assumptions, and Notation

The main ideas we'll use in this chapter are convolution and the Fourier transform, which you may have encountered in algorithms courses or in the study of various engineering or mathematics problems.

Fortunately, both convolution and Fourier transforms can be well understood in terms of familiar operations in graphics; we motivate the mathematics by showing its connection to graphics. Convolution, for instance, takes place in digital cameras, scanners, and displays. The Fourier transform may be less familiar, although the typical "graphical display" of an audio signal (see Figure 18.9), in which the amounts of bass, midrange, and treble are shown changing over time, shows, at each time, a kind of basic Fourier transform of a brief segment of the audio signal.

For us, the essential property of the Fourier transform is that it turns convolution of functions, which is somewhat messy, into multiplication of other functions, which is easy to understand and visualize.

The Fourier transformation (for functions on the real line) takes a function and represents it in a new basis; thus, this chapter provides yet another instance of the principle that expressing things in the right basis makes them easy to understand.

The same tools we use to understand images in this chapter will prove useful not only in analyzing image operations, however: They also appear in the study of the scattering of light by a surface, which can be interpreted as a kind

of convolution operation [RH04], and in rendering, in which the frequency analysis of light being transported in a scene can yield insights into the nature of computations necessary to accurately simulate that light transport [DHS+05].

Before discussing convolution and other operations, we return to the topic of Section 9.4.2: the principle that you must know the meaning of every number in a graphics program. Before we discuss operations like convolution and Fourier transforms on images, we have to know what the images *mean*. The difficulty, which we discussed briefly in Chapter 17, is that in some cases we just don't know. Alvy Ray Smith made a point of this in a paper titled "A Pixel Is *Not* A Little Square" [Smi95], in which he observes that the individual values in a pixel array do *not* in general represent the average of something over a small square on the image plane, and that algorithms that rely on this model of pixels are bound to fail in some cases. (More simply, he points out that a pixel does not represent a tiny square of constant value, even though the pixel may be displayed that way on an LCD screen!) As an extreme example, an object ID image contains, at each pixel, an identifier that tells which object is visible at that pixel. The pixel values in this case are not even necessarily numerical!

For this chapter only, to avoid messiness introduced by shifting by one-half in the x- and y-directions, we're going to use display screen coordinates in which the display pixel indexed by $(0,0)$ has display coordinates that range from $-\frac{1}{2}$ to $\frac{1}{2}$ in both x and y, and the display pixel named (i,j) is a small square *centered* at (i,j) rather than at $(i+\frac{1}{2}, j+\frac{1}{2})$. This means that pixel (i,j) is at x-location i and y-location j, and *not* in "row i and column j"; that is, we're using geometric indexing rather than image indexing. Because this chapter contains no actual algorithms that depend on display pixel coordinates, this should cause no problems for you.

For *this* chapter, we're going to assume (initially) that images contain physical measurements of light, measured in physical units. For a digital camera, this might be something like the average radiance along a ray hitting a small rectangle on a CCD sensor, or perhaps an integral of that radiance over the area of that rectangle, or the total light energy that arrived at the rectangle while the shutter was open. (Some digital cameras will let you get such information when you store a photo in "raw" mode, and will even tell you when the sensor was oversaturated so that the stored value is a false measurement.) For a rendered image, the value stored at a pixel might be the radiance along a ray that passed through the single point at the center of the image area corresponding to the pixel, or an average of radiances of several rays through the pixel, etc. It might even be an average of samples from a region around the pixel center, where the regions for adjacent pixel centers *overlap*.

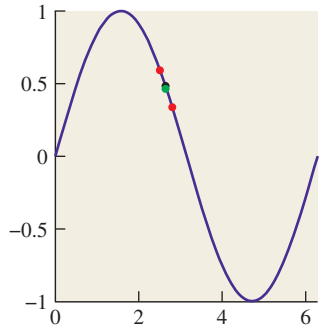


Figure 18.7: The original sample $\sin(x_0)$ in black, with two nearby random samples, shown in red. The average (green) of the three heights is very nearly $\sin(x_0)$.

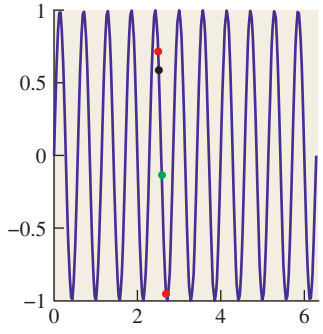


Figure 18.8: Random samples (red) of $x \mapsto \sin(11x)$ near $x = x_0$ are quite different from the sample at x_0 (black), so their average (green) is nearer to zero.

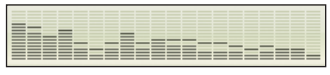


Figure 18.9: The spectrum of an audio signal displayed by several intensity bars.

18.2 Historical Motivation

When graphics researchers first wanted to draw a line on a rectangular grid of pixels, the most obvious thing to do was to write the line in the form $y = mx + b$, and for each integer x-value, compute $y = mx + b$, which was usually *not* an integer, round it off to an integer y', and put a mark at location (x, y'). This only works well if m is between -1 and 1; for greater slopes, it worked better to write $x = my + b$, that is, to swap the roles of x and y, but that's not germane to this discussion. The kind of line produced with this method is shown in Figure 18.10, where we've drawn the line using little squares as pixel marks.

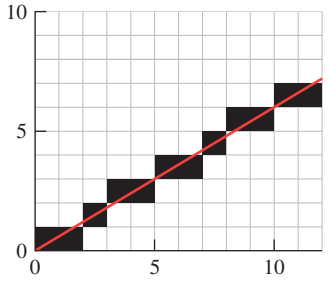


Figure 18.10: A line drawn with one "pixel" per column.

You can see that the line is fairly "jaggy"—it has jagged edges—but it's hard to imagine how to avoid this if your only choice is to draw a black or a white square. Fortunately, display devices improved to display multiple gray levels. To avoid the staircase-like appearance of the jaggy line, we can fill in gray squares at the steps, or be even more sophisticated—we can set the gray-level for each square to be proportional to the amount that square overlaps a unit-width line, as shown in Figure 18.11. The resultant line, viewed close up like this, looks a bit odd. But seen from an appropriate distance (as in Figure 18.12), it looks very good—far better than the jaggy black-and-white line.

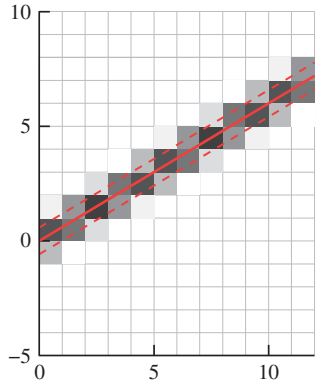


Figure 18.11: Gray-levels proportionally overlap with a unit-width line indicated in red.

The "compute the overlap between the square and the thing you're rendering" approach seems a bit ad hoc, but it also seems to work well in practice. The same idea, applied to text, produces fonts that are visually more appealing than the pure black-and-white fonts that arise when we use the "this pixel is black if the pixel center is within the character" approach (see Figures 18.13 and 18.14).

Merely computing overlaps with pixel squares isn't a cure-all, as you can see by considering a sequence of images generated by a small moving object. Figure 18.15 shows a red triangle moving through three pixels of a "one-dimensional image" in the course of five "frames." In the first and second frames, it's completely in the first square, tinting it pink; in the fourth and fifth, it's completely in the third. In frame three, it's in the middle square. The result is that although the object is moving with uniform speed through the pixels, it appears to "rush" through the middle pixel, which is pink for only half as long as the pixels on either side.

Figure 18.12: A grayscale rendering of a line, unmagnified.

To compensate for the differing amounts of time spent in each square, and thus the irregularity of the apparent motion, we can use a different strategy: Instead of measuring the area of the overlap between the object and the square, we can compute a *weighted* area overlap, counting area overlap near the square's center as more important than area overlap near the edge of the square. While this approach *does* address the irregularity of the pure area-measuring approach, there remains another problem: As a small, dark object moves from left to right, the total brightness of the image varies. When the object is near the dividing line between two squares, neither is darkened much at all; when it's near the middle of a square, that square is darkened substantially. The result is that the object appears to waver in brightness during the animation.

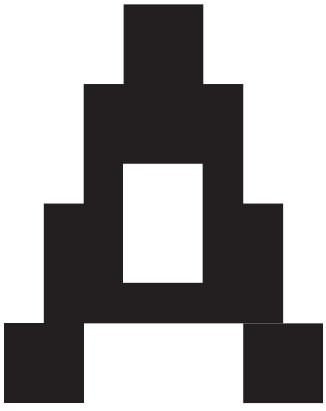


Figure 18.13: A black-and-white rendering of a letter "A".

A line of slope $3/5$, drawn in black-and-white, will contain two pixels in one row, one in the next row, two in the next, and then the pattern will repeat, in a $2, 1, 2, 2, 1, 2, \ldots$ fashion. The irregular "jaggedness" of this line corresponds precisely to the irregular time spent by the moving triangle in each square—the jaggedness of the line and the jerkiness of the motion are different instances of the same aliasing phenomenon.

The somewhat surprising solution is to use a weighting function that says that an object contributes to the brightness of square i if it overlaps anywhere from the center of square $i - 1$ to the center of square $i + 1$. Figure 18.16 shows this in a side view. The object, shown as a small, black line segment, contributes both the left pixel, whose weighting function is shown in blue, and the center pixel, whose weighting function is shown in red, but not the right pixel, whose weighting function is shown in green. The left-pixel contribution is small, because the black

line is near the edge of the "tent," while the center-pixel contribution is larger. Notice that the sum of the weighting functions is the constant function 1, so no matter where we place the object, its total contribution to image brightness will be the same.

We can express the weighted-area-overlap approach to determining pixel values mathematically. Let's suppose that $x \mapsto f(x)$ is the function shown in Figure 18.16 whose value is 1 for any point x within our object, and 0 elsewhere. And let's suppose that the red tent-shaped function in Figure 18.16 is called $x \mapsto g(x)$. Then the value we assign to the center pixel is given by

$$\int_{-\infty}^{\infty} g(x)f(x)\,dx. \tag{18.1}$$

The value assigned to the next pixel to the right, whose weighting function is just g shifted right one unit, is

$$\int_{-\infty}^{\infty} f(x)g(x-1)\,dx. \tag{18.2}$$

A similar expression holds for every pixel: The values we're computing are generated by multiplying f with a shifted version of g and then integrating. This operation—point-by-point multiplication of one function by a shifted version of another, followed by integration (or summation, in some cases)—appears over and over again in both graphics and mathematics and is called **convolution,** although we should warn you that the proper definition includes a negation so that we end up summing things of the form $f(x)h(i-x)$; this negation leads both to convenient mathematical properties and to considerable confusion. Fortunately for us, in almost all our applications the functions that we convolve against have the property that $h(x) = h(-x)$, so the negative sign has no impact.

In the next section, we'll discuss various kinds of convolutions, their applications in graphics, and some of their mathematical properties.

The remainder of this chapter consists of applying ideas from **signal processing** to computer graphics. In that context, functions on the real line (or an interval) are often called **signals** (particularly when the parameter is denoted t so that we can think of $f(t)$ as a value that varies with time). Convolving with a function like the "tent function" above, which is nonzero on just a small region, is called **applying a filter** or **filtering,** although the term can be used for convolution with *any* function.

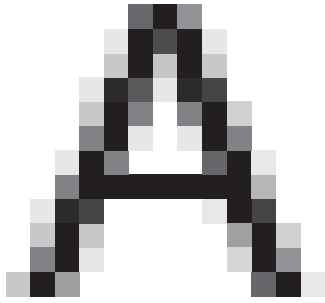


Figure 18.14: A grayscale rendering of a letter "A" (from a different font).

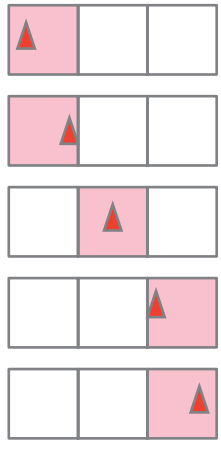


Figure 18.15: A small object moves left to right through a sequence of three pixels.

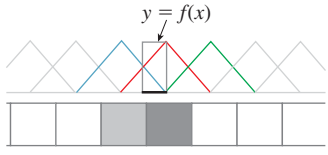


Figure 18.16: Weighted area measurement. The central red "tent-shaped" function is used as a weight for contributions to the center pixel.

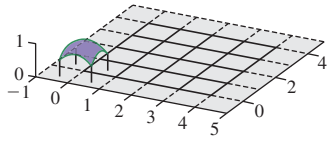


Figure 18.17: Sensor pixels in a digital camera, and the sensitivity function M for pixel $(0,0)$.

18.3 Convolution

As we said already, convolution appears over and over again in graphics and the physical world. Nearly every act of "sensing" involves some sort of convolution, for instance. For example, consider one row of sensor pixels in an idealized digital camera. We'll say that the light energy falling on the sensor at location (x,y) in one second is described as function $S(x,y)$, and that S is independent of time. The camera shutter opens for one second, and each pixel accumulates arriving energy over that period of time, after which an accumulated value is recorded as the pixel's value. But each pixel sensor—say, the one at pixel $(0,0)$—has a responsivity function, $(x,y) \mapsto M(x,y)$, that tells how much pixel response there is for each bit of arriving light (see Figure 18.17). We're being deliberately informal

about units here because it's the *form* of the computation we care about, not the actual values.

To determine the sensor pixel's response to the incoming light, we multiply each bit of light $S(x,y)$ by the responsivity $M(x,y)$, and sum up over the entire pixel:

$$value = \int_{-\frac{1}{2}}^{\frac{1}{2}} \int_{-\frac{1}{2}}^{\frac{1}{2}} S(x,y)M(x,y)\ dy\ dx. \tag{18.3}$$

If we extend the definition of M to the whole plane of the sensor by defining M to be 0 outside the unit square corresponding to pixel $(0,0)$, we can rewrite this as

$$value_{0,0} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} S(x,y)M(x,y)\ dy\ dx, \tag{18.4}$$

which may appear more complicated, but actually will result in simpler formulas elsewhere.

In a well-designed camera, the sensor responsivity should be the same for each pixel. What does this mean mathematically? It means, for instance, that for sensor pixel $(2,3)$, we'll want to multiply $S(x,y)M(x-2,y-3)$ and integrate, that is,

$$value_{2,3} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} S(x,y)M(x-2,y-3)\ dy\ dx, \tag{18.5}$$

and in general, the formula for sensor pixel (i,j) will be

$$value_{i,j} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} S(x,y)M(x-i,y-j)\ dy\ dx. \tag{18.6}$$

This expression has the form of a product of a function S with a shifted function M, integrated; the resultant value is a function of the shift amount, (i,j). That is the essence of a convolution, and indeed, Equation 18.6 is almost the definition of the convolution $S \star M$ of the two functions. Two small adjustments are needed. First, since S and M are both functions on all of $\mathbf{R^2}$, their convolution is defined to be a function on all of $\mathbf{R^2}$. The values we've described above are the *restriction* of that function to the integer grid. Second, it's very convenient to have a definition of convolution that makes $f \star g = g \star f$. For this to work out properly, there needs to be an extra negation; that is, we want Equation 18.6 to have the form

$$value_{i,j} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} S(x,y)\bar{M}(i-x,j-y)\ dy\ dx. \tag{18.7}$$

We can arrange this by defining $\bar{M}(x,y) = M(-x,-y)$. (For a typical sensor, the response function is symmetric, so $\bar{M}$ and M are the same.) This final form is just a 2D analog of the 1D convolution. Simplifying to one dimension, we can now define the **convolution** of two functions $f, g : \mathbf{R} \to \mathbf{R}$:

$$(f \star g)(t) = \int_{-\infty}^{\infty} f(x)g(t-x)\ dx. \tag{18.8}$$

Inline Exercise 18.1: (a) Pause briefly and examine that definition carefully. It's central to much of the remainder of this book.
(b) Perform the substitution $s = t-x, ds = -dx$ in the integral of Equation 18.8 to confirm that $(f \star g)(t) = (g \star f)(t)$.

We can say that image capture by our digital camera consists of convolving the incoming light with the "flipped" sensor response function $\bar{M}$, and then restricting to the integer lattice $\mathbf{Z} \times \mathbf{Z}$.

In almost all cases that we study, one of the functions f or g will be an even function, and hence the negation has no consequence at all. The two-dimensional convolution is defined very similarly. If $f, g : \mathbf{R^2} \to \mathbf{R}$ are two functions on $\mathbf{R^2}$, then

$$(f \star g)(s,t) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x,y)g(s-x,t-y)\, dx\, dy. \tag{18.9}$$

Convolution is also defined for two periodic functions of period P, but with the domain of integration replaced by any interval of length P.

Convolution can also be applied to discrete signals, that is, to a pair of functions $f, g : \mathbf{Z} \mapsto \mathbf{R}$; the definition is almost identical, except for the replacement of the integral with a summation:

$$(f \star g)(i) = \sum_{j=-\infty}^{\infty} f(j)g(i-j), \qquad \text{for } i \in \mathbf{Z}, \tag{18.10}$$

with an analogous definition for functions of two variables. If $f, g : \mathbf{Z} \times \mathbf{Z} \to \mathbf{R}$, then

$$(f \star g)(i,j) = \sum_{k=-\infty}^{\infty} \sum_{p=-\infty}^{\infty} f(k,p)g(i-k,j-p), \qquad \text{for } i,j \in \mathbf{Z}. \tag{18.11}$$

As an application of this kind of convolution, imagine that you have an image that is in very sharp focus, but you want to use it as a background for a composition in which it should appear out of focus, while the foreground objects should be in focus. One way to do this is to replace each pixel with an average of itself and its eight neighbors. Figure 18.18 shows the results on a small example. On a larger image, you might want to "blur" with a much larger block of ones, to achieve any noticeable effect. If we call $f(i,j)$ the value of the original image pixel at (i,j), and let $g(i,j) = 1$ for $-1 \le i,j \le 1$, and 0 otherwise, then the blurred-image pixel at (i,j) is exactly $(f \star g)(i,j)$. Notice, too, that the function g that we used in the blurring has the property that $g(i,j) = g(-i,-j)$, that is, it's symmetric about the origin, hence the negative sign in the definition of convolution is of no consequence.

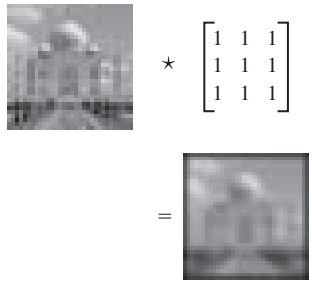


Figure 18.18: A 32×32 *image is convolved with a* 3×3 *block of* 1*s to blur the image.*

The process we've just described is usually called **filtering** f **with the filter** g**,** where the function that's nonzero only on a small region is called the "filter." Because convolution is symmetric, the roles *can* be reversed, however, and we'll have occasion to convolve with "filters" that are nonzero arbitrarily far out on the real line or the integers.

Inline Exercise 18.2: Consider convolving a grayscale image f with a filter g that's defined by $g(-1,-1) = g(-1,0) = g(-1,1) = -1$, $g(1,-1) = g(1,0) = g(1,1) = 1$, and $g(i,j) = 0$ otherwise.
(a) Draw a plot of g.
(b) Describe intuitively where $f \star g$ will be negative, positive, and zero. You might want to start out with some simple examples for f, like an all-gray image, or an image that's white on its bottom half and black on the top, or white on the left half and black on the right, etc. Then generalize.

We've defined convolution for two continuum functions (i.e., functions defined on $\mathbf{R}$) and for two discrete functions (i.e., defined on $\mathbf{Z}$). There's a third class of convolution that comes up in graphics: the discrete-continuum convolution. A familiar instance of this is display on a grayscale LCD monitor. Recall that for this chapter, the display pixel (i,j) is a small box *centered* at (i,j). Figure 18.19 shows the result of displaying a 2×2 image f (shown as a stem plot) with a "box" function b defined on $\mathbf{R}^2$ to produce a piecewise constant function on $\mathbf{R}^2$ representing emitted light intensity.

The emitted light at location (x,y) is given by

$$light(x,y) = f(i,j)box(x-i, y-j). \tag{18.12}$$

This doesn't quite look like a convolution, because there's no summation. But we can insert the summation without changing anything:

$$light(x,y) = \sum_{ij} f(i,j)box(x-i, y-j). \tag{18.13}$$

There's no change because the box function is zero outside the unit box. In the early days of graphics, when CRT displays were common, turning on a single pixel didn't produce a little square of light, it produced a bright spot of light whose intensity faded off gradually with distance. That meant that turning on pixel $(4,7)$ might cause a tiny bit of light to appear even at the area of the display we'd normally associate with coordinates $(12,23)$, for instance, or anywhere else. In that case, the summation in the formula for the light at position (x,y) was essential.

The general definition for the convolution of a discrete function $f : \mathbf{Z} \to \mathbf{R}$ and a continuum function $g : \mathbf{R} \to \mathbf{R}$ is

$$(f \star g)(x) = \sum_{i=-\infty}^{\infty} f(i)g(x-i) \qquad \text{for } x \in \mathbf{R}. \tag{18.14}$$

The result is a continuum function. We leave it to you to define continuous-discrete convolution, and to extend both definitions to the plane.

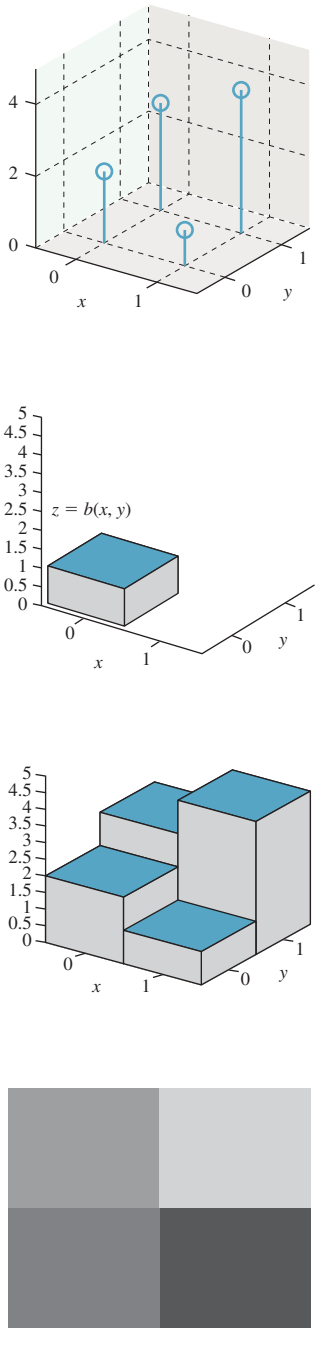


Figure 18.19: The values in a 2×2 grayscale image are convolved with a box function to get a piecewise constant function on a 2×2 square.

18.4 Properties of Convolution

As mentioned in Section 18.2, convolution has several nice mathematical properties. First, for all forms of convolution (discrete, continuous, or mixed) it's linear in each factor, that is,

$$(f_1 + cf_2) \star g = (f_1 \star g) + c(f_2 \star g) \text{ for any } c \in \mathbf{R}, \text{ and} \tag{18.15}$$

$$f \star (g_1 + cg_2) = (f \star g_1) + c(f \star g_2). \tag{18.16}$$

Second, it's commutative, which we'll show for the continuous case, answering the inline problem above:

$$(f \star g)(t) = \int_{-\infty}^{\infty} f(x)g(t-x)\,dx. \tag{18.17}$$

Substituting $s = t - x, ds = -dx$, and $x = t - s$, we get

$$\begin{aligned}(f \star g)(t) &= \int_{-\infty}^{\infty} f(x)g(t-x)\,dx && (18.18)\\ &= \int_{s=\infty}^{-\infty} f(t-s)g(s)\,(-ds) && (18.19)\\ &= \int_{s=-\infty}^{\infty} g(s)f(t-s)\,ds && (18.20)\\ &= (g \star f)(t). && (18.21)\end{aligned}$$

The proofs for the discrete and mixed cases are very similar.

Third, convolution is associative. The proof, which we omit, involves multiple substitutions.

Finally, continuous-continuous convolution has some special properties involving derivatives, such as $f' \star g = f \star g'$ (under some fairly weak assumptions). It also generally increases smoothness: If f is continuous and g is piecewise continuous, then $f \star g$ is differentiable; similarly, if f is once differentiable, then $f \star g$ is twice differentiable. In general, if f is p-times differentiable and g is k-times differentiable, then $f \star g$ is $(p + k + 1)$-times differentiable (again under some fairly weak assumptions).

Alas, for a fixed function f, the map $g \mapsto f \star g$ is usually not invertible—you can't usually "unconvolve." We'll see why when we examine the Fourier transform shortly.

18.5 Convolution-like Computations

Convolution appears in other places as well. Consider the multiplication of 1231 by 1111:

```
   1231
  x1111
  -----
   1231
  1231
 1231
1231
-------
1367641
```

In computing this product, we're taking four shifted copies of the number 1231, each multiplied by a different 1 from the second factor, summing them; this is essentially a convolution operation.

As another example, consider how a square occluder, held above a flat table, casts a shadow when illuminated by a round light source (see Figure 18.20). The brightness at a point P is determined by how much of the light source is visible

Figure 18.20: The square occluder casts a shadow with both umbra and penumbra when illuminated by a round light source.

from P. We can think of this by imagining the square is lit from each single point of the light source, casting a hard shadow on the surface, a shadow whose appearance is essentially a translated copy of the function f that's one for points in the square and zero elsewhere (see Figure 18.21). We sum up these infinitely many hard-shadow pictures, with the result being a soft shadow cast by the lamp. This has the form of a convolution (a sum of many displaced copies of the same function). We can also consider the dual: Imagine each tiny bit of the rectangle individually obstructing the lamp's light from reaching the table. The occlusion due to each tiny bit of rectangle is a disk of "reduced light"; when we sum up all these circular reductions, some table points are in all of them (the **umbra**), some table points are in just a few of the disks (the **penumbra**), and the remainder, the fully lit points, are visible to every point of the lamp. These two ways of considering the illumination arriving at the table—multiple displaced rectangles summed up, or multiple displaced disks summed up—correspond to thinking of $f \star g$ as many displaced copies of f, weighted by values of g, or as many displaced copies of g, weighted by values of f.

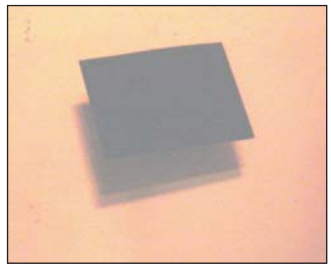


Figure 18.21: The square shadows cast by the occluder when it's illuminated from two different points of the light source (image lightened to better show shadows).

18.6 Reconstruction

Reconstruction is the process of recovering a signal, or an approximation of it, from its samples. If you examine Figure 18.4, for instance, you can see that by connecting the red dots, we could produce a pretty good approximation of the original blue curve. This is called **piecewise linear reconstruction** and it works well for signals that don't contain lots of high frequencies, as we'll see shortly.

We discussed earlier how the conversion of the light arriving at every point of a camera sensor into a discrete set of pixel values is modeled by a convolution, and how, if we display the image on an LCD screen, setting each LCD pixel's brightness to the value stored in the image, we're performing a discrete-continuous convolution, the discrete factor being the image and the continuous factor being a function that's 1 at every point of a unit-width box centered on $(0, 0)$ and 0 everywhere else. This second discrete-continuous convolution is another example of reconstruction, sometimes called **sample and hold** reconstruction.

The "take a photo, then display it on a screen" sequence (i.e., the sample-and-reconstruct sequence) is thus described by a sequence of convolutions. If taking a photo and displaying were completely "faithful," the displayed intensity at each point would be exactly the same as the arriving intensity at the corresponding point of the sensor. But since the displayed intensity is piecewise constant, the only time this can happen is when the original lightfield is also piecewise constant (if, for instance, we were photographing a chessboard so that each square of the chessboard exactly matched one sensor pixel). In general, however, there's no hope that sense-then-redisplay will ever produce the exact same pattern of light that arrived at the sensor. The best we can hope for is that the displayed lightfield is a reasonable approximation of the original. Since displays have limited dynamic range, however, there are practical limitations: You cannot display a photo of the sun and expect the displayed result to burn your retina.

18.7 Function Classes

There are several kinds of functions we'll need to discuss in the next few sections. The first is the one used to mathematically model things like light arriving at an image plane, which is a continuous function of position. We can treat such a

function as defined only on the image rectangle, R, or as being defined on the whole plane (which we'll treat as $\mathbf{R^2}$). In either case, we require that the integral of the square of f is finite,[1] that is,

$$\int_D f(x)^2 \, dx < \infty, \tag{18.22}$$

where D is the domain on which the function is defined. (Functions satisfying this inequality are called **square integrable;** the interpretation, for many physically meaningful functions, is that they represent signals of finite total energy.) The domain D might be the rectangle R, the whole plane $\mathbf{R^2}$, the real line $\mathbf{R}$, or some interval $[a, b]$ when we're discussing the one-dimensional situation. Functions that are square integrable form a vector space called L^2, where we often write something like $L^2(\mathbf{R^2})$ to indicate square-integrable functions on the plane. We say "f is L^2" as shorthand for "f is square integrable." The set of L^2 functions on any particular domain is generally a vector space. It takes a little work to show that L^2 is closed under addition, that is if f and g are L^2, then so is $f + g$; but we'll omit the proof, since it's not particularly instructive.

A function $x \mapsto f(x)$ in $L^2(\mathbf{R})$ must "fall off" as $x \to \pm\infty$, because if $|f(x)|$ is always greater than some constant $M > 0$, then $\int_{-K}^{K} f(x)^2 \, dx > \int_{-K}^{K} M^2 \, dx = 2KM^2$, which goes to infinity as $K \to \infty$.

The next class of functions is the discrete analog of L^2: the set of all functions $f : \mathbf{Z} \to \mathbf{R}$ such that

$$\sum_i f(i)^2 < \infty \tag{18.23}$$

is denoted ℓ^2; these are called **square summable.**

There are two ways in which ℓ^2 functions arise. The first is through sampling of L^2 functions. Sampling is formally defined in the next section, but for now note that if f is a *continuous* L^2 function on $\mathbf{R}$, then the samples of f are just the values $f(i)$ where i is an integer, so sampling in this case amounts to restricting the domain from $\mathbf{R}$ to $\mathbf{Z}$. The second way that ℓ^2 functions arise is as the Fourier transform of functions in $L^2([a, b])$ for an interval $[a, b]$, which we'll describe presently.

Finally, both ℓ^2 and L^2 have inner products. For $\ell^2(\mathbf{Z})$ we define

$$\langle a, b \rangle = \sum_{i=-\infty}^{\infty} a(i)b(i), \tag{18.24}$$

which is analogous to the definition in $\mathbf{R^3}$ of $\mathbf{v} \cdot \mathbf{w} = \sum_{i=1}^{3} v_i w_i$.

For $L^2(D)$, where D is either a finite interval or the real line, we define

$$\langle f, g \rangle = \int_D f(x)g(x) \, dx. \tag{18.25}$$

This inner product on L^2 has all the properties you might expect: It's linear in each factor, and $\langle f, f \rangle = 0$ if and only if $f = 0$, at least if we extend the notion of $f = 0$ to mean that f is zero "almost everywhere," in the sense that if we picked a

1. Later we'll consider complex-valued functions rather than real-valued ones. When we do so, we have to replace $f(x)$ with $|f(x)|$ in the integral.

random number t in the domain of f, then with probability 1, $f(t) = 0$. (In general, when we talk about L^2, we say that two functions are equal if they're equal almost everywhere.)

These inner-product definitions in turn let us define a notion of "length," by defining $\|f\| = \sqrt{\langle f,f\rangle}$, for $f \in L^2$, and similarly for ℓ^2. See Exercise 18.1 for further details.

18.8 Sampling

The term **sampling** is much used in graphics, with multiple meanings. Sometimes it refers to choosing multiple random points P_i $(i = 1, 2, \ldots, n)$ in the domain of a function f so that we can estimate the average value of f on that domain as the average of the values $f(P_i)$ (see Chapter 30). Sometimes (as in the previous edition of this book) it's used to mean "generating pixel values by some kind of unweighted or weighted averaging of a function on a domain," the discrete nature of the pixel array being the motivation for the word "sampling." In this chapter, we'll use it in one very specific way. If f is a *continuous* function on the real line, then *sampling* f means "restricting the domain of f to the integers," or, more generally, to any infinite set of equally spaced points (e.g., the even integers, or all points of the form $0.3 + n/2$, for $n \in \mathbf{Z}$).

For discontinuous functions, the definition is slightly subtler; for those who'd rather ignore the details, it's sufficient to say that if f is piecewise continuous, but has a jump discontinuity at the point x, then the sample of f at x is the average of the left and right limits of f at x. Thus, for a square wave (see Figures 18.22 and 18.23) that alternates between -1 and 1, the sample at any discontinuity is 0.

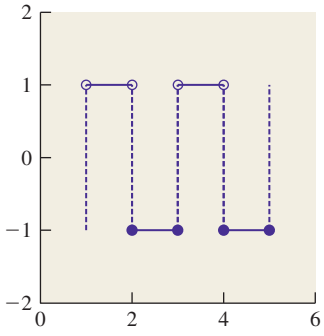


Figure 18.22: A "biased" square wave; at integer points the values are -1.

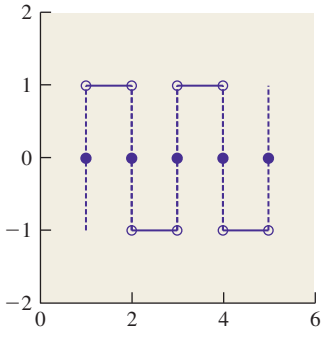


Figure 18.23: The samples of the biased square wave at integer points are all 0.

The more general notion of sampling is motivated by the physical act of **measurement.** If we think of the variable in the function $t \mapsto f(t)$ as time, then to *measure* f we must average its values over some nonzero period of time. If f is rapidly varying, then the shorter the period, the better the measurement. To define the sample of f at a particular time t_0, we therefore mimic this measurement process. First, we consider points $t_0 - a$ and $t_0 + a$, and define a function $\chi_{t_0,a} : \mathbf{R} \to \mathbf{R}$ where $\chi_{t_0,a} = 1$ if $t_0 - a \le t \le t_0 + a$ and 0 otherwise (see Figure 18.24). The function $\chi_{t_0,a}$ serves the role of the shutter in a camera: When we multiply f by $\chi_{t_0,a}$, the values of f are "let through" only on the interval $[t_0 - a, t_0 + a]$. Next, we let

$$U(a) = \frac{1}{2a} \int_{\mathbf{R}} f(t)\chi_{t_0,a}(t)\, dt. \tag{18.26}$$

$U(a)$ is the "measurement" of f in the interval $[t_0 - a, t_0 + a]$, in the sense that it's the average value of f on that interval. Problem 18.2 relates this to convolution. Finally, we define the **sample of** f **at** t_0 to be

$$\lim_{a \to 0} U(a), \tag{18.27}$$

that is, the limiting result of measuring f over shorter and shorter intervals. For a continuous function f, if a is small enough, then $f(s)$ will be very close to $f(t_0)$ for any $s \in [t_0 - a, t_0 + a]$, and the limit of $U(a)$ is just $f(t_0)$—the sample, defined by this measurement process, is exactly the *value* of f at t_0 as we said above; the full proof depends on the mean value theorem for integrals. But for a discontinuous

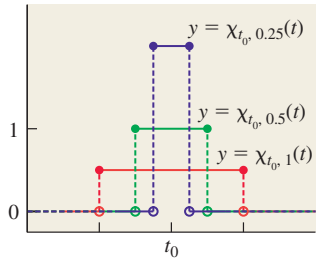


Figure 18.24: The function $\chi_{t_0,a}$ *is nonzero only on the interval* $[t_0 - a, t_0 + a]$.

function like the square wave we saw above, the measurement process averages values to the left and right of t_0, and the limit (in the case of a square wave) is the average of the upper and lower values. In cases messier than these simple ones, it can happen that the limit in Equation 18.27 can fail to exist, in which case the sample of f is not defined. We'll never encounter such functions in practice, though.

18.9 Mathematical Considerations

The somewhat complex definition of sampling suggests that the mathematical details of L^2 functions can be quite messy. That's true, and making precise statements about sampling, convolution, Fourier transforms, etc., is rather difficult. Often the statement of the preconditions is so elaborate that it's quite difficult to understand what a theorem is really saying. Rather than ignoring the preconditions or precision, which leads to statements that seem like nonsense except to the very experienced, and rather than providing the exact statements of every result, we'll restrict our attention, to the degree possible, to "nice" functions (see Figure 18.25) that are either continuous, or very nearly continuous, in the sense that they have a discrete set of discontinuities, and on either side of a discontinuity they are continuous and bounded (i.e., there's no "asymptotic behavior" like that of the graph of $y = 1/x$ near $x = 0$). Figure 18.26 shows some functions that are not nice enough to study in this informal fashion, but which also don't arise in practice.

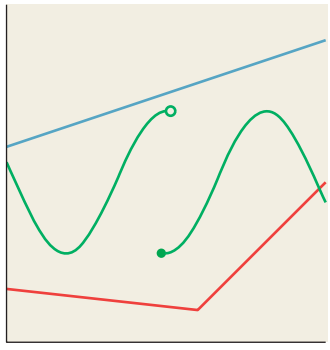


Figure 18.25: A few "nice" functions.

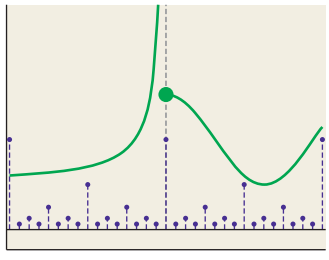


Figure 18.26: "Not-nice" functions. The blue function is 0 except at points of the form $p/2^q$, where its value is $1/2^q$.

The restriction to "nice" functions is enough to make most of the subtleties disappear. It's also appropriate in the sense that we're using these mathematical tools to discuss physical things, like the light arriving at a sensor. At some level, that light intensity is discontinuous—either another photon arrives or it doesn't—but at the level at which we're hoping to model the world, it's reasonable to treat it as a continuous function of time and position.

We're also typically studying the result of *convolving* the "arriving light" function with some other function like a box; the box is a nice enough function that convolving with it always yields a continuous function, and sampling this *continuous* function is easy—we just evaluate at the sample points. So the way we work with light tends to mean that we're working with functions that "behave nicely" when we look closely at the mathematics.

For the remainder of this chapter, we'll be considering things in one dimension: Our "images" will consist of just a row of pixels; our sensor will be a line segment rather than a rectangle, etc. We'll sometimes *illustrate* the corresponding ideas in two dimensions (i.e., on ordinary images), but we'll write the mathematics in one dimension only. So we'll be working with a function f defined on the real line, and its samples defined on the integers. To make things simpler, we'll restrict our attention to the case where f is an **even function** (see Figure 18.27), where $f(x) = f(-x)$ for every x. Examples of even functions are the cosine, and the squaring function. Restricting to even functions lets us mostly avoid using complex numbers, while losing none of the essential ideas.

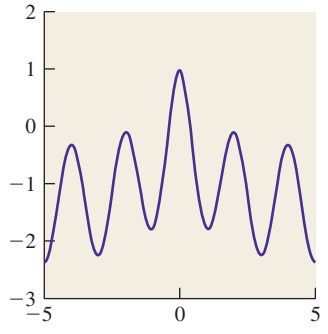


Figure 18.27: An even function is one symmetric about the y-axis.

To review what we've done so far, we've defined convolution, and observed that many operations like display of images, capturing images by sensing (i.e., photography or rendering), blurring or sharpening of images, etc., can be written in terms of convolution, and that sampling at a point is defined by a limit of integrals, while sampling at all points is a limit of convolutions.

For the remainder of the chapter, we're motivated by the question, "Suppose we have incoming light arriving at a sensor, and we want to make an image that best captures this arriving light for subsequent display or other uses; what should we store in the image array?" To answer this question, we need to do two things.

- Choose a new basis in which to represent images.
- Understand how convolution "looks" in this new basis.

The Fourier transform is how we'll transform images into the new basis. And in the new basis, convolution of functions becomes multiplication of functions, which is much easier to understand and reason about.

18.9.1 Frequency-Based Synthesis and Analysis

Consider the interval $H = (-\frac{1}{2}, \frac{1}{2}]$, which we'll use throughout this chapter; the letter "H" is mnemonic for "half." By writing a sum like

$$f(x) = \cos(2\pi x) + \frac{1}{3}\cos(6\pi x), \tag{18.28}$$

shown in Figure 18.28, we can produce an even function on that interval (i.e., one symmetric about the y-axis). In general, any sum of cosines of various integer frequencies will be an even function, because each component cosine is even. By changing how much of each frequency of cosine we mix in, we can get many different functions.

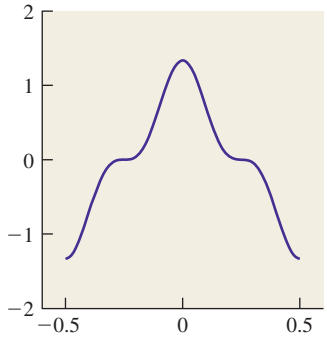


Figure 18.28: An even function on the interval H.

We can, for instance, find a combination of $\cos(0x)$, $\cos(2\pi x)$, and $\cos(4\pi x)$ that satisfies $f(0) = 1$, $f(1/6) = 0$, and $f(\frac{1}{2}) = 0$. These constraints are shown in Figure 18.29; the shaded constraints on the left are there because the function is even, so its values on the left half of the real line must match those on the right half of the line.

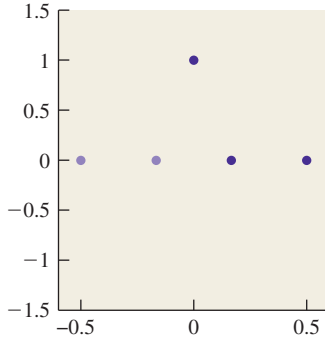


Figure 18.29: The problem.

We write

$$f(x) = a\cos(0x) + b\cos(2\pi x) + c\cos(4\pi x), \tag{18.29}$$

and then plugging in the constraints, we find that

$$\begin{aligned} 1 = f(0) &= a + b + c, && (18.30)\\ 0 = f(1/6) &= a + b\cos(\pi/3) + c\cos(2\pi/3) && (18.31)\\ &= a + b/2 - c/2, \text{ and} && (18.32)\\ 0 = f\left(\frac{1}{2}\right) &= a - b + c, && (18.33) \end{aligned}$$

from which we can determine that $a = 0$ and $b = c = 1/2$ (see Figure 18.30).

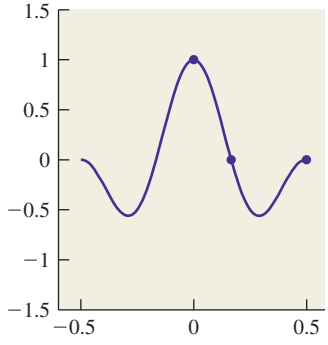


Figure 18.30: The solution.

This is easy to generalize: If we're given k constraints on the values of a function on the non-negative part of the interval I, then we can find a function written as a linear combination of $\cos(0x), \cos(2\pi x), \ldots, \cos(2\pi(k-1)x)$ that satisfies those constraints. The proof relies on elementary properties of the cosine and sine.

Thus, we can "synthesize" various even functions by summing up cosines of many frequencies. We can even "direct" our synthesized function to have certain values at certain points, as in the second example above. We can synthesize *odd* functions by summing up sines of various frequencies as well, and by mixing sines and cosines, we can even synthesize functions that are neither even nor odd.

Whatever function we synthesize, however, will be periodic of period 1, because each term in the sum is periodic with that period. Thus, if f is a sum of sines and cosines of different integer frequencies, we'll have $f(\frac{1}{2}) = f(-\frac{1}{2})$.

> We're using the term "frequency" here quite specifically: We say that $x \mapsto \cos(2\pi x)$ is a function of frequency 1. Some other texts say that $x \mapsto \cos(x)$ is a function of frequency 1. In the same way, some books prefer to define the Fourier transform on the interval $[-1, 1]$, or $[0, 1]$, or $[0, 2\pi]$; depending on the interval, this introduces a multiplicative constant in the definition of the inner product. We're following the convention of Dym and McKean [DM85] so that the interested reader may refer there for proofs, but there is no universal standard. Fortunately for us, we'll mostly be concerned with *qualitative* properties of the Fourier transform, for which the interval of definition and multiplicative constants are not important.

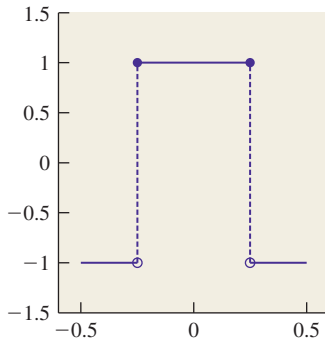


Figure 18.31: A "square wave" function.

More surprising, perhaps, is that *any* even continuous function f on the interval H that satisfies $f(\frac{1}{2}) = f(-\frac{1}{2})$ can be written as a sum of cosines of various integer frequencies. Even *discontinuous* functions can be *almost* written as such a sum. For instance, the square-wave function (see Figure 18.31) defined by

$$f(x) = \begin{cases} 1 & -\frac{1}{4} \le x \le \frac{1}{4} \\ -1 & \text{otherwise} \end{cases} \tag{18.34}$$

can be almost expressed by the infinite sum

$$\bar{f}(x) = \frac{4}{\pi}\left(\cos(2\pi x) - \frac{\cos(6\pi x)}{3} + \frac{\cos(10\pi x)}{5} - \dots\right) \tag{18.35}$$

$$= \sum_{k=0}^{\infty} \frac{4}{\pi(2k+1)}(-1)^k \cos(2\pi(2k+1)x). \tag{18.36}$$

We say "almost expressed" because $\bar{f}(\pm\frac{1}{4}) = 0$—the average of the values of f to the left and right of $\pm\frac{1}{4}$—rather than being equal to $f(\pm\frac{1}{4}) = 1$.

The sequence in Equation 18.35 can be approximated by taking a finite number of terms; a few of those approximations are shown in Figure 18.32.

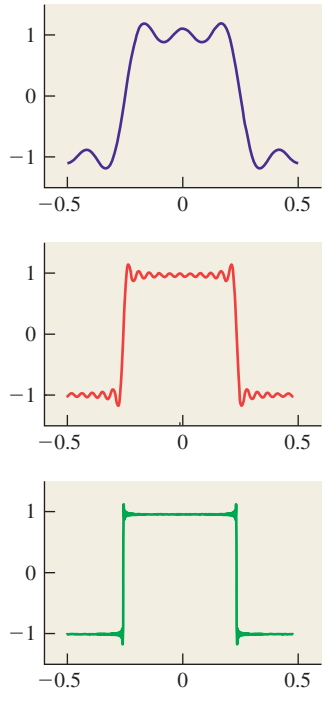


Figure 18.32: The square wave approximated by 2, 10, *and* 100 *terms. The slight overshoot in the approximations is called* ***ringing.***

As a more graphically oriented example, let's take one row of pixels from a symmetric image like that shown in Figure 18.33. We've actually taken half a row and flipped it over to get a perfectly symmetric line of 3,144 pixels, shown in Figure 18.34.

If we write this function as a sum of cosines, the sum will have 3,144 terms, which is hard to read. It starts out as

$$f(x) = 129.28\cos(0x) + 5.67\cos(2\pi x) - 2.35\cos(4\pi x) + \dots. \tag{18.37}$$

We can make an abstract *picture* of this summation by plotting the *coefficients*: At 0 we plot 129.28, at 1 we plot 5.67, at 2 we plot −2.34, etc. The result is shown in Figure 18.35, except that since the coefficient of $\cos(0x)$ is actually 141.8, we've adjusted the y-axis so that you can see the other details, thus hiding the large coefficient for the $\cos(0x)$ term. (That coefficient is just the average of all the pixel values.)

Notice that as the frequencies get higher, the coefficients get smaller. In fact, in natural images, this "falling off with higher frequencies" is commonplace. Notice too that the function shown in Figure 18.34 has lots of variation at a very small scale, while the one shown in Figure 18.28 has variation only at a very large scale. If we made a plot for Figure 18.28 analogous to Figure 18.35, it would have only two nonzero values (at frequencies 0 and 1). In general, details at small scales mean there must be high frequencies in the image, just as we observed in Section 18.11. Since $x \mapsto \cos(2\pi kx)$ has "features" at a scale of $\frac{1}{2k}$, in general any sum that stops at the $\cos(2\pi kx)$ term will have no features smaller than $\frac{1}{2k}$. This kind of relationship between the pattern of coefficients and the appearance of the pixel-value plot is a powerful reasoning tool. We'll next formalize it using the Fourier transform.

But first, it's useful to understand how the frequency decomposition of an image actually *looks,* so Figure 18.36 shows these to you, for a grayscale version of the Taj Mahal image (again, symmetrized). The frequency-0 part of the image is the average grayscale value; we've actually included this in both the middle- and high-frequency images to prevent having to use values less than 0.

Figure 18.33: The Taj Mahal. Original image by Jbarta, at http://upload.wikimedia.org/wikipedia/commons/b/bd/Taj_Mahal,_Agra,_India_edit3.jpg.

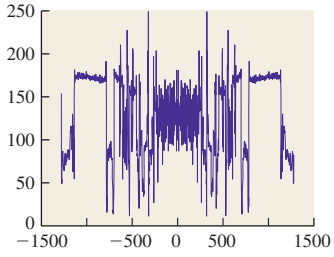


Figure 18.34: The grayscale pixel values for one horizontal line of the image.

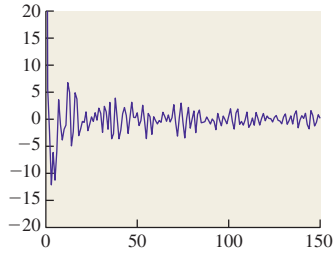


Figure 18.35: The first few coefficients for the sum representing the row of Taj Mahal pixels. We plot the coefficient of $\cos(2\pi kx)$ *at position k.*

18.10 The Fourier Transform: Definitions

In the next two sections we'll define several different Fourier transforms, all of them closely related. We'll only hint at the proofs of various claims, and instead rely mostly on suggestive examples. As motivation for you as you read these sections, here are the three main features of the Fourier transform that we'll use in applications to computer graphics.

- The Fourier transform turns convolution into multiplication, and vice versa. If we write $\mathcal{F}$ for the Fourier transform, this means that

$$\mathcal{F}(f \star g) = \mathcal{F}(f)\mathcal{F}(g) \text{ and} \tag{18.38}$$

$$\mathcal{F}(fg) = \mathcal{F}(f) \star \mathcal{F}(g). \tag{18.39}$$

- If we define a function g like the one shown in Figure 18.37, with peaks that are equally spaced and very narrow, then the Fourier transform of g looks rather like g itself, except that the closer the spacing of the peaks in g, the wider the spacing of the peaks in the transform.
- Multiplying a function f by a function like g approximates "sampling f at equispaced points." Thus, functions like g can be used to study the effects of sampling. Because of the convolution-multiplication duality, we'll see that the sampled function has a Fourier transform that's a sum of translated replicates of the original function's transform.

18.11 The Fourier Transform of a Function on an Interval

We hinted in Section 18.9.1 that if we had an even function in $L^2(H)$, then its Fourier transform was the set of coefficients used to write the function as a sum of cosines. In general, however, the Fourier transform is defined for *any* L^2 function,

not just even ones. The most basic definition is a little messy. For each integer $k \geq 0$, define

$$a_k = \int_{-\frac{1}{2}}^{\frac{1}{2}} f(x)\cos(kx)\,dx \text{ and} \tag{18.40}$$

$$b_k = \int_{-\frac{1}{2}}^{\frac{1}{2}} f(x)\sin(-kx)\,dx. \tag{18.41}$$

Notice that b_0 is always 0.

The sequences $\{a_k\}$ and $\{b_k\}$ are called the Fourier transform of f. If f is continuous and $f(\frac{1}{2}) = f(-\frac{1}{2})$, then it turns out that

$$f(x) = \sum_{k=-\infty}^{\infty} a_k\cos(kx) + b_k\sin(kx). \tag{18.42}$$

Surprisingly, the annoyance of having an unnecessary value (b_0), the vagueness of "the Fourier transform consists of two sequences," and the somewhat surprising appearance of the negative sign in the definition of b_k can all be resolved by generalizing to complex numbers.

Instead of real-valued functions $f : [-\frac{1}{2}, \frac{1}{2}] \to \mathbf{R}$, we'll consider complex-valued functions. And instead of considering the sine and cosine separately, we'll define

$$e_k(x) = \cos(2\pi kx) + \mathbf{i}\sin(2\pi kx) = e^{2\pi \mathbf{i} kx}. \tag{18.43}$$

Inline Exercise 18.3: Show that $(e_k(x)+e_{-k}(x))/2 = \cos(2\pi kx)$, and $(e_k(x)-e_{-k}(x))/(2\mathbf{i}) = \sin(2\pi kx)$, so that any function written as a sum of sines and cosines can also be written as a sum of e_ks, and vice versa.

The only other change is that the definition of the inner product must be slightly modified to

$$\langle f, g\rangle = \int f(x)\overline{g(x)}\,dx, \tag{18.44}$$

where $\overline{a+b\mathbf{i}} = a - b\mathbf{i}$ is the **complex conjugate.** Making this change ensures that the inner product of f with f is always a non-negative real number so that its square root can be used to define the length $\|f\|$.

With this inner product, the set of functions $\{e_k : k \in \mathbf{Z}\}$ is *orthonormal,* that is,

$$\langle e_k, e_j\rangle = \begin{cases} 0 & j \neq k \\ 1 & j = k \end{cases}; \tag{18.45}$$

the proof is an exercise in calculus and trigonometric identities.

We define

$$c_k = \int_{-\frac{1}{2}}^{\frac{1}{2}} f(x)\overline{e_k(x)}\,dx = \langle f, e_k\rangle. \tag{18.46}$$

The TajMahal image

Frequencies less than 15 (low frequency)

Frequencies 15 to 70 (middle frequency)

Frequencies greater than 70 (high frequency)

Figure 18.36: The low-, middle-, and high-frequency components of the Taj Mahal image.

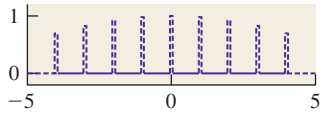


Figure 18.37: A function with equally spaced peaks that fade off as $x \to \pm\infty$.

It then turns out that for a continuous L^2 function f satisfying $f(\frac{1}{2}) = f(-\frac{1}{2})$,

$$f(x) = \sum_k c_k e_k(x), \tag{18.47}$$

that is, computing the inner product of f with each basis element e_k lets us write f as a linear combination of the e_k's. This is exactly analogous to the situation in $\mathbf{R^3}$, where a vector is the sum of its projections onto the three coordinate axes. The only difference here is that the sum is infinite, and so a proof is needed to establish that it converges.

The **Fourier transform** of f is now defined to be the sequence $\{c_k : k \in \mathbf{Z}\}$. With this revised definition, we see that the Fourier transform of f is just the list of coefficients of f when it's written in a particular orthonormal basis. Such "lists of coefficients" form a vector space under term-by-term addition and scalar multiplication, and the Fourier transform is a *linear* transformation from L^2 to this new vector space. Be sure you understand this: *The Fourier transform is just a change of representation*. It's a very important one, though, because of the multiplication-convolution property.

The function $f \in L^2(H)$ is often referred to as being in the **time domain,** while its Fourier transform is said to be in the **frequency domain.** Since one is a function on an interval and the other is a function on the integers, the distinction between the two is quite clear. But for functions in $L^2(\mathbf{R})$, the Fourier transform is also in $L^2(\mathbf{R})$, and so being able to talk about the two domains is helpful. We'll sometimes use "value domain" or "value representation" for the original function, and "frequency representation" for its Fourier transform, because $f(x)$ tells us the *value* of f at x, while c_k tells us how much frequency-k content there is in f.

We mostly won't care about the particular values c_k in what follows, but we'll want to be able to take a big-picture look at these numbers and say things like "For this function, it turns out that $c_k = 0$ whenever $|k| > 200$," or "The complex numbers c_k get smaller and smaller as k gets larger." (Recall that the "size" of a complex number $z = a+b\mathbf{i}$ is called its **modulus,** and is $|z| = \sqrt{a^2 + b^2}$.) Because of this big-picture interest, rather than trying to plot c_k for $k \in \mathbf{Z}$, we instead plot $|c_k|$. The advantage is that $|c_k|$ is a real number rather than a complex one, so it's easier to plot. The plot of these absolute values is called the **spectrum** of f, and it tells us a lot about f. (The word "spectrum" arises from a parallel with light being split into all the colors of the spectrum.)

The Fourier transform takes a function in $L^2(H)$ and produces the sequence of coefficients c_k. It's useful to think of this sequence as a function defined on the integers, namely $k \mapsto c_k$. In fact, the sum

$$\sum_k |c_k|^2 \tag{18.48}$$

turns out to be the same as $\int_{-\frac{1}{2}}^{\frac{1}{2}} |f(x)|^2 \, dx$, which is finite because f is an L^2 function. This means that $k \mapsto c_k$ is an ℓ^2 function, and thus the Fourier transform takes $L^2(H)$ to $\ell^2(\mathbf{Z})$. From now on we'll denote the Fourier transform with the letter $\mathcal{F}$, so

$$\mathcal{F} : L^2(H) \to \ell^2(\mathbf{Z}) : f \mapsto \mathcal{F}(f). \tag{18.49}$$

Notice that $\mathcal{F}(f)$ is a *function*: $\mathcal{F}(f)(k)$ is defined to be c_k, the kth Fourier coefficient for f. For simplicity, we'll sometimes denote the Fourier transform of f by $\hat{f}$.

We'll often use two properties of the Fourier transform.

- First, if f is an even function, then each c_k is a real number (i.e., its imaginary part is 0). For even functions, we can therefore actually plot c_k rather than $|c_k|$; that's what we did in Figure 18.35, although we only plotted it for $k \geq 0$.
- Second, if f is real-valued (as are all the functions we care about, the real value being something like "the intensity of light arriving at this point"), then its Fourier transform is even, that is, $c_k = c_{-k}$ for every k. That's why we showed the plot of c_k for $k \geq 0$ in Figure 18.35: The values for $k < 0$ would have added no information.

We have one example of the Fourier transform already: We wrote the square-wave function s, as a sum of cosines in Equation 18.35. From that sum, we can read off

$$\hat{s}(k) = \begin{cases} 0 & k \text{ even} \\ (-1)^n \frac{4}{\pi n} & k = 2n+1 \end{cases}. \tag{18.50}$$

The plot of $\hat{s}$ is shown in Figure 18.38.

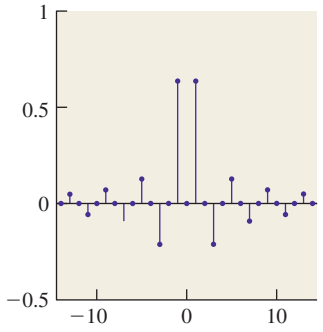


Figure 18.38: The Fourier transform of the square wave.

18.11.1 Sampling and Band Limiting in an Interval

Now suppose that we have a function f on the interval H with $\mathcal{F}(f)(k) = 0$ for all $|k| > k_0$. Such a function is said to be **band-limited at** k_0. The function f can be written as a sum of sinusoidal functions, all of frequency less than or equal to k_0. Since the "features" of a sinusoidal function of frequency k (the "bumps") are of size $\frac{1}{2k}$, the features of f must be no smaller than $\frac{1}{2k_0}$. We can say that the function f is "smooth at the scale $\frac{1}{2k_0}$." In a technical sense, f is completely smooth, but what we mean is that f has no bumpiness smaller than $\frac{1}{2k_0}$.

Turning this notion around, suppose that the graph of f has a sharp corner, or a discontinuity. Then f *cannot* be band-limited—it must be made up of sinusoids of arbitrarily high frequencies! This is important: A function that's discontinuous, or nondifferentiable, *cannot* be band-limited. The converse is false, however—there are plenty of smooth functions that contain arbitrarily high frequencies.

The set of all functions band-limited at k_0 is a vector space—if we add two band-limited functions, we get another band-limited function, etc. The dimension of this vector space is $2k_0 + 1$, with coordinates provided by the numbers $c_0, c_{\pm 1}, \ldots, c_{\pm k_0}$. (This is the dimension as a *real* vector space; each number c_j has a real and an imaginary part, contributing two dimensions, except for c_0, which is pure real.)

If we evaluate the function f at $k_0 + 1$ equally spaced points in the interval $H = (-\frac{1}{2}, \frac{1}{2}]$ we get $k_0 + 1$ complex numbers, which we can treat as $2k_0 + 2$ real numbers. If we ignore any one of these, we're left with $2k_0 + 1$ real numbers. That is to say, we've defined a linear mapping from the band-limited functions to $\mathbf{R}^{2k_0+1}$. This mapping turns out to be bijective. (The proof involves lots of trigonometric identities and some complex arithmetic.) What that tells us is somewhat remarkable:

> If f is band-limited at k_0, then any $k_0 + 1$ equally spaced samples of f determine f uniquely. Conversely, if you are given values for $k_0 + 1$ equally spaced samples (except for either the real or complex part of one value), then there's a unique function f, band-limited at k_0, that takes on those values at those points.

This is one form of the **Shannon sampling theorem** [Sha49] or simply **sampling theorem.** We can apply this to real-valued functions, whose Fourier transforms are even functions. This means that $c_{-1} = c_1$, and $c_{-2} = c_2$, etc. So, of the $2k_0 + 1$ degrees of freedom, we have only $k_0 + 1$ degrees of freedom for a real-valued function. In this case, the sampling theorem says:

> Suppose that f and g are real-valued functions on $[-\frac{1}{2}, \frac{1}{2}]$, and $x_0, \ldots, x_{k_0}$ are $k_0 + 1$ evenly spaced points in that interval, for example,
>
> $$x_j = -\frac{1}{2} + \frac{j}{k_0 + 1}, \tag{18.51}$$
>
> *and* $y_j = f(x_j)$ *for* $j = 0, \ldots, k_0$, *and* $y'_j = g(x_j)$.
>
> If $y_j = y'_j$ for all j, then f and g are equal, that is, a function band-limited at k_0 is completely determined by $k_0 + 1$ equally spaced samples. Furthermore, given any set of values $\{y_j\}_{j=0}^{k_0}$, there is a unique function, f, band-limited at k_0, with $f(x_j) = y_j$ for every j.

The sampling theorem was proved by Shannon in 1949, but Borel stated part of it as early as 1897. Part of it was also suggested by Nyquist in 1928. Several others appear to have developed all or part of it independently. Meijering [Mei02] gives some of the history.

Peeking ahead, this theorem is important because we generally build an image by taking equispaced samples of some function f, and we hope that the image really "captures" whatever information is in f. The sampling theorem says that *if* f is band-limited at some frequency, and *if* we take an appropriate number of samples for that frequency, then we can reconstruct f from the samples, that is, the image is a faithful representation of the function f.

This should make you ask, "Well, what happens if I take k_0 samples of a real-valued function that's *not* band-limited at k_0? What band-limited function do those correspond to?" We'll address this soon.

Inline Exercise 18.4: On the interval $H = (-\frac{1}{2}, \frac{1}{2}]$, consider the three points $-\frac{1}{3}, 0$, and $\frac{1}{3}$.
(a) What real-valued function, f_1, band-limited at $k_0 = 1$, has values $1, 0$, and 0 at these points? What functions f_2 and f_3 correspond to value sets $0, 1, 0$ and $0, 0, 1$? (You may want to use a computer algebra system to solve these parts.)
(b) Now find a band-limited function whose values at the three points are $-\frac{1}{2}, 1, -\frac{1}{2}$.
(c) What are the samples of $x \mapsto \cos(4\pi x)$ at these three points? Does this contradict the sampling theorem?

The sampling theorem can be read in reverse: If I'm taking samples with a spacing h between them, what's the highest frequency I can tolerate in my signal if I want to be able to reconstruct it from the samples? The answer is that the wavelength of the signal must be greater than twice h. The frequency, known as the **Nyquist frequency,** is therefore π/h.

Inline Exercise 18.5: Suppose you prefer the convention that $x \mapsto \sin(x)$ has frequency 1. What's the Nyquist frequency if the sample spacing is h?

18.12 Generalizations to Larger Intervals and All of R

If instead of functions on $H = (-\frac{1}{2}, \frac{1}{2}]$ we want to study functions on the interval $(-M/2, M/2]$ of length M, we can make analogous definitions. The definition of the Fourier transform gets an extra factor of $\frac{1}{M}$; the limits of integration change to $\pm M/2$, and instead of using the function e_k, for $k \in \mathbf{Z}$, we must use

$$e_{\frac{k}{M}}(t) = \cos\left(\frac{2\pi k}{M}t\right) + \mathbf{i}\sin\left(\frac{2\pi k}{M}t\right). \tag{18.52}$$

The Fourier transform now sends $L^2(-M/2, M/2)$ to $\ell^2(\frac{1}{M}\mathbf{Z})$, that is, functions on the set of all integer multiples of $1/M$. Thus, as the interval we're considering gets wider and wider (i.e., as M increases), the spacing between the frequencies involved in representing functions on that interval gets narrower and narrower.

It's natural to "take a limit" and consider what happens when we let $M \to \infty$. It turns out that in addition to the Fourier transform defined for $L^2(-M/2, M/2)$, we can define a Fourier transform for $L^2(\mathbf{R})$.

For $f \in L^2(\mathbf{R})$, we define $\mathcal{F}(f) : \mathbf{R} \to \mathbf{R}$ by the rule

$$\mathcal{F}(f)(\omega) = \int_{-\infty}^{\infty} f(x)e_\omega(x)\,dx, \tag{18.53}$$

where

$$e_\omega(x) = \cos(2\pi\omega x) + \mathbf{i}\sin(2\pi\omega x). \tag{18.54}$$

We can think of $\mathcal{F}(f)(\omega)$ as telling "how much frequency ω stuff there is in f," but this is a little misleading; it's perhaps better to say that $\mathcal{F}(f)(\omega)$ says "how much f looks like a periodic function of frequency ω."

Just as in the case of finite intervals, if $\mathcal{F}(f)(\omega) = 0$ for $|\omega| > \omega_0$, we say that f is **band-limited** at frequency ω_0.

Before we leave the subject of Fourier transforms, there's one last topic to cover: If we consider a periodic function h of period one, then h is definitely not in $L^2(\mathbf{R})$, because it doesn't tend to zero at $\pm\infty$, so the integral of Equation 18.53 won't generally converge. On the other hand, the corresponding integral *over just one period* of the function is the one used in defining the Fourier transform on an interval, Equation 18.46. Thus, we can use the interval formulation to talk about Fourier transforms for periodic functions as well.

◇ Roughly speaking, if we truncate a periodic function f of period one by setting $f(x) = 0$ for $|x| > M$, the $L^2(\mathbf{R})$ transform of the resultant function tends to be concentrated near integer points, and its value there tends to be proportional to M. As M gets larger, the concentration grows greater, until in the limit, the $L^2(\mathbf{R})$ transform is zero except at integer points, where it's infinite. By dividing by M, at each stage we can convert the infinite values to finite ones, and they look just like the $L^2(H)$ transform of a single period of f.

18.13 Examples of Fourier Transforms

18.13.1 Basic Examples

We've already seen (in Figures 18.34 and 18.35) the Fourier transform of one row of a natural image. The rapid falloff of $\mathcal{F}(f)(\omega)$ as ω grows is typical; in general, you can expect the Fourier transform to fall off like $1/\omega^a$ for some $a > 1$. For a

synthetic image, which has sharp edges (e.g., a checkerboard), you might expect a $1/\omega$ falloff, as we saw in the case of a square wave. In general, we'll plot signals in blue and their transforms in magenta, although on a few occasions we'll plot several signals on one axis and their transforms on another axis, using the same color for each signal and its transform. We'll plot discrete signals—ones whose domain is $\mathbf{Z}$—using stemplots.

18.13.2 The Transform of a Box Is a Sinc

Let

$$b(x) = \begin{cases} 1 & -0.5 \le x \le 0.5 \\ 0 & \text{otherwise,} \end{cases} \tag{18.55}$$

be a box function defined on the real line. Because it's an even function and it's real-valued, its Fourier transform will be even and real-valued. We can evaluate $\mathcal{F}(b)(\omega)$ directly from the definition.

$$\begin{aligned}
\mathcal{F}(b)(\omega) &= \int_{-\infty}^{\infty} b(x)\overline{e_\omega(x)}\,dx && (18.56)\\
&= \int_{-\frac{1}{2}}^{\frac{1}{2}} \overline{e_\omega(x)}\,dx \qquad \text{because } b(x) = 0 \text{ for } |x| > \frac{1}{2} && (18.57)\\
&= \int_{-\frac{1}{2}}^{\frac{1}{2}} \cos(2\pi\omega x)\,dx - \mathbf{i}\int_{-\frac{1}{2}}^{\frac{1}{2}} \sin(2\pi\omega x)\,dx && (18.58)\\
&= \int_{-\frac{1}{2}}^{\frac{1}{2}} \cos(2\pi\omega x)\,dx \qquad \text{because sin is odd} && (18.59)\\
&= \left.\frac{\sin(2\pi\omega x)}{2\pi\omega}\right|_{-\frac{1}{2}}^{\frac{1}{2}} && (18.60)\\
&= \frac{\sin(\pi\omega)}{\pi\omega}. && (18.61)
\end{aligned}$$

The calculation above works for all $\omega \neq 0$; for $\omega = 0$, we have $\mathcal{F}(b)(0) = 1$, which you should verify by writing out the integral.

This computation is shown pictorially in Figure 18.39, which is adapted from Bracewell [Bra99], an excellent reference for those interested in practical signal processing.

Inline Exercise 18.6: Repeat the preceding computation to compute the Fourier transform of a box of width a, that is, a function that's one on the interval $[-a/2, a/2]$ and zero elsewhere. Hint: Substitute $u = x/a$ in the integral to avoid doing any further work at all.

This is the only Fourier transform of a function on the real line that we'll actually compute directly like this. The resultant function is so important that it gets its own name:

$$\operatorname{sinc}(x) = \begin{cases} \frac{\sin(\pi x)}{\pi x} & x \neq 0 \\ 1 & x = 0 \end{cases}. \tag{18.62}$$

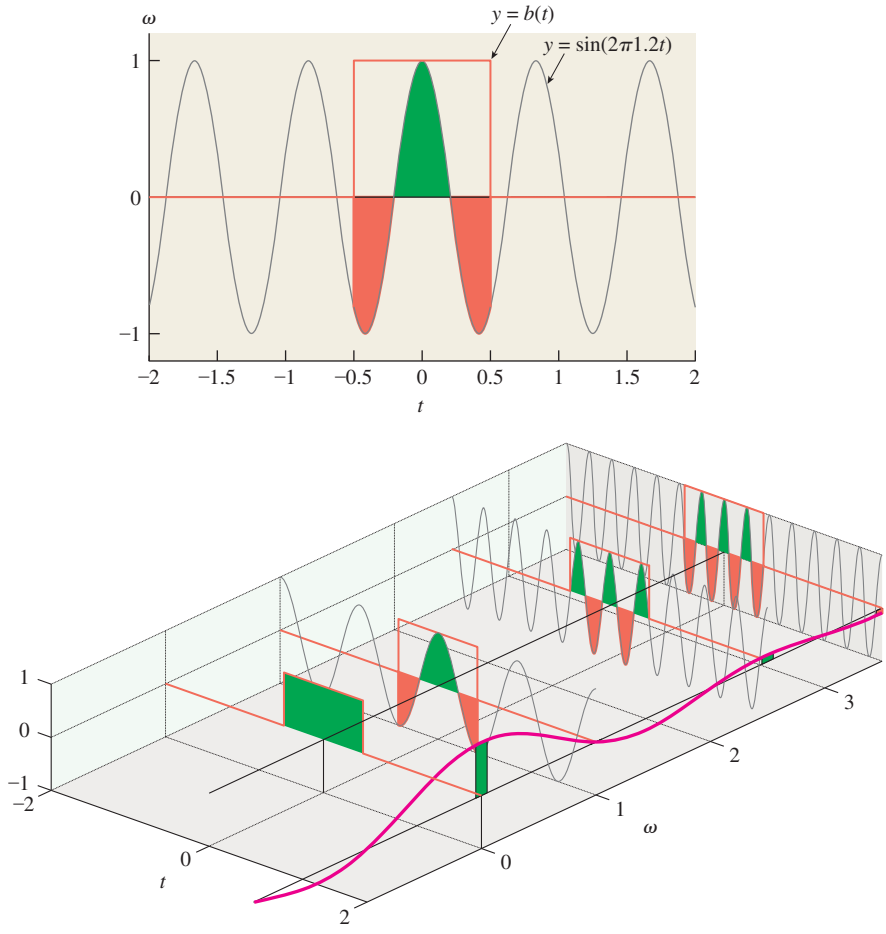


Figure 18.39: To compute the Fourier transform of the box b, for each frequency ω, we multiply $\sin(2\pi\omega x)$ *by b and compute the area beneath the resultant function, with positive area (above the tω-plane) shown in green and negative area in red. Top: The computation for* $\omega = 1.2$*. Bottom: Computations for several values of ω. For each one, we plot, at the right, the total area computed. This gives a function of the frequency ω, shown as a smooth magenta curve; the result is evidently* $\omega \mapsto \operatorname{sinc}(\omega)$*.*

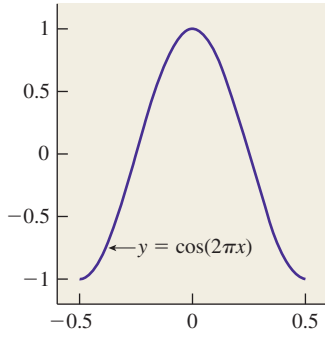


Figure 18.40: $x \mapsto \cos(2\pi x)$*.*

The function name is often pronounced "sink." Despite being described by cases, the function is smooth and infinitely differentiable; its Taylor series is just the series for $\sin(\pi x)$ divided by πx:

$$\operatorname{sinc}(x) = 1 - \frac{(\pi x)^2}{3!} + \frac{(\pi x)^4}{5!} - \dots. \tag{18.63}$$

18.13.3 An Example on an Interval

Consider the function $f(x) = \cos(2\pi x)$ (see Figure 18.40) on the interval H. Direct evaluation of the integral shows that $\mathcal{F}(f)(1) = \frac{1}{2}$ and $\mathcal{F}(f)(-1) = \frac{1}{2}$, and $FT(f)(k) = 0$ for all other k (see Figure 18.41). Thus, $f(x) = \frac{1}{2}e_1(x) + \frac{1}{2}e_{-1}(x)$, which is also obvious from the definition of $e_k(x)$.

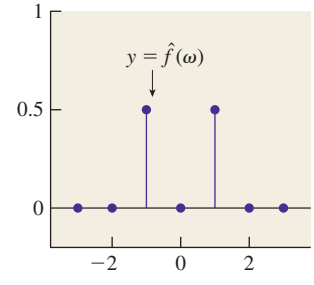


Figure 18.41: The Fourier transform for Figure 18.40, $k \mapsto \mathcal{F}(f)(k)$*.*

18.14 An Approximation of Sampling

If we again take a single row of the Taj Mahal signal and think of it as a function on the interval $[-\frac{1}{2}, \frac{1}{2}]$, we can multiply it by a function like the one shown in Figure 18.42, which removes most of the signal and retains only a small neighborhood of many evenly spaced points (we actually used a sampling function with about 100 peaks). Figure 18.43 shows the result near the center of the row, using pixel coordinates for the x-axis. As you can see, the resultant signal consists of many small peaks. This spiky signal has a Fourier transform that looks somewhat like the original Fourier transform, replicated over and over again (see Figures 18.44 and 18.45). This replication will be explained soon.

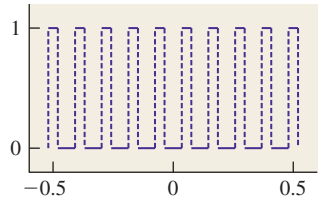


Figure 18.42: A sampling function.

The replication in this example isn't exact by any means. That's partly because we've used "wide" peaks to do the sampling, and partly because the Taj Mahal data itself is made of samples rather than being a true function on the real line, and we didn't do interpolation to turn it into such a function. But if the Taj data *were* a continuous function defined on the interval, and if our sampling peaks were very narrow, the Fourier transform would consist of a sum of almost exact replicates of the transform of the unsampled image.

You'll also notice that the transform of the sampled image isn't as large (on the y-axis) as the original (look carefully at the labels on the y-axis). That's because in our "sampling" process we've removed a lot of the data and replaced it with zeroes, hence every integral tends to get smaller.

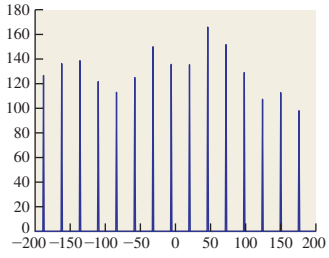


Figure 18.43: The Taj function multiplied by the sampler.

18.15 Examples Involving Limits

We need two more examples, each of which involves not a single function but a *sequence* of functions.

18.15.1 Narrow Boxes and the Delta Function

As you found when you computed the transform for a box of width a, as the box grows narrower, the transform grows wider: It's sinc-like, but instead of having zeroes at integer points, it has zeroes at multiples of $1/a$. You may also have noticed that, just as with the sampled Taj Mahal data, it gets smaller in the vertical direction: While the transform of the unit-width box reached height 1 (at $\omega = 0$), the transform of a box of width a reaches height a at $\omega = 0$.

Let's consider now

$$g(x,a) = \frac{1}{a} b\left(\frac{x}{a}\right), \tag{18.64}$$

which for any nonzero a is a box of width a and height $1/a$ so that the area under the box is always 1. Figure 18.46 shows a few examples. For any a, the transform of $x \mapsto g(x,a)$ is a sinc-like function with value 1 at $\omega = 0$, but as $a \to 0$, the "width" of the sinc grows greater and greater. Figure 18.47 shows the results.

The sequence of functions $x \mapsto g(x,a)$, as $a \to 0$, produces a sequence of Fourier transforms that approaches the constant function $\omega \mapsto 1$. In many engineering textbooks, the "limit" of this sequence is defined to be "the delta function $x \mapsto \delta(x)$," and its Fourier transform is observed to be the constant function 1. This literally makes no sense: The sequence of functions does not approach a limit at

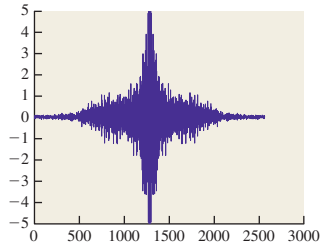


Figure 18.44: The transform of the Taj Mahal data.

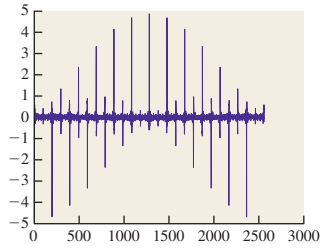


Figure 18.45: The transform of the "sampled" Taj Mahal data.

$x = 0$, and the supposed Fourier transform is not in L^2, because the integral of its square is not finite. Nonetheless, with some care one can work with the delta function by constantly remembering that the ordinary rules don't actually apply to it, and it's really a proxy for a limiting process.

⚠ The only way in which we'll ever want to use the δ function, or the comb function defined in the next section, is inside a mapping like

$$f \mapsto \int_{\mathbf{R}} \delta(x) f(x)\, dx, \tag{18.65}$$

that is, to define a real-valued function on L^2, which is a *covector* for L^2. If we replace $\delta(x)$ in Equation 18.65 with $g(x, a)$, and take a limit as $a \mapsto 0$, the resultant sequence of covectors actually *does* converge, although this requires proof. Thus, the δ function, at least inside an integral, makes some sense.

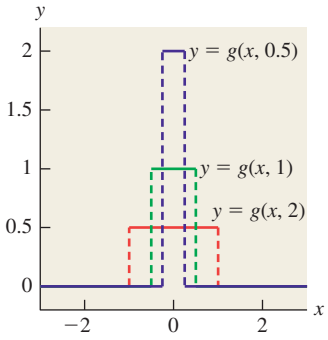


Figure 18.46: $x \mapsto g(x, a)$ for several values of a.

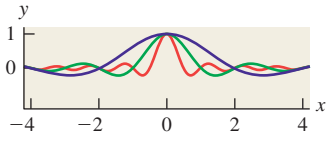


Figure 18.47: The Fourier transforms of the examples in Figure 18.46, matched by color.

18.15.2 The Comb Function and Its Transform

In much the same way we just analyzed a sequence of ever-narrower-and-taller box functions, we can consider a sequence of $L^2(\mathbf{R})$ functions that approaches a **comb,** a function with an "infinitely narrow box" at every integer. Figure 18.48 shows how we can do this: We place boxes of width a and height $1/a$ at each integer point, but then multiply their heights by a "tapering" function of width proportional to $1/a$ so that the total area under all the boxes is finite, hence the functions are all in $L^2(\mathbf{R})$.

Figure 18.49 shows the transforms of these functions. Just as with the delta function, the transforms seem to approach a limit, but in this case the limit is again the comb function (i.e., the transform grows larger and larger at integer points, while heading toward zero at all noninteger points).

We'll use the symbol ψ for the comb; informally, we say that $\mathcal{F}(\psi) = \psi$.

If we create a comb with spacing c instead of 1, its transform is a comb with spacing $1/c$, just as we saw with the box and the sinc.

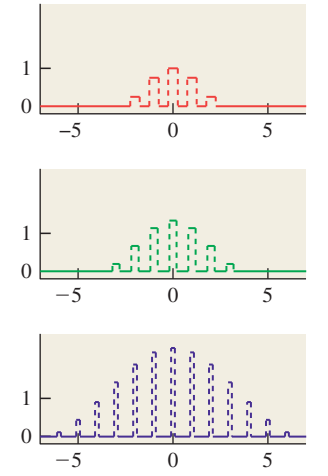


Figure 18.48: Functions that approach a comb.

18.16 The Inverse Fourier Transform

We've already said that if we take a function f in $L^2(H)$ (or a periodic function of period one) and compute its Fourier transform $c_k = \mathcal{F}(f)(k)$, then we can recover f by writing

$$\sum_k c_k e_k(t). \tag{18.66}$$

For a nice function f, this sum equals f except at points of discontinuity of f, and possibly the endpoints, if $f(\frac{1}{2}) \neq f(-\frac{1}{2})$. Thus, we've defined an **inverse transform** that takes a sequence of coefficients and produces an L^2 function on the interval (or a periodic function of period one).

There's a similar "inverse transform" defined for $L^2(\mathbf{R})$:

$$\mathcal{F}^{-1}(g)(x) = C \int_{-\infty}^{\infty} g(\omega) e_\omega(x)\, d\omega, \tag{18.67}$$

with the same kind of property: If we transform a nice function f to get g, and then inverse-transform g, we get back a function that's equal to f almost everywhere. This means that we can go back and forth between "value space" and "frequency space" with impunity.

18.17 Properties of the Fourier Transform

We've already noted that the Fourier transform is linear. And in studying the transform of the scaled box function, you should have observed that if

$$g(x) = f(ax) \tag{18.68}$$

then

$$\mathcal{F}(g)(\omega) = \frac{1}{a}\mathcal{F}(f)\left(\frac{\omega}{a}\right) \tag{18.69}$$

$$\mathcal{F}(f)(\omega) = a\mathcal{F}(g)(\omega a). \tag{18.70}$$

The proof follows directly from the definition after the substitution $u = ax$.

We'll call this the **scaling property** of the Fourier transform: When you "scale up" a function on the x-axis, its Fourier transform "scales down" on the ω-axis, and vice versa, as shown schematically in Figure 18.50.

Like most linear transformations, the Fourier transform is *continuous;* this means that if a sequence of functions f_n approaches a function g, then $\mathcal{F}(f_n)$ approaches $\mathcal{F}(g)$, assuming that both the f's and the g are all in L^2.

The Fourier transform has two final properties that make it important to us. The first is that it's *length-preserving,* that is,

$$\|\mathcal{F}(f)\| = \|f\| \tag{18.71}$$

for every $f \in L^2(\mathbf{R})$. The proof is a messy tracing through definitions, with some careful fiddling with limits in the middle.

The second property, whose proof is similar but messier, is the **convolution-multiplication theorem.** It states that

$$\mathcal{F}(f \star g) = \mathcal{F}(f)\mathcal{F}(g), \text{ and} \tag{18.72}$$

$$\mathcal{F}(fg) = \mathcal{F}(f) \star \mathcal{F}(g), \tag{18.73}$$

for any $f, g \in L^2(\mathbf{R})$. The same formulas apply when the Fourier transform is replaced by the inverse Fourier transform. The second formula also applies to functions defined on the interval H, or periodic functions of period one, although the convolution on the right is a convolution of *sequences* instead of a convolution of functions on the real line.

The convolution-multiplication function explains why it's generally difficult to deconvolve. Suppose that $\hat{g}$ is everywhere nonzero. Then convolving with g turns into multiplication by $\hat{g}$ in the frequency domain. If we let $h = f \star g$, then $\hat{h} = \hat{f}\hat{g}$. Now suppose we let $u = 1/\hat{g}$. Multiplying $\hat{h}$ by u gives $\hat{f}$. If U is the inverse Fourier transform of u, then *convolving* h with U will recover f, by the convolution-multiplication theorem. There is one problem in this formulation, however: If $\hat{g}$ is an L^2 function, then $u = 1/\hat{g}$ is generally *not* an L^2 function. But it may be well approximated by an L^2 function, so an approximate deconvolution is possible. On the other hand, suppose that $\hat{g}(\omega_0) = 0$ for some ω_0. Then it's impossible to even define u, let alone take its inverse transform. Roughly speaking, filtering by g removes all frequency-ω_0 content from f, and there's nothing we can do to recover that content later from the filtered result h.

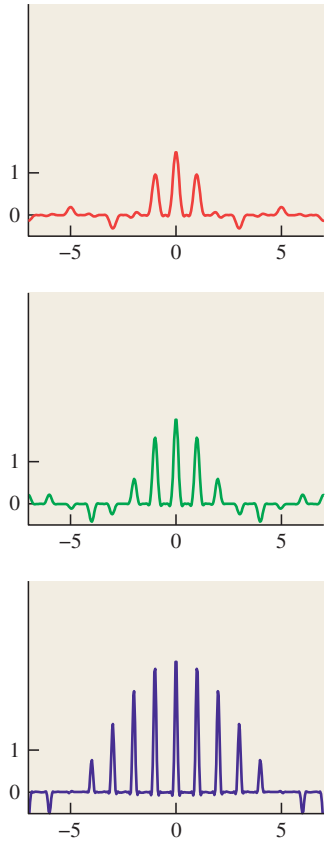


Figure 18.49: The transforms of the function in Figure 18.48.

18.18 Applications

We've defined two kinds of Fourier transform, and have observed that they are linear, continuous, and length-preserving, and satisfy the multiplication-convolution theorem. We can think of the Fourier transform as taking a "value representation" of a function f (i.e., the usual representation, where $f(x)$ is the value of f at the location x) into a "frequency representation," where $\mathcal{F}(f)(\omega)$ tells us "how much f looks like a sinusoid of frequency ω."

We'll now look at two applications of these ideas: band limiting and sampling.

18.18.1 Band Limiting

We've said that a function g is band-limited at ω_0 if $\mathcal{F}(g)(\omega) = 0$ for $\omega > \omega_0$, that is "if g contains only frequencies up to ω_0." This is illustrated schematically in Figures 18.51 and 18.52. For the remainder of this section, we'll fix ω_0 so that "band-limited" means "band-limited at ω_0."

Now we'll consider a similar computation on the interval H. Before we do so, we need one further fact: Just as the Fourier transform of a box was a sinc function, the inverse transform of a box is *also* a sinc function, as you can check by writing out the integrals.

Now suppose that f is a function in $L^2(H)$. What band-limited function g is *closest* to f? We'll answer this using the Fourier transform. Figure 18.53 shows the idea. In the top row we see f and its transform. It's mostly made up of frequencies less than 30, so we've truncated the transform to show the interesting parts. If we remove all the high frequencies (we've kept frequencies 17 and lower in this example), we get the function in the lower right. Removing all those frequencies amounts to multiplying by a *box* of width 34 (ranging from $\omega = -17$ to $\omega = +17$), that is, the function

$$B(\omega) = b(\omega/34). \tag{18.74}$$

Since multiplication by a box in the frequency domain is the same as convolution by a sinc in the value domain, the inverse transform of the lower-right signal, shown at the lower left, can also be obtained by convolving the original signal with an appropriately scaled sinc, namely

$$S(x) = \text{sinc}(34x). \tag{18.75}$$

Notice that the result is a far smoother signal, g.

The signal g appears quite similar to the original signal f. It is in fact the band-limited signal that's *closest* to f. That's easy to see by looking on the right-hand side of the figure. The Fourier transform is *distance preserving,* that is, the distance between f and g is the distance between $\hat{f} = \mathcal{F}(f)$ and $\hat{g} = \mathcal{F}(g)$. So to find the band-limited function closest to f, we need only look for the *transform* that is closest to $\hat{f}$, but is zero outside the band limit. The only freedom we have in picking $\hat{g}$ is to adjust it between frequencies -17 and $+17$; by making it match $\hat{f}$ there, we make the difference of $\hat{f}$ and $\hat{g}$ as small as possible.

Inline Exercise 18.7: ◇ Write out this argument with integrals.

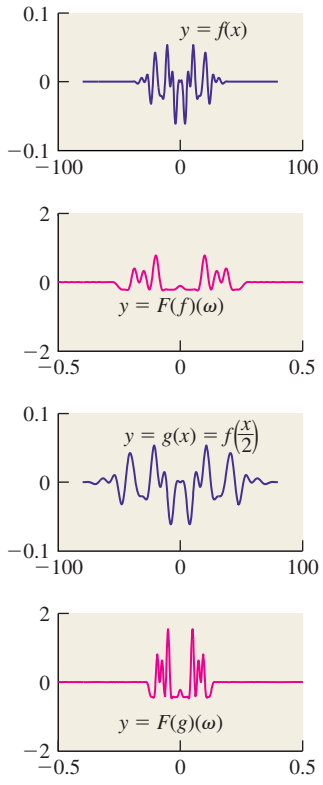


Figure 18.50: When we stretch the graph of f on the x-axis, the graph of $\mathcal{F}(f)$ (in magenta) compresses on the ω-axis and stretches on the y-axis.

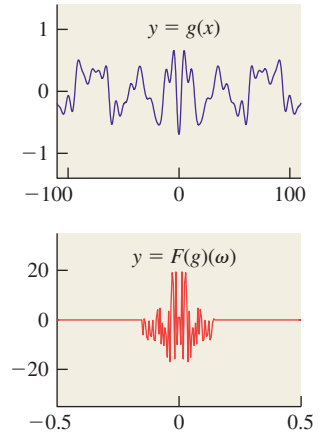


Figure 18.51: A band-limited function and its Fourier transform.

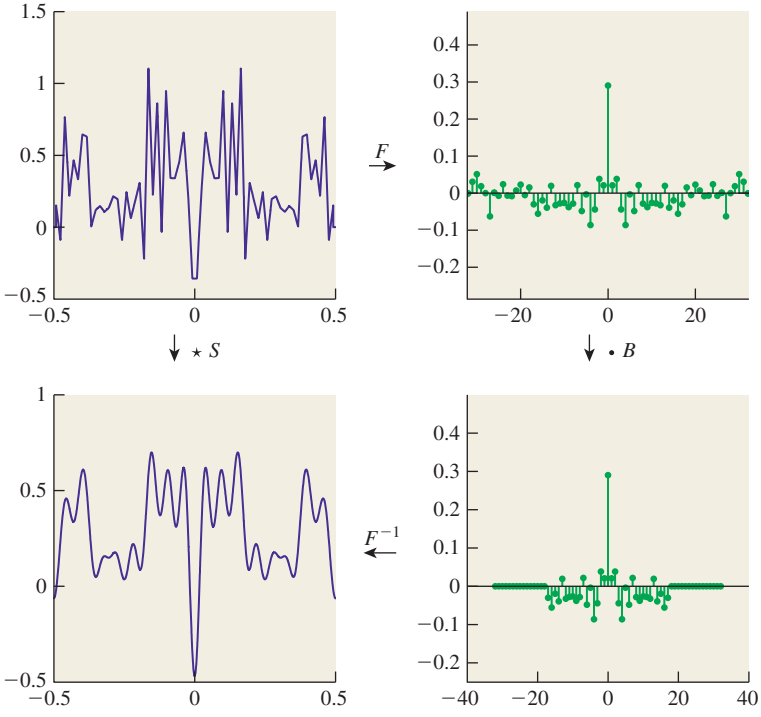


Figure 18.53: To get from $\hat{f}$ to $\hat{g}$, we multiply by a box $B(\omega) = b(\frac{\omega}{34})$; in other words, we remove all high frequencies. To get from f to g, we convolve with the inverse transform of B, namely a sinc *of width* $1/34$*, that is,* $x \mapsto 34\,\mathrm{sinc}(34\,x)$.

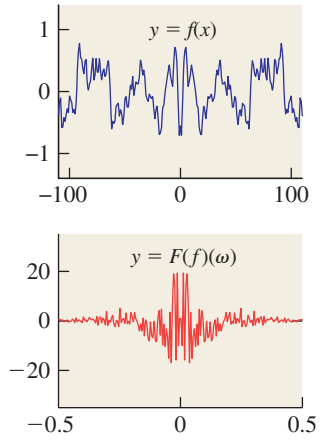


Figure 18.52: A non-band-limited function and its transform.

There's nothing special about the number 17 in this example. We can "band-limit" the function $f \in L^2(H)$ at any frequency ω_0.

We can summarize the preceding two pages: If f is a function in $L^2(H)$, then to band-limit f at frequency ω_0 we must convolve it with the function $x \mapsto S(x) = 2\omega_0\,\mathrm{sinc}(2\omega_0 x)$, or, correspondingly, multiply its Fourier transform by $\omega \mapsto B(\omega) = b(\frac{\omega}{2\omega_0})$. The result is the band-limited function g that's closest to f.

If you're thinking to yourself, "Gosh, computing the convolution involves an integral, and doing that at every single point sounds *really* expensive," you're right. Fortunately, we'll never need to actually do this in practice.

> If you *did* want to approximate such a convolution, the practical method is to take lots of samples of f, compute the "fast Fourier transform" (a discrete version of the Fourier transform that runs in $O(n \log n)$ time on n samples) on these samples, remove all the frequencies greater than ω_0, and then transform back again. That's how we made this chapter's figures.

18.18.2 Explaining Replication in the Spectrum

As a second application, let's revisit what we saw in Figure 18.45: When we multiplied the Taj Mahal data by a "sampling" function, the Fourier transform began to look periodic, as if it were made of multiple copies of the original transform, overlaid and summed up.

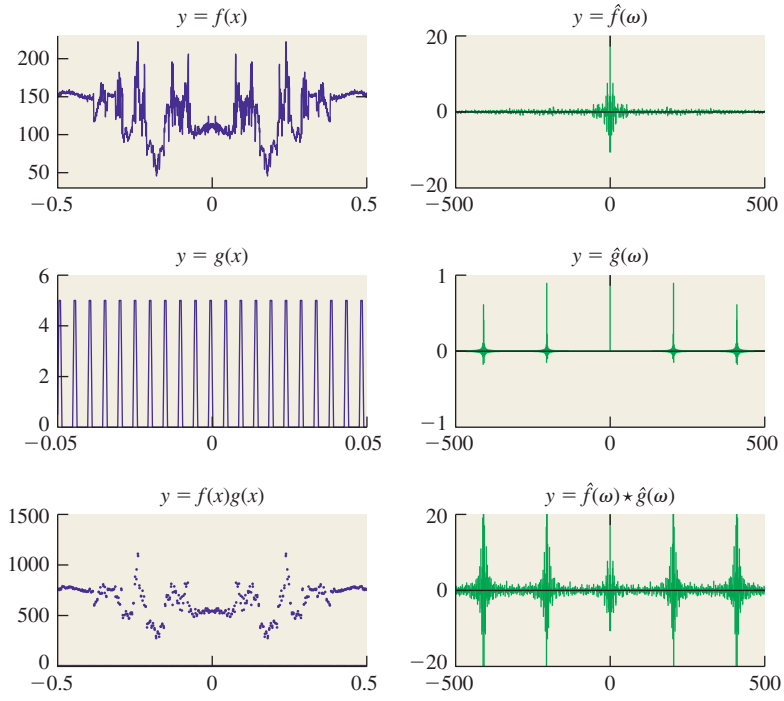


Figure 18.54: The Taj Mahal data (top left) is multiplied by a narrow comblike function (left middle—note the different scale on the x-axis!), with closely spaced peaks, to produce the "sampled" version at the bottom left. The Fourier transform of the original signal (top right) is convolved with the transform of the comblike signal (middle right, comblike with widely spaced peaks) to produce the transform of the sampled signal (bottom right), showing overlapping replicates of the transform of the original signal.

Figure 18.54 shows the situation, which we now explain. Let's say that the original Taj Mahal data is described by a function $x \mapsto f(x)$. The "sampled" version, which we'll call h, was generated by multiplying f by a function g that approximated a comb function, that is, that consisted of a bunch of narrow peaks of area 1, with spacing about $1/200$, to produce a signal $h = fg$. This means that $\hat{h} = \hat{f} \star \hat{g}$. But since g is an approximation of the comb function, its Fourier transform is an approximation of the transformed comb function, which is just another comb function. Since the spacing for g is about $1/200$, the spacing for $\hat{g}$ is about 200. So $\hat{h}$ is just $\hat{f}$ convolved with a comblike function with spacing of 200. That convolution consists of multiple copies of $\hat{f}$, one at each comb tooth, summed up. This explains the approximate periodicity of the Fourier transform $\hat{h}$.

18.19 Reconstruction and Band Limiting

We now further examine the relationship between a function, $f \in L^2(\mathbf{R})$, and its samples at integer points, which define a function $\bar{f} \in \ell^2(\mathbf{Z})$. We need a new definition: f is **strictly band-limited at** ω if $\hat{f}(\omega) = 0$ for $\omega \geq \omega_0$. Note the shift from "$>$" to "$\geq$." The main result of this section is as follows.

- If $\mathcal{F}(f)(\omega) = 0$ for $|\omega| \geq 1/2$, then f can be recovered from $\bar{f}$, that is, the map $f \mapsto \bar{f}$ is invertible once the domain is restricted to the functions strictly band-limited at $1/2$.
- For any function $g \in \ell^2$, there are multiple functions whose samples are given by g, so in general, it's impossible to reconstruct an arbitrary L^2 function from its samples.

The corresponding statements hold for a function $f \in L^2(H)$, sampled at a collection of n evenly spaced points in the interval; in that case, if the function is band-limited at any frequency below $n/2$, then it's reconstructible from its samples. (Indeed, that's the essence of the sampling theorem.)

We'll actually show exactly *how* to reconstruct a band-limited function from its samples, in that rather than just showing that the sampling process is invertible, we'll explicitly describe an inverse. We'll also describe some approximations to the inverse that are easily computable, that is, functions with the property that if f is band-limited, then sampling f, followed by the approximate-reconstruction operation, will yield a function very close to f.

We'll start with the easy part: showing there are multiple functions with the same samples. Recall that for a continuous function, "sampling at x" simply means "evaluating the function at x." So to show that there are two different functions with the same samples at integer points, consider $f_1(x) = 0$ and $f_2(x) = \sin(\pi x)$. At every integer point, these two have exactly the same value (namely 0). Thus, if you're given the samples of one of the two functions, you cannot possibly tell which one it was. Of course, f_2 isn't in $L^2(\mathbf{R})$, because it doesn't approach zero as $x \to \pm\infty$, but this quibble is easily resolved:

Inline Exercise 18.8: Show that if f is any function in L^2, then $x \mapsto f(x)f_2(x)$ is a function whose samples match those of f_1, so the sampling operation is a *many*-to-one map from $L^2(\mathbf{R}) \to \ell^2(\mathbf{Z})$.

Note, however, that the function f_2 has frequency $\frac{1}{2}$, so it's *just* above the Nyquist limit: We expect it to produce aliasing.

For the corresponding situation on the interval $H = (-\frac{1}{2}, \frac{1}{2}]$, a similar example suffices. If we consider the n equally spaced sample points $x_j = -\frac{1}{2} + \frac{j}{n}, j = 0, 1, \ldots, n-1$, then the function $g_1(x) = 0$ and the function $g_2(x) = \sin(\pi n(x + \frac{1}{2}))$ both take on the value 0 at every x_j.

We say that the function g_2 is an **alias** of the function g_1, because from the point of view of their samples, they appear to be the same. If we had a way to construct a function on the whole interval from a set of samples, then the samples of g_1 and g_2, being identical, would produce identical results.

Now let's turn to the more difficult part: showing that *if a function is appropriately band-limited, it can be reconstructed from its samples.*

As we saw earlier, if we have a function $f \in L^2(\mathbf{R})$ and multiply it by a "comb-like" function in L^2, the Fourier transform of the result starts to look periodic, consisting of multiple copies of the transform of f. As we take better and better approximations of the comb, the Fourier transform becomes closer and closer to the convolution of $\hat{f}$ with a comb. The limit of the comb approximations doesn't exist, of course, but the collection of samples of f *does* exist, and is a function $\bar{f}$ in ℓ^2. In the frequency domain, the Fourier transforms of the convolutions of f with more and more comblike approximations of the comb get closer and closer to a

periodic function. The limit *is* a periodic function, but that's not in $L^2(\mathbf{R})$, because it doesn't go to zero as $\omega \to \pm\infty$. On the other hand, it turns out that this periodic limit *is* the Fourier transform of $\bar{f}$. The proof of these claims is quite subtle; Dym and McKean [DM85] provide the necessary details for those who have studied real analysis.

To summarize the preceding paragraph briefly, if we take an $L^2(\mathbf{R})$ function f and sample it at integer points to get $\bar{f} \in \ell^2(\mathbf{Z})$, then

$$\mathcal{F}(\bar{f}) = \mathcal{F}(f) \star \psi, \tag{18.76}$$

where ψ is the comb function.

Suppose that f is strictly band-limited, that is, $\mathcal{F}(f)(\omega) = 0$ for $|\omega| \geq \frac{1}{2}$. Then $\mathcal{F}(\bar{f})$ consists of disjoint replicates of $\mathcal{F}(f)$. To recover the Fourier transform of f from this, we need only multiply it by a box of width 1, which is the function b. Multiplication by b in the frequency domain corresponds to convolution with $\mathcal{F}^{-1}(b)$ in the value domain. The inverse Fourier transform of b is the function $x \mapsto \operatorname{sinc}(x)$. We conclude that *to reconstruct a band-limited function from its samples, it suffices to convolve the samples with sinc.*

This is a pretty big result. It says, for instance, that if you have an image created by sampling a band-limited function f, you can recover f from the image by convolving with a sinc. In one dimension, that means that if the samples are called f_j, you can compute

$$f(x) = \sum_j f_j \operatorname{sinc}(j - x). \tag{18.77}$$

If x happens to be an integer—say, $x = 3$—then this sum becomes

$$f(3) = \sum_j f_j \operatorname{sinc}(j - 3). \tag{18.78}$$

The arguments to sinc in this sum are all integers, and sinc is 0 at every integer point, except that $\operatorname{sinc}(0) = 1$. So the sum simplifies to say

$$f(3) = f_3 \operatorname{sinc}(0) = f_3. \tag{18.79}$$

That's good: It says that to reconstruct the value of f at an integer point, you just need to look at the sample there. What if x is *not* an integer? Then the sinc is nonzero at every argument, and the value of f at x involves a sum of infinitely many terms. This is clearly not practical to implement.

We'll soon discuss other approaches to reconstruction, but the central idea—that we can reconstruct a function from its samples by convolving with sinc—remains important. We'll use it repeatedly in the next chapter when we discuss shrinking and enlarging images. Typically we'll apply this theoretical result multiple times to determine what computation we should be doing, and then, with the ideal computation at hand, we'll determine a good approximation.

We've now seen two applications of convolving with sinc. The first is that for any function $f \in L^2(\mathbf{R})$, $f \star \operatorname{sinc}$ is the band-limited function closest to f. As you'll recall, that's because in the frequency domain, convolution with sinc becomes "multiplication by a box," which removes all high frequencies from f, while leaving the low frequencies untouched. That's the Fourier transform of the band-limited function closest to f, hence its inverse transform is the band-limited function in the value domain closest to f. The second application is in reconstruction: To reconstruct a band-limited function from its samples, we convolve the samples with a sinc.

When we take a function and sample it as shown in Figure 18.55, it's natural, when viewing the samples, to mentally "connect the dots," as in Figure 18.56. We'll now study how this compares with reconstruction with sinc.

The first step is to recognize that "connecting the dots" gives the same results as convolving with the "tent function" b_1 of Figure 18.57. When we remember that $b_1 = b \star b$, we can see that connect-the-dots reconstruction is really just "convolve with b twice."

How does that look in the frequency domain? Ideally, we want to multiply by the unit-width box b in the frequency domain to get rid of all too-high frequencies. What we're doing instead is multiplying by $\mathcal{F}(b) = \text{sinc}$ twice—in other words, we're multiplying by sinc^2. How does this compare to multiplying by a box? Figure 18.58 shows the two functions, and you can see that they're somewhat similar. Because sinc^2 is nonzero for frequencies greater than $\frac{1}{2}$, it *does* allow some high-frequency components of f to masquerade as low-frequency components. But the peak of $\omega \mapsto \text{sinc}^2(\omega)$ outside $\omega \leq \frac{1}{2}$ occurs at about $\omega \approx 1.43$, where the value is about 0.047, that is, at most 5% of any too-high frequency manages to survive as an alias. Thus, sinc^2 does a decent job of band limiting. But what about its effects on frequencies that are low enough, that is, those that *should* be unattenuated? As we approach $\omega = \frac{1}{2}$ from below, sinc^2 falls off fairly rapidly. In fact, $\text{sinc}^2(\frac{1}{2}) \approx 0.23$, so signals near the Nyquist limit are attenuated to about one-quarter of the ideal. But by half the Nyquist limit, the attenuation is only about 20%. We should expect connect-the-dots reconstruction to work well in this region, but badly at or a little above the Nyquist limit.

Clearly if we were to convolve the tent function b_1 with the box b once more to get a new function b_2, its Fourier transform would be sinc^3, and it would better approximate the ideal box. On the other hand, convolving with the function b_2 would involve blending together not just two samples, but three.

In the value domain, we can regard the tent function as an approximation of the sinc. The tent looks somewhat like the central hump of the sinc, and hence their transforms are somewhat similar. Pursuing this idea, we could produce a piecewise quadratic or piecewise cubic function that better fit the first few lobes of the sinc, and whose Fourier transform would therefore be more like a box. Such an approximation is what's used by image-manipulation programs like Adobe Photoshop when the user selects "bicubic" interpolation.

When we display an image on an LCD monitor, we effectively take the sample values and use them to control the intensity of a square display pixel. The analog in one dimension is that we take each sample and expand it into a unit-width constant function, that is, we convolve the sampled signal with the box function b. In the frequency domain, that means we're multiplying by sinc *once,* which is much less effective at band-limiting the signal than multiplying by sinc^2. The result is more substantial aliasing than in the case of the tent reconstruction.

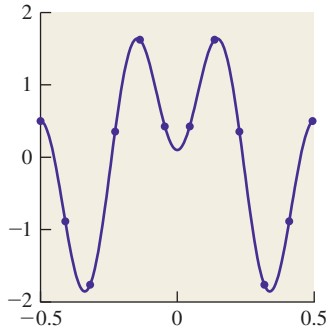


Figure 18.55: A function, sampled at equispaced points.

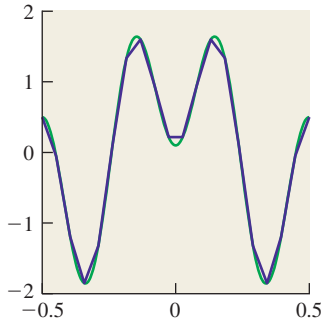


Figure 18.56: Reconstructing by connecting the dots.

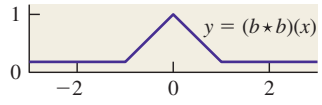


Figure 18.57: The tent function.

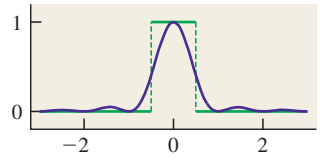


Figure 18.58: Comparing sinc^2 *with a box.*

18.20 Aliasing Revisited

In Section 18.2, we discussed line rendering for a grayscale LCD monitor using rounding, unweighted area sampling, and weighted area sampling. We'll now reexamine each approach using the value-frequency duality.

First, following the book's first principle—Know Your Problem—let's state the problem clearly. Given a line $y = mx + b$ with slope $0 < m < 1$, there's a

function f that's 1 for any point (x, y) whose distance to the line is less than $\frac{1}{2}$, and 0 otherwise. This function can be described as the "Am I in a unit-width stripe defined by $y = mx + b$" function. This function has sharp transitions from black to white (or 0 to 1), so its (two-dimensional) Fourier transform contains arbitrarily high frequencies. In fact, any horizontal slice of this function (i.e., $x \mapsto f(x, y_0)$) looks like a bump of some width, and hence its 1D Fourier transform looks like a sinc, which is nonzero for arbitrarily high frequencies. The function f is not in $L^2(\mathbf{R}^2)$, but outside the image we're going to render—say, a 100×100 image—we can define f to be 0, and then it *will* be in L^2. Our goal is to have the pattern of light emitted by the display be as near to a band-limited approximation of f (or a multiple of it, to deal with units) as possible.

The "rounding" approach turns out to be equivalent to sampling a slightly fatter version of the function f: The pixel (x, y) is "illuminated" if its *vertical* distance to $y = mx + b$ is no more than $\frac{1}{2}$. The samples of this function constitute an ℓ^2 function on the integer grid. Its spectrum is the result of convolving the spectrum of f with a two-dimensional comb, resulting in many high-frequency components aliasing as low-frequency ones. Displaying these samples on the LCD monitor amounts to convolving the image with a 2D box function, that is, multiplying by a (2D) sinc in the frequency domain. So the rounding approach, in the frequency domain, looks like convolution with a comb, followed by multiplication by a sinc. The end result is nowhere near a band-limited approximation of the function f, and the result, as we saw, looks bad.

In the next approach to line rendering, there were three steps.

1. Convolve with a 2D box to compute area overlaps.
2. Sample at integer points.
3. Convolve with a 2D box to display.

In frequency space, we multiply $\mathcal{F}(f)$ by a sinc, convolve with a 2D comb, and then multiply by sinc again. As we already saw, multiplying f by sinc weakly band-limits it—too-high frequencies are attenuated, although not perfectly. Convolution with the comb introduces the high-frequency parts that passed the weak band limiting as low-frequency aliases. And multiplying by sinc again weakly band-limits the results. The effect of the extra sinc in the first step is noticeable, and the grayscale line rendering is far nicer.

In weighted area sampling, the first step is replaced by convolution with a 2D tent rather than a 2D box; in the frequency domain, we're multiplying by sinc^2, which is a far more effective band limiter. The final results are correspondingly better.

In the last two cases, we've only approximately band-limited during the sampling process; in the first one, we never band-limited at all. And in all cases, the display process produces an image that contains a great many high frequencies at the edges between display pixels. But we have, at least in the last two cases, got an approximation of the ideal solution.

Or have we? We've actually failed in three ways. First, the notion of "nearness" used in this chapter is the L^2 distance, and we've already mentioned that this doesn't actually correspond very closely to a perceptual notion of similarity of images, so it's possible that we've optimized for the wrong thing. Second, we've been concerned about high frequencies, but in practice, once the pixels are small enough that the high-frequency components of the pattern of emitted light are so high-frequency that we cannot detect them with our eyes, these high

frequencies don't matter. Of course, a high frequency that aliases to a low frequency that we *can* detect *does* matter. But if some line-rendering approach leaves in a few components that are just above the Nyquist limit, their aliases will be just below the Nyquist limit, which may be well *above* the range of frequencies our eyes can detect, and therefore may not matter at all.

The third failure is of a larger kind: We've taken the view that we must first band-limit, then sample, then reconstruct, and we've looked at each step separately, and accepted approximations in each one. To some degree, we've approximated the steps of the solution rather than the solution itself violating the Approximate the Solution Principle. Here's an alternative problem to consider: Among all possible patterns of light *that the display can produce,* which one is L^2 closest to the function f? If we know that we're going to be displaying the result on a square-pixel LCD screen, isn't *this* a reasonable question to be asking? It turns out that the solution to this "nearest displayable image" problem is produced by unweighted area sampling. But actually using unweighted area sampling generates some interesting artifacts of its own: Vertical and horizontal lines look sharper, indeed *are* sharper, than diagonal ones, and lines in horizontal motion appear to speed up and slow down as we saw in the case of the moving triangle at the start of the chapter. Does this matter? That depends on our eyes' ability to detect variations in speed of motion for various speeds. You might want to write a program to experiment with this and draw your own conclusions. The real lesson of this example is that it's worth thinking about sampling and reconstruction *together* rather than as separate processes.

18.21 Discussion and Further Reading

If we want an image (i.e., a rectangular array of samples from some function f) to be faithfully reconstructible (i.e., we want to be able to recover f from the samples), then the process that generated the image must be lossless (i.e., an invertible map of vector spaces). In general, this requires that we restrict the original function to some subset of L^2, and the usual choice is "the band-limited functions."

Unfortunately, in practice we're often confronted with functions we'd like to sample but which are not band-limited. The solution is to find a way to convert such a function f into a nearby function f_0 that *is* band-limited. The ideal way to do so is to convolve f with a sinc, but that's impractical in general. Convolving with other, simpler, filters like the box can give a decent approximation.

In practice, this means that if you want to write a ray tracer, you shouldn't just sample one ray at the center of each pixel. Instead, you should shoot many rays per pixel and average them. This is a low-budget approximation of box-filtering the "incoming light" function. Alternatively, you could compute a *weighted* sum of the ray values, approximating the convolution of the incoming light with some filter like the sinc or the tent, or any other filter you like.

Although the sinc filter is the ideal "low-pass" filter, it has some problems in practice. If you have a wide box function, for instance, and you filter it with sinc, the result contains **ringing**—little wiggles on either side of the discontinuity. That's not "wrong" in any sense, but it presents a problem for display: Because some of the resultant values are negative, you want to make those parts of your display "even blacker than black," which is impractical. This is yet another reason to favor a tent filter, or some other everywhere-positive approximation of a sinc.

The idea that "details below the scale of our eyes' ability to detect them don't matter" can be used in an interesting way: The three colored strips in a typical LCD display pixel can be individually adjusted in ways that give finer control than adjusting all three together, even for a grayscale image. For instance, the antialiased letter "A" shown in Figure 18.14 was originally rendered this way; Figure 18.59 shows how it looked. This technology, used in font rendering, is called TrueType [BBD+99]. Ideas like this can surely be used in other ways in graphics as well.

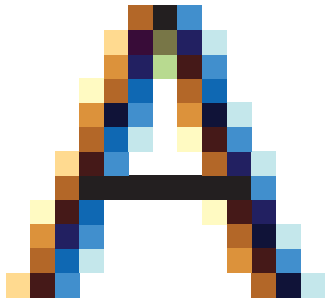


Figure 18.59: Enlarged color rendering of a black character on a white background.

The Nyquist limit, from the discussion in this chapter, appears to be absolute: You can't sample signals with frequencies above the Nyquist limit and hope to reconstruct them. But that's not completely true. Suppose that the Nyquist frequency for some sampling rate is ω_0. Then a signal whose Fourier transform is nonzero strictly between $-\omega_0$ and ω_0 can be perfectly reconstructed from its samples. But it's also the case that a signal whose Fourier transform is nonzero strictly between $5\omega_0$ and $7\omega_0$ can be perfectly reconstructed from its samples, *provided we know at the time of reconstruction these limits on the transform.* Indeed, if we know the samples of a function f and we know an interval I of length $2\omega_0$ with the property that f's transform is nonzero only strictly within I, then we can reconstruct f. Similarly, if we know that the transform of f is sparse—that is, nonzero at relatively few points—we can use this sparsity to reconstruct f even if its transform is *not* constrained to an interval of length ω_0. This is part of the subject of the relatively new field of **compressive sensing** [TD06].

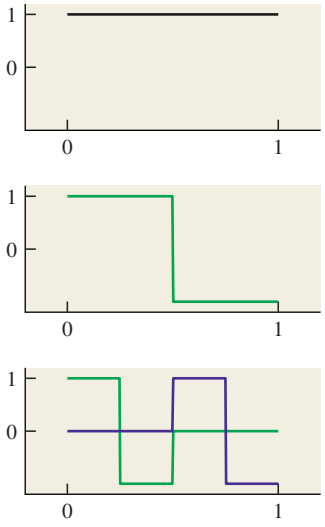


Figure 18.60: The Haar wavelets for $k = 0, 1,$ and 2. For $k > 0$ there are 2^{k-1} basis functions.

We started this chapter by saying that every L^2 function on an interval can be (nearly) written as a sum of sines and cosines. You might reasonably ask, "Why sines and cosines? Why not boxes of varying width, or tentlike functions, or some other collection of functions?" The first answer, which we'll return to in a moment, is that you *can* write an L^2 function as a sum of things other than sines and cosines, and it's often worthwhile to do so. But the Fourier decomposition has proven widely useful in engineering, mathematics, and physics. Why? One answer is based on the principle that if you have a linear transformation from a space to itself, it's often easiest to understand that transformation when you change to a basis made up of eigenvectors, for then the transformation is just a nonuniform scaling transformation. Many of the laws of physics appear as second-order linear differential equations, like $F = ma$, in which the unknown position x is described by saying that its second derivative, a, must satisfy $F = ma$, where the mass m and the force F are typically known, and x is required to satisfy some boundary conditions as well. In the event that F and m are constant over time, this is a second-order equation with constant coefficients. The solutions to such equations can be generally written as sums of exponentials, where the exponent may be real or complex. The complex case leads to sines and cosines. Thus, expressing things as a sum of sines and cosines arises naturally because of the world's defining equations being second-order linear equations. In fact, Fourier introduced the transform in the process of describing how to solve the **heat equation,** which describes how the heat in a solid evolves over time in response to initial and boundary conditions.

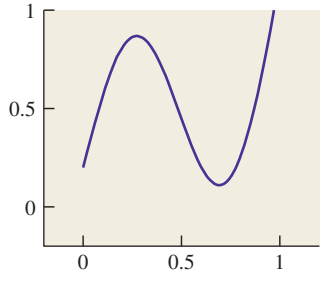


Figure 18.61: A nice function on the unit interval.

Oppenheim and Schaefer [OS09] make a similar case for discrete signals, saying that every linear shift-invariant system has, as its fundamental solutions, combinations of sines and cosines. **Shift-invariant** in this context means that if you regard the system as one that takes an input signal that's a function of t and produces an output signal that's another function of t, then delaying the input

(i.e., shifting from $t \mapsto s(t)$ to $t \mapsto s(t-h)$) produces the exact same output, but shifted by the same amount. In physical terms, it says that the system will behave the same way tomorrow that it does today.

We return now to the first answer—that you *can* write functions as sums of things other than sines and cosines. Two of the key features of the Fourier decomposition are localization in frequency and orthogonality. The first means that we can look at the Fourier transform of a signal and to the degree that the signal is mostly made up of one frequency, the Fourier transform will be small except at that frequency. The second means that the inner product of $\exp(\mathbf{i}n\pi x)$ and $\exp(\mathbf{i}k\pi x)$ is 0 unless $k = n$; this means that it's easy to write a function f in the Fourier basis by just computing the inner product of f with $\exp(\mathbf{i}n\pi x)$ for each n, and using the resultant numbers as coefficients in the linear combination.

In his 1909 dissertation under Hilbert, Haar showed that every L^2 function on $[0,1]$ could be well approximated by a sum of functions that were constant on intervals of size 2^{-k} (for $k = 0, 1, 2, \ldots$) and localized in *space,* that is, each function is nonzero only on two such intervals. These functions (and corresponding ones for larger k) are called the **Haar wavelets.** Figure 18.60 shows a few of the functions.

Figure 18.61 shows a function f on the unit interval, while Figure 18.62 shows an approximation of it that's constant on intervals of length $\frac{1}{8}$. Figure 18.63 shows how that approximation can be written as a linear combination of the Haar wavelets: The next-to-bottom row is the weighted sum of the $k = 3$ wavelets, each portion drawn in a different color; the next up is the sum of the $k = 2$ wavelets, etc. The top row is a constant function whose value is the average value of f on the interval. The height of the vertical red bar in the bottom row is the sum of the heights of the red bars in all rows above it.

If we take a limit and approximate the signal by such functions at finer and finer scales, the coefficients of the resultant (infinite) linear combination is called the **Haar wavelet transform** of the original function.

While Haar wavelets are conceptually very simple, and share some properties of the Fourier basis, they lack some others. For instance, they are not infinitely differentiable; indeed, they're not even once-differentiable. In the mid-1990s, Haar wavelets, and several other more complex forms of wavelets, with varying degrees of smoothness, were widely adopted in computer graphics, with applications all the way from line drawing [FS94] to rendering [GSCH93]. For those who wish to learn more, we highly recommend *Wavelets for Computer Graphics: A Primer,* by Stollnitz et al. [SDS95], as a gentle introduction to the subject, motivated by examples from graphics.

By the way, there's an analog to the Shannon theorem for Haar wavelets (or generally for any other basis in which you choose to write a function): If you have enough samples of a function that's known to be a combination of a fixed number of certain basis functions, you can reconstruct the function. For some classes the locations of the samples may be restricted in various ways (e.g., for Haar wavelets, they should not be of the form $p/2^q$, where p and q are integers), but the general idea still works.

The Fourier basis is a great way to represent one- and two-dimensional signals, but in graphics we also tend to encounter functions on the sphere, or on $\mathbf{S^2} \times \mathbf{S^2}$, like the BSDF. There's an analogous basis for $\mathbf{S^2}$ called **spherical harmonics,** which we touch on briefly in Chapter 31. A nice introduction to these is provided by Sloan [Slo08].

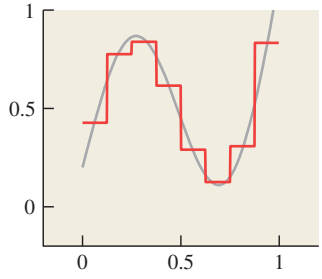


Figure 18.62: An approximation of the function that's constant on intervals of size $\frac{1}{8}$.

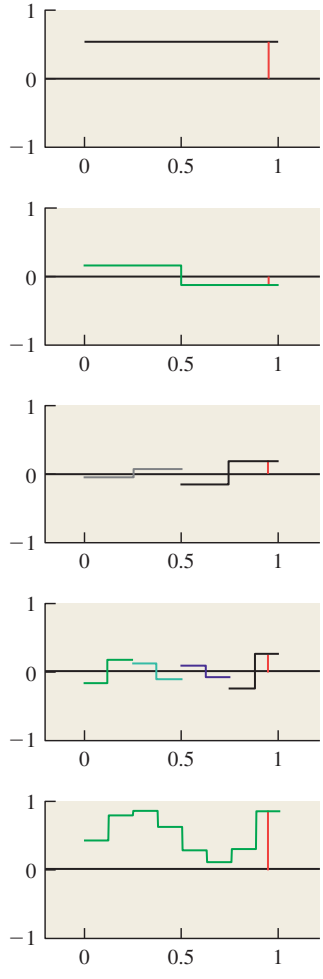


Figure 18.63: The approximation written as a sum of spatially localized functions of various scales.

18.22 Exercises

Exercise 18.1: (a) We defined the length $\|f\|$ for a function in L^2. How do you know that the length of any function $f \in L^2$ is finite?
(b) Show that $\|f+g\|^2 - \|f\|^2 - \|g\|^2 = 2\langle f, g\rangle$ for any $f, g \in L^2$, and conclude that the inner product of f and g is therefore always finite as well.

Exercise 18.2: (An exercise in definitions.) Define $g_a(t) = \frac{1}{2a}$ for $|t| < a$ and 0 otherwise.
(a) Show that for any function $f\colon \boldsymbol{R} \to \boldsymbol{R}$, the value $U(a)$ defined in Equation 18.26 is $(f \star g_a)(t_0)$.
(b) Show that the sample of f at t_0 is $\lim_{a\to 0}(f \star g_a)(t_0)$.

Chapter 19

Enlarging and Shrinking Images

19.1 Introduction

In Chapters 17 and 18 you learned how images are used to store regular arrays of data, typically representing samples of some continuous function like "the light energy falling on this region of a synthetic camera's image plane." You also learned a lot of theoretical information about how you can understand such sampled representations of functions by examining their Fourier transforms. In this chapter, we apply this knowledge to the problems of adjusting image sizes (scaling images up and down, as shown in Figure 19.1, which are synonyms for "enlarging" and "shrinking"), and performing various operations such as edge detection.

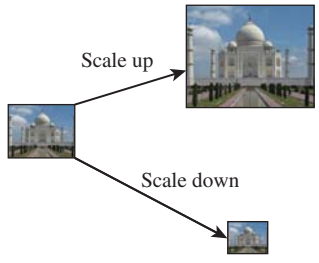


Figure 19.1: Terminology for image scaling.

We assume that the values stored in an image array form a signal that is real-valued, *not* discrete; nothing we say here applies to an object-ID image, for example.

Just as we did in Chapter 18, in this chapter we study primarily grayscale images. Scaling up or down a grayscale image entails all the main ideas without the complications of three color channels. Furthermore, we continue to study the effects of transformations on a single row of image pixels, because the extension to two dimensions really has no important properties beyond those of one dimension, but the notation is substantially more complex. We do, however, return to two dimensions when providing code for scaling up and down, and when we analyze the efficiency of computing convolutions.

In this chapter, we use the following ideas from Chapter 18.

- Sampling and convolution operations on a signal can be profitably viewed in both the value and the frequency domains.
- The convolution-multiplication theorem. Convolution in the value domain corresponds to multiplication in the frequency domain, and vice versa.

- The transform of the unit-width box-function b is the unit-spacing sinc function, $\text{sinc}(\omega) = \frac{\sin(\pi\omega)}{\pi\omega}$, while the transform of sinc is the box b.
- The scaling property. If $g(x) = f(ax)$, then

$$\mathcal{F}(g)(\omega) = \frac{1}{a}\mathcal{F}(f)\left(\frac{\omega}{a}\right) \quad \text{and} \tag{19.1}$$

$$\mathcal{F}(f)(\omega) = a\mathcal{F}(g)(a\omega). \tag{19.2}$$

- Band limiting and reconstruction. If
  - f is a function in $L^2(R)$, and
  - $\mathcal{F}(f)(\omega) = 0$ for $|\omega| \geq \frac{1}{2}$, and
  - y_i is the sample of f at i for $i \in \mathbf{Z}$

  then f can be reconstructed from the y_i's by convolution with sinc, that is,

$$f(x) = \sum_{i=-\infty}^{\infty} y_i \,\text{sinc}(x - i). \tag{19.3}$$

  Furthermore, if f is *not* band-limited at $\frac{1}{2}$, then sampling of f will produce aliases, in the sense that there is a band-limited function g whose samples are the same as those of f, and reconstruction using Equation 19.3 will produce g rather than f.
- A sequence of L^2 functions that approach a comb with unit tooth-spacing have Fourier transforms that approach a comb with unit tooth-spacing. Rather than talking about sequences that approach a comb, we'll use the symbol ψ as if there really were such a thing as a "comb function," and you'll understand that in any such use, there's an implicit argument about limits being suppressed.

19.2 Enlarging an Image

In this and the next section, we'll consider the problem of enlarging, or scaling up, and shrinking, or scaling down, an image. You might think that such operations would be straightforward, at least in some cases. If we have a 300×300 image, for example, and want a 150×150 version, it seems as if simply throwing away alternate rows and alternate columns would provide the desired result. Exercise 19.7 shows that this simple solution leads to very bad results, so we'll need a different approach. Fortunately, the different approach we describe will solve the problem of scaling up and down not only by small integer factors, but by any factor at all. The scaling-up operation, which we address in this section, is relatively easy. The scaling-down operation, discussed in the next section, has additional subtleties.

We'll work in one dimension as usual, so we'll start with a 300-sample discrete signal (which we'll call the source) that we want to turn into a 400-sample discrete signal (which we'll call the target). We'll assume that the 300-sample image was generated by sampling a function $S \in L^2(\mathbf{R})$ at 300 consecutive integer points.

We'll also assume that the signal $S : \mathbf{R} \to \mathbf{R}$ was strictly band-limited at $\omega_0 = \frac{1}{2}$ so that there was no aliasing when the samples were taken.

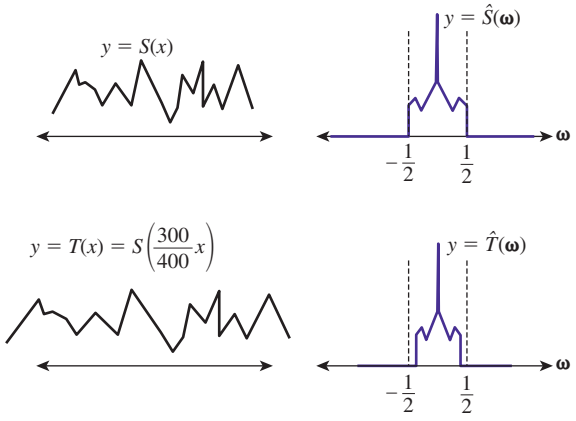


Figure 19.2: When the band-limited function S at left is stretched along the x-axis to form T, the transform of S at right is compressed, resulting in a more tightly band-limited transform.

To resample S at 400 points, we're going to imagine three idealized but impractical steps, which we'll later refine to a practical algorithm.

1. Reconstruct S from the 300 samples in the image.
2. Stretch S along the x-axis by a factor of $\frac{400}{300}$.
3. Resample S at the 400 sample points $i = 0, \ldots 399$.

When we reconstruct S in step 1, it's band-limited at $\omega_0 = \frac{1}{2}$. When we take 400 samples, if we want to avoid aliasing the signal must again be band-limited at $\frac{1}{2}$. Fortunately, when we *stretch* the signal S on the x-axis, the Fourier transform *compresses* along the ω-axis; the resultant signal is therefore still band-limited at $\frac{1}{2}$. You can see, however, that when we want to *shrink* an image there will be additional challenges. Figure 19.2 shows this schematically.

There is a difficulty with the idea of reconstructing a signal from its samples at integer points. To do so, we need to know the value at *every* integer point, not just 300 of them. For now, we're going to hide this problem by treating the source image outside the range 0 to 299 as being zero. We'll return to this assumption in Section 19.4.

By the way, for a function f on $\mathbf{Z}$ or $\mathbf{R}$, the **support** of f is the set of all places where it's nonzero, that is,

$$support(f) = \{x : f(x) \neq 0\}. \tag{19.4}$$

If this set is contained in some finite interval, we say f has **finite support;** on the other hand, if the support is not contained in any finite interval, then f has **infinite support.** Thus, the box function has finite support, while sinc has infinite support. We've just chosen to treat our image samples as a function with finite support. With this assumption, we can recover the original signal S as shown in Listing 19.1. Note that rather than reconstructing the entire original signal, which would require an infinite amount of work, we've merely given ourselves the ability to evaluate this reconstructed signal at any particular point, thus converting the abstract first step into something practical.

Listing 19.1: Evaluating the original signal by convolving source image samples with sinc.

```
// Reconstruct a value for the signal S from 300 samples in the
// image called "source".
double S(x, source) {
   double y = 0.0;
   for (int i = 0; i < 300; i++) {
      y += source[i] * sinc(x - i);
   }
   return y;
}
```

Notice that the sum is finite because we've assumed that `source[i]` is `0` except for $i = 0, \ldots, 299$. In general, the sum would have to be infinite, or, in practice, a sum over a very wide interval.

Now that we've reconstructed the original band-limited function, S, we need to scale it up to produce a new function T. If we think of each sample of S as representing the value of S at the middle of a unit interval, then the 300 samples we have for S, at locations $0, \ldots, 299$, represent S on the interval $[-0.5, 299.5]$. We want to stretch this interval to $[-0.5, 399.5]$.

To do so, we write

$$T(x) = S\left(\frac{300}{400}(x + 0.5) - 0.5\right). \tag{19.5}$$

Once again, rather than building the signal T, we've merely provided a practical way to evaluate it at any point where we need it.

Inline Exercise 19.1: Verify that $T(-0.5) = S(-0.5)$ and $T(399.5) = S(299.5)$.

We must now sample T at integer locations. Since T is band-limited, it must be continuous, so sampling at x amounts to evaluation at x.

Clearly the numbers 300 and 400 can be generally replaced by numbers N and K where $K > N$, and every step of the process remains unchanged. The resultant code is shown in Listing 19.2.

Listing 19.2: Scaling up a one-dimensional source image.

```
// Scale the N-pixel source image up to a K-pixel target image.
void scaleup(source, target, N, K)
{
   assert(K >= N);
   for (j = 0; j < K; j++) {
      target[j] = S((N/K) * (j + 0.5) - 0.5 , source, N );
   }
}

double S(x, source, N) {
   double y = 0.0;
   for (int i = 0; i < N; i++) {
      y += source[i] * sinc(x - i);
   }
   return y;
}
```

To scale up a source image in two dimensions, we simply scale each row first, then scale each column (see Figure 19.3).

Inline Exercise 19.2: Convince yourself that scaling rows-then-columns results in the same image as scaling columns-then-rows.

19.3 Scaling Down an Image

We now turn to the more complicated problem of scaling *down* an image I, that is, making it smaller. We'll assume that the source image has N pixels and the target image has only $K < N$ pixels.

Once again we can reconstruct from integer samples by convolving with sinc to get the original signal S. But in the next step, when we build

$$T(x) = S\left(\frac{K}{N}x\right), \tag{19.6}$$

the resultant function is no longer band-limited at $\frac{1}{2}$; instead, it's band-limited at $\frac{N}{2K} > \frac{1}{2}$. Before we could safely sample T, we would have to band-limit it to frequency $\frac{1}{2}$ by convolving with sinc.

Let's look at the process in the frequency domain, as shown schematically in Figure 19.4. In reconstructing S, we convolved with sinc, that is, we multiplied by a width-one box in the frequency domain. When we squashed S to produce T, we stretched the Fourier transform of S correspondingly, and then needed to once again multiply by a unit-width box.

What if we instead multiplied by a box of width K/N *before* stretching, as shown in Figure 19.5? Then when we stretch the result by N/K, we'll already be band-limited at $\omega_0 = \frac{1}{2}$. So, in the frequency domain, the sequence of operations is as follows.

1. Multiply by $\omega \mapsto b(\omega)$ to reconstruct.
2. Multiply by $\omega \mapsto b((N/K)\omega)$ to band-limit to $\frac{K}{2N}$.
3. Stretch by a factor of N/K; the result is band-limited at $\frac{1}{2}$.
4. Convolve with ψ as a result of sampling at integer points.

Notice that the first two steps can be combined: Multiplying by a wide box and then a narrow box gives the same result as multiplying by just the narrow box! This means that instead of reconstructing and then band-limiting, we can reconstruct and band-limit in a single step, by using a wider sinc-like function in the reconstruction process. Instead of $x \mapsto \text{sinc}(x)$, we need to use $x \mapsto \frac{K}{N}\text{sinc}(\frac{K}{N}x)$. After that, all the remaining steps are the same. The program is shown in Listing 19.3.

Listing 19.3: Scaling down a one-dimensional source image.

```
1  void scaledown(source, target, N, K)
2  {
3     assert(K <= N);
4     for (j = 0; j < K; j++) {
5        target[j] = SL((N/K) * (j + 0.5) - 0.5, source, N, K);
6     }
```

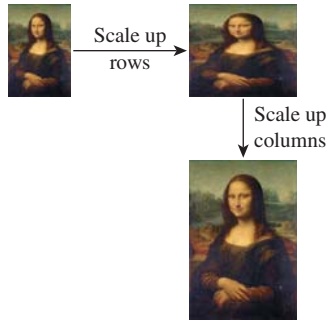


Figure 19.3: Scaling up rows and then columns to enlarge an image (Original image from http://en.wikipedia.org/wiki/File:Mona_Lisa,_by_Leonardo_da_Vinci,_from_C2RMF_retouched.jpg.)

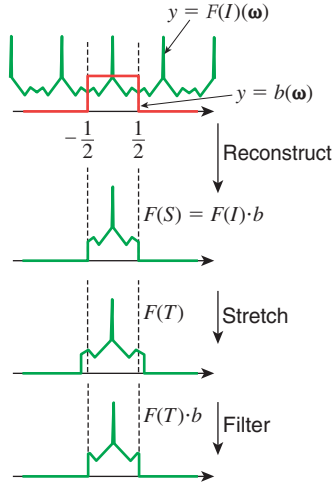


Figure 19.4: Band-limiting S after scaling.

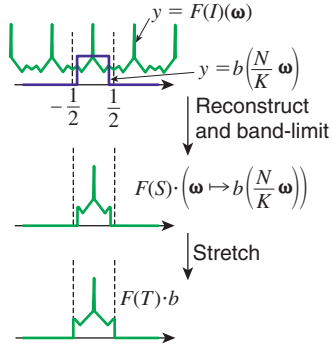


Figure 19.5: Band-limiting S before scaling.

```
}

// computed a sample of S, reconstructed and bandlimited at K/2N.
double SL(x, source, N, K) {
   double y = 0.0;
   for (int i = 0; i < N; i++) {
      y += source[i] * (K/N) * sinc((K/N) * (x - i));
   }
   return y;
}
```

19.4 Making the Algorithms Practical

These are *almost* practical algorithms for image scaling. They both, however, rely on the assumption that we can use zeroes for the samples outside the source image; the result is that near the edges of the reconstructed source function, there are reduced values. For instance, suppose we start with a 7-pixel image, where every pixel has the value 1, and we scale up to a 20-pixel image. Figure 19.6 shows the seven pixels in a black stem plot, with the 20 pixels drawn in a connected green path on top of them. For pixels near the edges, ringing gives values that are greater than 1, and very near the edges the values are close to 0.

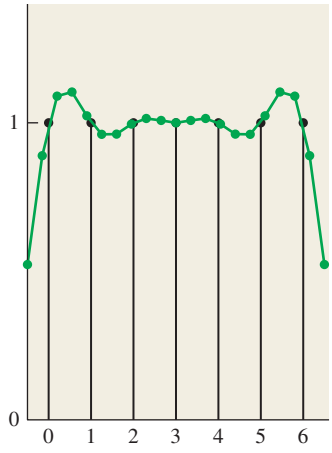


Figure 19.6: Reconstructing 20 samples from seven samples, all 1*s.*

There are five solutions, shown schematically in Figure 19.7, none of them perfect.

1. Extend by zeroes, which we've used so far.
2. Extend by reflection.
3. Extend by constants.
4. Limit the reconstruction filter to finite support, and use one of the approaches above.
5. Adjust the filter near the edges to ignore missing values.

We already discussed the problem with option 1: If we try to reconstruct a constant image, we get ringing artifacts at the edges as the band-limited function tries to drop to zero as quickly as possible. The benefit, however, is clear: We can limit our infinite summation to a finite one.

Option 2, in which we "hallucinate" some values for pixels outside the source image, fails to produce an L^2 function, for if we reflect the source image each time we reach an edge, we create a tiling of the plane by copies of the source image; the L^2 norm of this is infinitely many times the L^2 norm of the source image, that is, ∞.

Option 3 means that as you examine one row of the image and run off the right-hand side of the image, you simply reuse the last pixel in the row as the value of all subsequent pixels, and you do the corresponding thing for the left, top, and bottom edges, and even the corners. This too leads to a signal that's not in L^2.

Although options 2 and 3 lead to signals that are not in L^2, one solution is to say that reconstruction with the sinc is unrealistic: How can a sample at some point that's miles away affect the value at a point within the image? Indeed, since the effect falls off as the inverse distance, that miles-away point will tend to have very little impact on the reconstruction. We can replace the sinc filter with some new filter g that looks like sinc but has finite support, and hope that its Fourier

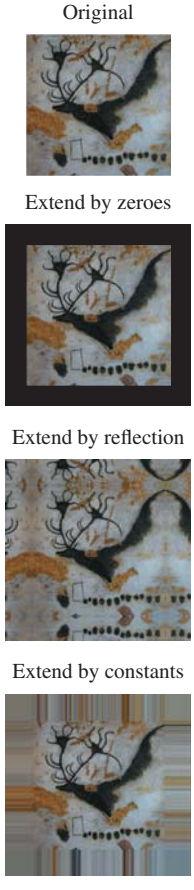


Figure 19.7: Image-extension options.

transform looks like the box function b. Unfortunately, it's impossible for a function $f \in L^2(\mathbf{R})$ to have finite support and have $\hat{f}$ also have finite support [DM85]. But we *can* find finitely supported filters whose transforms are *nearly* zero outside a small interval, as we'll discuss presently. If we agree to replace sinc by such a filter, then the whole question of whether the extension of the source image is L^2 disappears: We can simply extend the source image by an amount R that's greater than the support of the filter, and then fill in with zeroes beyond there. This is option 4, listed above.

Finally, option 5 presents one last approach that's somewhat unprincipled, but works well in practice. When we compute the value for a pixel with a summation of the form

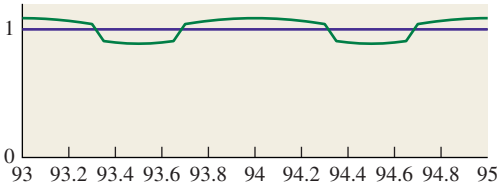
$$target(j) = \sum_i source(i)\text{sinc}(j - i), \tag{19.7}$$

we're computing a weighted sum of source pixels. In this particular case, the weights come from the sinc function, but more generally we're computing something of the form

$$target(j) = \sum_i source(i)w(j - i), \tag{19.8}$$

where w is a list of weights. If these weights don't sum to one, then filtering a *constant* image results in an image whose values are different from that constant. An example, based on filtering in the horizontal direction with a truncated sinc, is shown in Figure 19.8. The constant signal is shown in blue; it was sampled at integer points and reconstructed with $x \mapsto 1.4\text{sinc}(1.4x)$, truncated for $|x| > 3.5$. The resultant function was sampled far more finely and plotted in green, showing substantial ripple.

Typically, for things like a truncated sinc, the sum W of the weights is very close to one at each pixel. The sum may vary from one pixel to the next, however, resulting in a ripple when we try to reconstruct a constant image, as the figure shows. We can fix this problem somewhat by dividing by the sum of the weights at each pixel, which removes the ripple. In effect, we've altered the filter so that its Fourier transform is zero at all multiples of the sampling frequency. We've



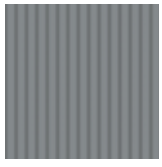


Figure 19.8: Top: Filtering a constant 1D signal with a truncated sinc *leads to ripple. Bottom: A gray image, filtered and resampled with the same truncated* sinc *in the horizontal direction, ends up striped.*

done so at the cost of modifying the filter, however, so its Fourier transform is no longer what we expected. As long as the ripple is small, though, the alteration to the function is small, and hence the alteration to the transform is small as well.

The useful characteristic of this unrippling operation is that it suggests a way to deal with the missing data at the edges of images as well: We compute the sum

$$target(j) = \sum_i source(i)w(j-i), \tag{19.9}$$

but simply ignore all terms where $source(i)$ is outside the image. We sum up the weights as usual, but *only for the terms we included,* and then divide by this sum. In the interior of the image, every pixel we need is available and we're just unrippling. At the edges, we end up estimating the pixel values near the edge based only on the image pixels near the edge, and not on some hallucinated values on the other side of the edge. This method works surprisingly well in practice.

19.5 Finite-Support Approximations

We've said that the ideal reconstruction filter is sinc, but that it's often necessary to compromise in practice and use a finite-support filter. We've already seen the box and tent filters, and that the tent is the convolution $b \star b$ of the box with itself. We can further convolve to get smoother and smoother filters as discussed extensively in Chapter 22, which is about the construction of piecewise-smooth curves. The convolution of the tent with itself (or $b \star b \star b \star b$), known as the **cubic B-spline filter** and plotted in Figure 19.9, is shown in Chapter 22 to be

$$b_3(x) = \begin{cases} \frac{1}{6}(3|x|^3 - 6|x|^2 + 4), & 0 \le |x| \le 1 \\ \frac{1}{6}(-(|x|-1)^3 + 3(|x|-1)^2 - 3(|x|-1) + 1), & 1 \le |x| \le 2 \\ 0, & \text{otherwise} \end{cases} \tag{19.10}$$

There is a slight difference from the function in Chapter 22, however: This chapter's version of b_3 has support $[-2, 2]$, while in Chapter 22 the support is $[0, 4]$.

Notice that $b_3(0) = \frac{2}{3}$, while $b_3(\pm 1) = \frac{1}{3}$. This means that when we reconstruct an image using b_3, the reconstructed value at the image points is not the original image value, but a blend between it and its two neighbors.

An alternative is the Catmull-Rom spline (see Chapter 22 for details), γ_{CR}, which is zero at every integer except $\gamma_{CR}(0) = 1$, as shown in Figure 19.10. This means that reconstructing with this filter leaves the image points unchanged, and merely interpolates values between them. On the other hand, the Catmull-Rom spline takes on negative values, which means that interpolating with it may produce negative results, which is a problem: We can't have "blacker than black" values. The formula for the Catmull-Rom spline is

$$\gamma_{CR}(x) = \frac{1}{2}\begin{cases} -3(1-|x|)^3 + 4(1-|x|)^2 + (1-|x|), & -1 \le x \le 1 \\ (2-|x|)^3 - (2-|x|)^2, & 1 \le |x| \le 2 \\ 0, & \text{otherwise} \end{cases} \tag{19.11}$$

A $\frac{2}{3} - \frac{1}{3}$ blend of the Catmull-Rom curve and the B-spline curve was recommended by Mitchell and Netravali [MN88] as the best all-around cubic to use for image reconstruction and resampling. That filter (shown in Figure 19.11) is given by

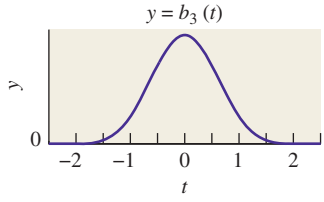


Figure 19.9: The cubic B-spline filter.

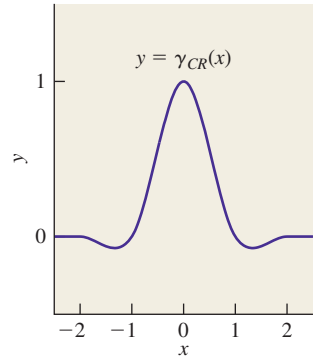


Figure 19.10: The Catmull-Rom spline filter.

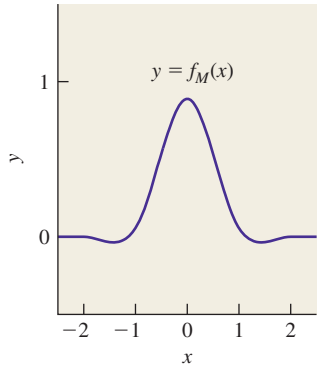


Figure 19.11: The Mitchell-Netravali filter.

$$f_M(x) = \frac{1}{18}\begin{cases} -21(1-|x|)^3 + 27(1-|x|)^2 + 9(1-|x|) + 1, & -1 \le x \le 1 \\ 7(2-|x|)^3 - 6(2-|x|)^2, & 1 \le |x| \le 2 \\ 0, & \text{otherwise} \end{cases} \tag{19.12}$$

Table 19.1 shows several filters and their Fourier transforms.

19.5.1 Practical Band Limiting

The function sinc is band-limited at $\omega_0 = \frac{1}{2}$. The cubic B-spline filter b_3 is *not,* and it passes frequencies much larger than $\frac{1}{2}$, albeit attenuated. As we showed in Chapter 18, the consequence of this is aliasing. A signal at frequency $0.5 + a$ appears as an alias at frequency $0.5 - a$. In particular, integer-spaced samples of a signal of frequency near 1 look like samples of a signal of frequency near 0. Indeed, any near-integer frequency aliases to a low frequency.

If we use a filter like b_3 to reconstruct a continuum signal from image data, the resultant signal will contain aliases, because the support of $\hat{b}_3$ is not contained in $H = (-\frac{1}{2}, \frac{1}{2}]$. Fortunately, outside H the Fourier transform falls off fairly rapidly; nonetheless, there is some aliasing.

We can address this with a compromise: We can stretch b_3 along the x-axis so that the Fourier transform compresses along the ω-axis, pushing more of the support inside H and reducing aliasing. This has the unfortunate consequence that frequencies that we'd like to preserve, near $\pm\frac{1}{2}$, end up attenuated. The compromise involved is one of trading off blurriness (i.e., a lack of frequencies near $|\omega| = \frac{1}{2}$) against low-frequency aliasing (i.e., frequencies near $|\omega| = 1$). For frequencies in between these extremes, aliasing still occurs. But if a frequency just above $\frac{1}{2}$ aliases to one just below $\frac{1}{2}$, it's often removed during image shrinking anyhow, and in practice turns out to not be as noticeable as an alias at a frequency near 0.

19.6 Other Image Operations and Efficiency

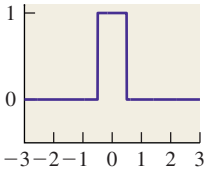
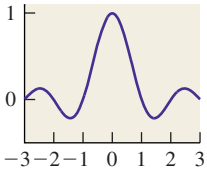
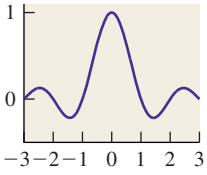
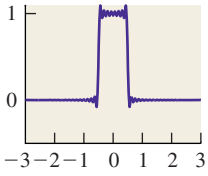
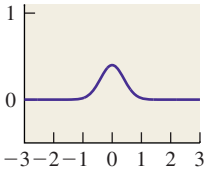
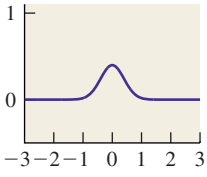
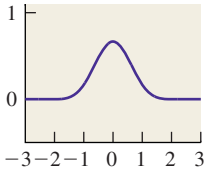
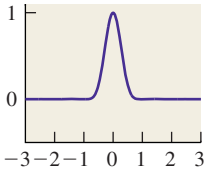
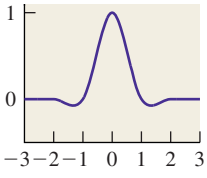
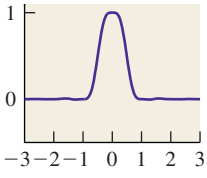
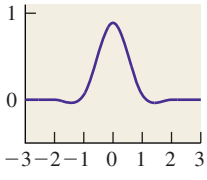
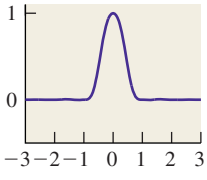
In general, convolving an image with a discrete filter of small support can be an interesting proposition. We already saw how convolving with a 3×3 array of ones can blur an image (although to keep the mean intensity constant, you need to divide by nine).

In general, we can store a discrete filter in a $k \times k$ array `a`, and the image in an $n \times n$ array `b`; the result of the convolution will then be an $(n+k) \times (n+k)$ array `c`; the code is given in Listing 19.4.

Listing 19.4: Convolving using the "extend by zeroes" rule.

```
1 void discreteConvolve (float a[k][k], float b[n][n], float c[n+k-1][n+k-1])
2    initialize c to all zeroes
3    for each pixel (i,j) of a
4       for each pixel (p,q) of b
5          row = i+p;
6          col = j+q;
7          if (row < n+k-1) && (col < n+k-1))
8             c[row][col] += a[i][j] * b[p][q];
```

Table 19.1: Filters and their Fourier transforms.

Comments	$y = f(x)$	$y = \mathcal{F}(f)(\omega)$
The unit box filter; transform is sinc.		
The sinc filter with spacing one; transform is the unit box.		
The Gaussian filter. Transforms to a scaled Gaussian. Shown is $g(\sqrt{\pi}x)$, which transforms to itself.		
The cubic B-spline filter. Fairly band-limited, but attenuates signals except near $\omega = 0$.		
The Catmull-Rom filter. Less band-limited, but better signal preservation near $\omega = 0$.		
Mitchell-Netravali. A compromise between the B-spline and the Catmull-Rom filters.		

A much nicer blur than the one we saw with the box filter comes from convolving with a Gaussian filter, that is, samples of the function $g_\sigma(x) = \frac{1}{\sqrt{2\pi\sigma}} \exp(-\frac{x^2+y^2}{\sigma^2})$, where σ is a constant that determines the amount of blurring: If σ is small, then the convolution will be very blurry; if it's large, there will be almost no blurring. The value $\sigma = 1$ produces a bit less blurring than convolving with a 3×3 array of ones. You can also blur preferentially in one direction or another, using a filter defined by

$$f(x,y) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\begin{bmatrix} x & y \end{bmatrix} \mathbf{S} \begin{bmatrix} x \\ y \end{bmatrix}\right), \tag{19.13}$$

where $\mathbf{S}$ is any symmetric 2×2 matrix. The axes of greatest and least blurring are the eigenvectors of $\mathbf{S}$ corresponding to the least and greatest eigenvalues, respectively. The amount of blur is inversely related to the magnitude of the eigenvalues.

If B is any blurring filter, and your image is I, then $B \star I$ is a blurred version of I; roughly speaking, convolution with B must remove most high frequencies from I, leaving the low-frequency ones. This means that if we compute $I - rB \star I$ for some small $r > 0$, we'll be removing the blurred version and should leave behind a sharpened version. Unfortunately, this also darkens the image: If I initially contains all ones, then all entries of $I - rB \star I$ will be $1 - r$. We can compensate by using

$$S_r = (1 + r)I - rB \tag{19.14}$$

to sharpen the image. In the case where B is the 3×3 box

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{19.15}$$

the result is

$$\frac{1}{9}\begin{bmatrix} -r & -r & -r \\ -r & 9+8r & -r \\ -r & -r & -r \end{bmatrix}. \tag{19.16}$$

The results, using $r = 0.6$, are shown in Figure 19.12, where the blur and sharpening have been applied to a very low-resolution version of the *Mona Lisa,* magnified so that you can see individual pixels.

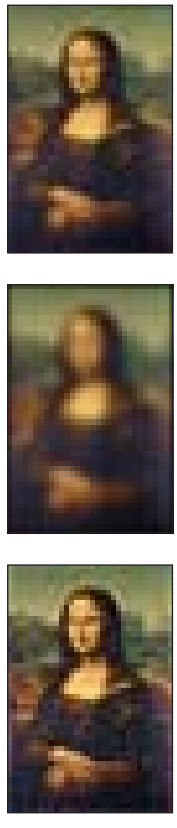


Figure 19.12: Mona Lisa, *blurred and sharpened.*

Inline Exercise 19.3: Verify this expression for the sharpening filter.

You can apply this idea to any blurring filter B to get an associated sharpening filter.

If we convolve an image I with the 1×2 filter $\begin{bmatrix} 1 & -1 \end{bmatrix}$, it will turn any constant region of I into all zeroes. But if there's a vertical edge (i.e., a bright pixel to the left of a dark pixel), the convolution will produce a large value. (If the bright pixel is to the right of the dark one, it will produce a large negative value.) Thus, this filter serves to detect (in the sense of producing nonzero output) vertical edges. A similar approach will detect horizontal edges. And using a wider filter, like $\begin{bmatrix} 1 & 1 & 1 & -1 & -1 & -1 \end{bmatrix}$, will detect edges at a larger scale, while

producing relatively little output for small-scale edges. By computing both the vertical and horizontal "edge-ness" for a pixel, you can even detect edges aligned in other directions. In fact, if we define

$$H = \begin{bmatrix} -1 & 1 \end{bmatrix} \star I \text{ and} \tag{19.17}$$

$$V = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \star I, \tag{19.18}$$

then $\begin{bmatrix} H(i,j) & V(i,j) \end{bmatrix}$ is one version of the **image gradient** at pixel (i,j)—the direction, in index coordinates, in which you would move to get the largest increase in image value.

The images H and V are slightly "biased," in the sense that H takes a pixel to the *right* and subtracts the current pixel to estimate the change in I as we move horizontally, but we could equally well have taken the current pixel minus the pixel to the *left* of it as an estimate. If we average these two computations, the current pixel falls out of the computation and we get a new filter, namely $\frac{1}{2}\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$. This version has the advantage that the value it computes at pixel (i,j) more "fairly" represents the rate of change at (i,j), rather than a half-pixel away. Figure 19.13 shows the blurred low-resolution *Mona Lisa,* the result of $\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$-based edge detection along rows and along columns, and a representation of the gradient computed from these. (We've trimmed the edges where the gradient computation produces meaningless results.)

For more complex operations like near-perfect reconstruction, or edge detection on a large scale, we need to use quite wide filters, and convolving an $N \times N$ image with a $K \times K$ filter (for $K < N$) takes about K^2 operations for each of the N^2 pixels, for a runtime of $O(N^2K^2)$. If the $K \times K$ filter is **separable**—if it can be computed by first filtering each row and then filtering the columns of the result—then the runtime is much reduced. The row filtering, for instance, takes about K operations per pixel, for a total of N^2K operations; the same is true for the columns, with the result that the entire process is $O(N^2K)$, saving a factor of K.

Figure 19.13: Mona Lisa, *row-wise edge detection, column-wise edge detection, and a vector representation of the gradient.*

19.7 Discussion and Further Reading

It's clear that aliasing—the fact that samples of a high-frequency signal can look just like those of a low-frequency signal—has an impact on what we see in graphics. Aliasing in line rendering causes the high-frequency part of the line edge to masquerade as large-scale "stair-steps" or jaggies in an image. Moiré patterns are another example. But one might reasonably ask, "Why, when the eye is presented with such samples, which *could* be from either a low- or a high-frequency signal, does the visual system tend to interpret it as a low one?" One possible answer is that the reconstruction filter used in the visual system is something like a tent filter—we simply blend nearby intensities together. If so, the preferred reconstruction of low frequencies rather than high frequencies is a consequence of the rapid falloff of the Fourier transform of the tent. Of course, this discussion presupposes that the visual system is doing some sort of *linear* processing with the signals it receives, which may not be the case. At any rate, it's clear that without perfect reconstruction, even signals *near* the Nyquist rate can be reconstructed badly, so it may be best, when we produce an image, to be certain that it's band-limited

further than required by the pure theory. The precise tuning of filter choices remains something of a dark art.

Digital signal processing, like the edge detection, blurring, and sharpening mentioned in this chapter, is a rich field. Oppenheim and Schafer's book [OS09] is a standard introduction. Many of the obvious techniques don't work very well in practice. Our example of edge detection and gradient finding with the *Mona Lisa* was carefully applied to a blurred version of the image, because the unblurred version generated many edges and the gradients appeared random. Such tricks form an essential part of the tool set of anyone who works with image data.

19.8 Exercises

◈ **Exercise 19.1:** Suppose you're building a ray tracer and you want to build a 100×100 grid on the image plane. The image plane has uv-coordinates on it, and the image rectangle ranges from $-\frac{1}{2}$ to $\frac{1}{2}$ in u and v. Where should you place your samples? Consider the one-dimensional problem instead: We need 100 samples between $u = -\frac{1}{2}$ and $u = \frac{1}{2}$. One choice is $u_i = -\frac{1}{2} + (i/99), i = 0, \ldots, 99$. These points range from $u_0 = -\frac{1}{2}$ to $u_{99} = \frac{1}{2}$. The other natural choice is to space the points evenly so that they are separated by $1/100$, that is, $u_i = -\frac{1}{2} + (i/100) + (1/200)$.

(a) Suppose that instead of 100 points, you want $N = 3$ points. Plot the u_i for each method.

(b) Do the same for $N = 2$ and $N = 1$.

(c) Imagine that instead of 100 points ranging from $-\frac{1}{2}$ to $\frac{1}{2}$, you wanted 200 points ranging from -1 to 1. You might expect the "middle" 100 points from this wider problem to correspond to your 100 points for the narrower problem. Which formula has that property?

Exercise 19.2: One favorite filter is the Gaussian, defined by $g(x, y) = Ce^{-\pi(x^2+y^2)}$. It has three important properties. First, it is its own Fourier transform. Second, it's circularly symmetric: Its value on the circle of radius r around the origin is constant. Third, it's a product of two one-dimensional filters. Do the algebra to show the second and third properties.

◈ **Exercise 19.3:** The cubic spline filter B can be converted into a circularly symmetric filter on the plane by saying $\overline{B}(x, y) = B(\sqrt{x^2 + y^2})$. Unfortunately, this circularly symmetric filter is not separable. The separable filter $C(x, y) = B(x)B(y)$ built from B is not circularly symmetric. How different are they? Numerically estimate the integral of $C(x, y) - \overline{B}(x, y)$ over the plane.

Exercise 19.4: Apply the unrippling approach, with the missing-weights modification, on the cubic B-spline filter, applied to scaling up a signal by a factor of 10.

(a) Apply the technique to a ten-sample signal where every sample value is 1.

(b) Apply it to a ten-sample signal where sample i is $\cos(\pi ki/20)$ for $k = 1, 4$, and 9. Comment on the results.

Exercise 19.5: Here's an alternative approach to reconstructing a band-limited signal from its samples.

(a) Argue why the operation must be well defined (i.e., why there is exactly one signal in $L^2(\mathbf{R})$, band-limited at $\omega_0 = \frac{1}{2}$, with any particular set of values on the integers).

(b) Argue that reconstruction must be linear, that is, if $f_i : \mathbf{R} \to \mathbf{R}$ is the reconstruction of the discrete signal $s_i : \mathbf{Z} \to \mathbf{R}$ for $i = 1, 2$, then $f_1 + f_2$ is the reconstruction of the discrete signal $s_1 + s_2$.

(c) Argue that if for some k, $s_3(n) = s_1(n+k)$ for every $n \in \mathbf{Z}$, then the reconstruction f_3 of s_3 is just $f_3(x) = f_1(x+k)$. We say that reconstruction is **translation equivariant.**
(d) Explain why the reconstruction of the signal s with $s(0) = 1$, and $s(n) = 0$ for all other n, is just $f(x) = \text{sinc}(x)$.
(e) From your answers to parts (a) through (d), conclude that reconstruction for *any* discrete signal comes from convolution with sinc.

Exercise 19.6: Suppose we band-limit $f(x) = \text{sinc}(x)$ to $\omega_0 = \frac{1}{4}$. Describe the resultant signal precisely.

Exercise 19.7: We suggested that one bad way to downsample a 300×300 image to a 150×150 image was to simply take pixels from every second row and every second column. Suppose the source image is a checkerboard: Pixel (i, j) is white if $i + j$ is even, and it's black if it's odd. At a distance, this image looks uniformly gray.
(a) Show that if we choose pixels from rows and columns with odd indices, the resultant subsampled image is all white.
(b) Show that if we use odd row indices, but even column indices, the resultant subsampled image is all black.

Chapter 20

Textures and Texture Mapping

20.1 Introduction

As we said in Chapter 1, **texture mapping** can be used to add detail to the appearance of an object. Texture mapping has little to do with either texture (in the sense of the roughness or smoothness you feel when you touch something) or maps; the term is an artifact of the early history of graphics.

Why is it called texture mapping? Informally speaking, a long time ago small details of models, whether geometric or color-related, were represented in the model of the object by "painting" them onto the model, as in a trompe l'oeil painting. You could paint a tiny bright spot on a model to look like a highlight, regardless of the lighting in the scene. These details were called texture, and they were stored in an image array. The scene modeler also had to associate each vertex of the model to a location in the image array, thus "mapping" the model to the image (although the general goal was just the opposite—this mapping was used to "apply" the image to the model, like a decal). Soon it became clear that rather than mapping the **albedo** (i.e., the fraction of power reflected) we could map other parameters of the lighting model, like the normal vector. Varying the normal vector made the model appear rippled or bumpy; in other words, the appearance of *texture*. But the word "texture" was already used to mean something different, so this was called bump mapping. Later it became clear that we could also store small variations in surface *position* in a map, and while this should have been called bump mapping, since it actually added bumps to a surface, that term was in use, so it was called displacement mapping. Finally, when programmable GPUs were becoming common, their texture memory was the only randomly addressable data structure available to the programmer, and it was often used to store n-dimensional arrays, pointers,

or other things (i.e., it was treated as ordinary memory). Thus, the particular meaning of "texture" is rather time-dependent; when you read a paper on the subject, you'll need to know when it was written to know what the term means.

Figure 20.1: Texture image for soda can (Courtesy of Kefei Lei).

A typical use of texture mapping is to make something that looks painted, like a soft-drink can. First you make a 2D image I that looks like an unrolled version of the vertical sides of the can (see Figure 20.1). Then you give the image coordinates u and v that range from 0 to 1 in the horizontal and vertical directions. Then you model a cylinder, perhaps as a mesh of a few hundred polygons based on vertex locations like

$$P_{ij} = \left(r\cos\frac{2\pi i}{10}, h\frac{j}{5}, r\sin\frac{2\pi i}{10} \right), \tag{20.1}$$

where r is the can's radius, h is its height, and $i = 0, \ldots, 10$ and $j = 0, \ldots, 5$ (see Figure 20.2). A typical triangle might have vertices P_{11}, P_{12}, and P_{21}.

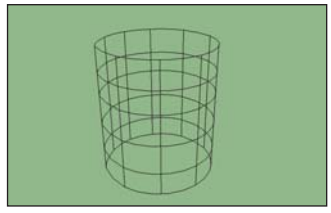


Figure 20.2: A wireframe rendering of the vertical surface of the soda can (Courtesy of Kefei Lei).

Now you *also* assign to each vertex so-called uv-coordinates: a u- and a v-value at each vertex. In this example, the u-coordinate of vertex P_{ij} would be $i/10$ and the v-coordinate would be $j/5$. Notice that the vertices $P_{0,0}$ and $P_{10,0}$ are in identical locations (at the "seam" of the can), but they have different uv-"coordinates." Because coordinates should be unique to a point (at least in mathematics), it might make more sense to call these uv-"values," but the term "coordinates" is well established. We will, however, refer henceforth to **texture coordinates** rather than "uv-coordinates," both because sometimes we use one or three rather than two coordinates, and because the tying of concepts to particular letters of the alphabet can be problematic, as in the case where a single mesh has two different sets of texture coordinates assigned to it.

When it comes time to render a triangle, it gets rasterized, that is, broken into tiny fragments, each of which will contribute to one pixel of the final result. The coordinates of these fragments are determined by interpolating the vertex coordinates, but at the same time, the renderer (or graphics card) interpolates the texture coordinates. Each fragment of a triangle gets different texture coordinates. During the rendering step in which a color is computed for the fragment, often based on the incoming light, the direction to the eye, the surface normal, etc., as in the Phong model of Chapter 6, the material color is one of the items needed in the computation. Texture mapping is the process of using the texture coordinate for the fragment to look up the material color in the image I rather than just using a fixed color. Figure 20.3 shows the effect.

Figure 20.3: The sides of the soda can texture-mapped with the image (Courtesy of Kefei Lei).

We've omitted many details from this brief description, including a step in which fragments are further reduced to samples, but it conveys the essential idea, which has been generalized in a great many ways.

A value (e.g, the color) associated to a fragment of a triangle is almost always the result of a computation, one that has many parameters such as the incoming light, the surface normal, the vector from the surface to the eye, the surface color (or other descriptions of surface scattering like the bidirectional reflectance distribution function or BRDF), etc. Ordinarily, many of these parameters either are constants or are computed by interpolating values from the triangle's vertices. If instead we barycentrically interpolate some texture coordinates from the triangle vertices, these coordinates can be used as arguments to one or more *functions* whose values are then used as the parameters. A typical function is "look

up a value at location (u, v) in an image whose domain is parameterized by $0 \leq u, v \leq 1$," but there are many other possibilities.

Much of this chapter elaborates on this idea, discussing the parameters that can be varied, the codomains for texture coordinates (i.e., the meaning of the values that they take on), and mechanisms for defining mappings from a mesh to the space of coordinates.

You'll notice that this entire notion of "texture mapping" is really just a name for *indirection* as applied in a specific context: We use an index (the texture coordinates) to determine a value. This determination may be performed by a table lookup (as in the case of the soft-drink can above, where the texture image serves as the "table") or by a more complex computation, in which case it's often called **procedural texturing.**

An example of the power of various mapping operations is given by this book's cover image, in which almost every object you see has had multiple maps applied to it—color, texture, displacement, etc.—resulting in a visual richness that would be almost impossible to achieve otherwise.

20.2 Variations of Texturing

We'll illustrate several variations on the idea of texture mapping, working with the Phong reflection model as an exemplar. The ideas we present are largely independent of the reflection model, and they apply more generally. Because the unit square $0 \leq u, v \leq 1$ occurs so often in this chapter, we'll give it a name, U, which we'll use in this chapter only.

Recall that in the Phong model of Chapter 6, the light scattered from a surface is defined by various constants (the diffuse reflection coefficient k_d, the specular reflection coefficient k_s, the diffuse and specular colors C_d and C_s, and the specular exponent n), dot products of various unit vectors, including the direction vector to the light source, the direction vector from the surface to the eye, the surface normal vector, and finally, the arriving light. There is also, in some versions of the model, an ambient term, modeled by k_a and C_a, indicating an amount of light emitted by the surface independent of the arriving light. Each of these things—the constants, the vectors, and even the way you compute the dot product—is a candidate for mapping.

20.2.1 Environment Mapping

If our surface is mirrorlike (i.e., $k_d = k_a = 0$, and n is very large, or even infinite), then the light scattered toward the eye, as determined by the Phong model, is computed by reflecting the eye ray through the surface normal to get a ray that may point toward a light source (in which case light is scattered) or not. If all sources are point sources, this tends to result in no rendered reflection at all, since the probability of a ray hitting any particular source is zero. If the sources are area sources, we see them reflected in our object.

We can replace our model of light arriving at the surface point P from one based on a few point or area sources to one based on a texture lookup: We can treat the reflected eye vector as a point of the unit sphere, and use it to index into a texture-mapped sphere to look up the arriving light. In practical terms, if we write this eye vector in polar (world) coordinates, (θ, ϕ), for instance, we can use $u = \frac{\theta}{2\pi}$, $v = \frac{\phi}{\pi} + \frac{1}{2}$ to index into an **environment map** that contains, at location (u, v), the light arriving from the corresponding direction.

Notice that as the point P moves, the light arriving at the point P from direction $\mathbf{v}$ doesn't change. Thus, this **environment map** is useful for modeling the specular reflection of an object's surroundings, but it is not good for modeling reflections of nearby objects, where the direction from P to some object will change substantially as we vary P.

Where do we get an environment map in the first place? A fisheye-lens photograph taken from the center of a scene can provide the necessary input, although the mapping from pixels in the photograph to pixels in the environment map requires careful resampling. In practice, it's common to use multiple ordinary photographs, and a rather different mapping strategy from the one we've suggested here, but the key idea—doing a lighting "lookup" rather than iterating over a small list of point or area lights—remains the same.

Debevec has written an interesting first-person history of reflection mapping in general [Deb06], tracing its first use in graphics to a paper by Blinn and Newell in 1976 [BN76], which is long before digital photographs were available! The "environment map" in this case was a scene created in a drawing program.

Inline Exercise 20.1: We've carefully suggested using environment mapping in a ray tracer for describing the lighting of a mirrorlike surface. What would be entailed in using environment mapping on a glossy surface? A diffuse one? Would it substantially increase the computation time over that of a simpler model in which the light was specified by a few point and area lights?

20.2.2 Bump Mapping

In **bump mapping,** we fiddle with another of the ingredients in the Phong model: the surface normal, $\mathbf{n}$. A typical version of this uses texture coordinates on a model to look up a value in a bump map image and uses the resultant values to alter the normal vector a little.

The exact meaning assigned to the RGB values from this image depends on the implementation, but a simple version uses just R and G to "tilt" the normal vector as follows. We'll assume that at each surface point P we have a pair of unit vectors $\mathbf{t}_1$ and $\mathbf{t}_2$ that are tangent to the surface, are mutually perpendicular, and vary continuously across the surface. In theory, it may be impossible to find such a pair of vector fields (see Chapter 25), and in Section 20.3 we'll discuss this further, but in practice usually only the continuity assumption is violated, and only at a few isolated points. For example, on the unit sphere, tangents to the lines of constant latitude and constant longitude can play the role of $\mathbf{t}_1$ and $\mathbf{t}_2$, failing the continuity requirement only at the north and south poles. If we arrange to not perturb the normal vector at either pole, then the lack of continuity of $\mathbf{t}_1$ and $\mathbf{t}_2$ has no effect.

With $\mathbf{t}_1$ and $\mathbf{t}_2$ in hand, we'll show how to adjust the normal vector. The R and G values from the bump map, typically bytes representing values $-128, \ldots, 127$, or unsigned bytes representing 0 to 255, are adjusted to range from -1 to 1 by writing (in the first case)

$$r = \max\left(\frac{R}{127}, -1.0\right) \tag{20.2}$$

and a corresponding expression for g. (The loss of -128 as a distinct value is deliberate. If we'd divided by 128, we could not represent $+1.0$.)

Inline Exercise 20.2: Write a formula to convert values from 0 to 255 into values from -1 to 1.

We can now adjust the normal vector $\mathbf{n}$, as shown in Figure 20.4, by computing

$$\mathbf{n}' = \mathcal{S}(\mathbf{n} + r\mathbf{t}_1 + g\mathbf{t}_2). \tag{20.3}$$

At the extreme (when $r = g = 1$) this tilts the normal vector by about 54°. If a larger amount of tilt is needed, one can redefine r and g to range from -2 to 2, or even more.

One advantage to this scheme, originally proposed by Blinn [BN76], is that $\mathbf{n} + r\mathbf{t}_1 + g\mathbf{t}_2$ can never be zero, because its dot product with $\mathbf{n}$ is one. This means that it can always be normalized.

By the way, the particular form we've described for converting image-pixel values into values between -1.0 and 1.0 is part of the OpenGL standard and is called the **signed normalized fixed-point 8-bit representation** (see Chapter 14). There are other standard representations as well, using more bits per pixel, unsigned rather than signed values, etc. There's nothing sacred about any of these. They've proven to be convenient over the years, so they become standardized.

A more direct approach is to store in the texture image three floating-point numbers (i.e., we treat the bits in the red channel as a float, and do the same for green and blue), and use this triple of numbers as the coordinates of the (unnormalized) normal vector, from which we can compute $\mathbf{n}$; this is one of the most common approaches today.

It's possible to specify a set of normal vectors that are not actually consistent with each other, that is, they do not form the normal vector field for any surface (see Exercise 20.2). When rendered, these can look peculiar. So a third approach to bump mapping is to have the texture contain a *height* for each texture coordinate (u, v), indicating that we are to imagine that the surface is displaced from its current position by that amount along the original normal vector. The resultant surface (which is never actually created!) has tangent vectors in the u- and v-directions (i.e., the direction of greatest increase of u, and correspondingly of v), whose cross product we compute and use as the "normal" vector during shading computations. This clearly requires substantially more computation, but far less bandwidth, since we need only one value per location rather than two or three.

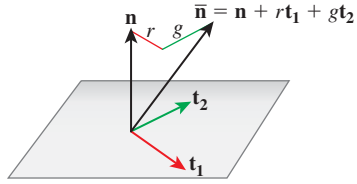


Figure 20.4: The vector $\bar{\mathbf{n}}$ will be normalized to produce the new normal $\mathbf{n}'$; the values r and g (for "red" and "green") vary across the surface, and are looked up in a texture map.

20.2.3 Contour Drawing

This example differs from the previous ones, because each surface point will have only *one* texture coordinate. A point P of a smooth surface is on a **contour** if the ray from P to the eye E, $\mathbf{r} = E - P$, is perpendicular to the surface normal $\mathbf{n}$. Thus, to render a surface by drawing its contours, all we must do is compute this dot product, and when it's near zero, make a black mark on the image, or leave the image white otherwise. To do so we make a one-dimensional texture map like the one shown in Figure 20.5. And as a texture coordinate, we compute

$$d = \mathbf{r} \cdot \mathbf{n} \tag{20.4}$$

$$u = \frac{d+1}{2}. \tag{20.5}$$

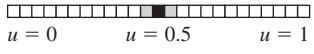


Figure 20.5: A contour-drawing 1D texture.

As the dot product ranges from -1 to 0 to 1, u ranges from 0 to $\frac{1}{2}$ to 1. At $u = \frac{1}{2}$, the dark color we find in the texture map creates a dark spot in the image. To be clear: This "rendering" algorithm involves no lighting or reflection or anything like that. It's simply following the rule that we can draw the contours of an object to make a reasonably effective picture of it. An actual implementation of a slightly fancier version of this algorithm is given in Section 33.8.

20.3 Building Tangent Vectors from a Parameterization

We'll now describe how to find a **frame** (i.e., a basis) for the tangent space at each point of a parameterized smooth surface, and then, working by analogy, describe a similar construction for a mesh.

The sphere, which we'll use as our smooth example, is often parameterized by

$$P(\theta, \phi) = (\cos\theta\cos\phi, \sin\phi, \sin\theta\cos\phi), \tag{20.6}$$

where θ ranges from 0 to 2π, and ϕ ranges from $-\frac{\pi}{2}$ to $\frac{\pi}{2}$.

If we hold θ constant in $P(\theta, \phi)$ and vary ϕ, we get a line of longitude; similarly, if we hold ϕ constant and vary θ, we get a line of latitude. If we compute the tangent vectors to these two curves, we get

$$\frac{\partial P}{\partial \phi}(\theta, \phi) = \begin{bmatrix} -\cos\theta\sin\phi & \cos\phi & -\sin\theta\sin\phi \end{bmatrix}^{\mathbf{T}} \text{ and} \tag{20.7}$$

$$\frac{\partial P}{\partial \theta}(\theta, \phi) = \begin{bmatrix} -\sin\theta\cos\phi & 0 & \cos\theta\cos\phi \end{bmatrix}^{\mathbf{T}}. \tag{20.8}$$

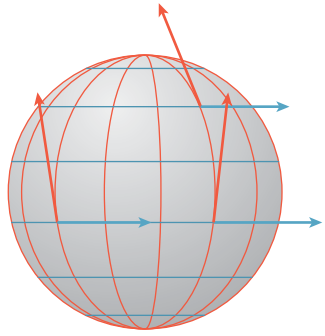


Figure 20.6: The sphere, with vertical lines of constant θ and horizontal lines of constant ϕ drawn in red and blue, respectively, and tangent vectors to those curves at a few points.

These vectors, drawn at the location $P(\theta, \phi)$, are tangent to the sphere at that point, and they happen to be perpendicular as well (see Figure 20.6). Except at the north and south poles (where $\cos(\phi) = 0$), the vectors are nonzero, so the two of them constitute a frame at almost every point. In general, it's topologically impossible to find a smoothly varying frame at every point of an arbitrary surface, so our situation, with a frame at almost every point, is the best we can hope for.

In general, if we have any surface parameterized by a function like P of two variables—say, u and v—then $\frac{\partial P}{\partial u}(u, v)$ and $\frac{\partial P}{\partial v}(u, v)$ are a pair of vectors at the point (u, v) that form a basis for the tangent plane there, except in two circumstances: One of the vectors may be zero, or the two vectors may be parallel. In both situations, the parameterization is degenerate in some way, and for "nice" surface parameterizations this should happen only at isolated points. To get an even nicer framing, you can normalize the first vector and compute its cross product with the normal vector to get the second; the result will be an orthonormal frame at every place where the first vector is nonzero.

We can now proceed analogously on a mesh for which each vertex has xyz-coordinates and uv-texture coordinates assigned. We'll do so one face at a time. The assignment of (u, v) coordinates to each vertex defines an affine (linear-plus-translation) map from the xyz-plane of the triangular face to the uv-plane (or vice versa). We'd like to know what the curve of constant u or constant v looks like,

in analogy with the curves of constant θ and ϕ in the sphere example above, so that we can compute the tangent vectors to these curves. Fortunately, for an affine map a curve of constant v is just a line; all we need to do is find the direction of this line.

Suppose the face has vertices P_0, P_1, and P_2, with associated texture coordinates $(u_0, v_0), (u_1, v_1)$, and (u_2, v_2). We'll study everything relative to P_0, so we define the edge vectors $\mathbf{w}_1 = P_1 - P_0$ and $\mathbf{w}_2 = P_2 - P_0$, and similarly define $\Delta u_i = u_i - u_0$ $(i = 1, 2)$, and similarly for v (see Figure 20.7).

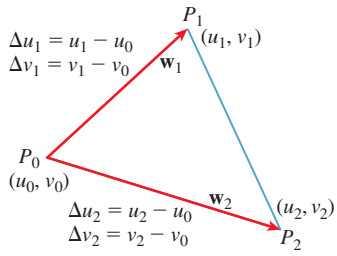


Figure 20.7: Names for computing a line of constant v on a single face.

Since v varies linearly (or affinely, to be precise) along each edge vector, consider the vector $\mathbf{w} = \Delta v_2 \mathbf{w}_1 - \Delta v_1 \mathbf{w}_2$. How much does v change along this vector? It changes by Δv_1 along $\mathbf{w}_1$, so along the first term, it changes by $\Delta v_2 \Delta v_1$; a similar argument shows that on the second term, it changes by $\Delta v_1 \Delta v_2$. Hence on the sum, $\mathbf{w}$, v remains constant. We've found a vector on which v is constant! We can do the same thing for u, so the frame for this triangle has, as its two vectors,

$$\mathbf{f}_1 = \mathcal{S}(\Delta v_2 \mathbf{w}_1 - \Delta v_1 \mathbf{w}_2) \text{ and} \tag{20.9}$$

$$\mathbf{f}_2 = \mathcal{S}(\Delta u_2 \mathbf{w}_1 - \Delta u_1 \mathbf{w}_2). \tag{20.10}$$

Unfortunately, if we perform the same computation for an adjacent triangle, we'll get a different pair of vectors. We can, however, at each vertex of the mesh, average the $\mathbf{f}_1$ vectors from all adjacent faces and normalize, and similarly for the $\mathbf{f}_2$ vectors. We can then interpolate these averaged values over the interior of each triangle. There's always the possibility that either one of the average vectors at a vertex will be zero, or that when we interpolate we'll get a zero at some interior point of a triangle. (Indeed, this will *have* to happen for most closed surfaces except those that have the topology of a torus.) But this is just the piecewise-linear version of the problems we already encountered for smooth maps. If we're using this framing to perform bump mapping, we'll want to avoid assigning a nonzero coefficient at any point at which one of the frame vectors is zero.

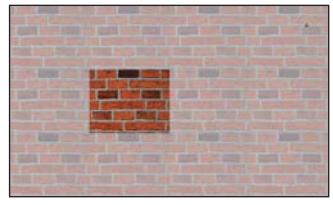


Figure 20.8: The texture image, shown dark, can be replicated across the whole plane (shaded squares) so that texture coordinates outside the unit square can be used.

20.4 Codomains for Texture Maps

The texture values that we define at vertices, and which are interpolated across faces of a triangular mesh, are represented as numbers. When we have two texture coordinates u and v, we're implicitly defining a mapping from our mesh to a unit square in the uv-plane. The codomain of the texture-coordinate assignment in this case is the unit square. There are two generalizations of this.

First, some systems allow texture coordinates to take on values outside the range $U = \{(u, v) | 0 \leq u \leq 1, 0 \leq v \leq 1\}$. Before the coordinates are actually *used,* they are reduced $\bmod 1$, that is, u is converted to $u - \text{floor}(u)$, and similarly for v. The net effect can be viewed in one of two ways.

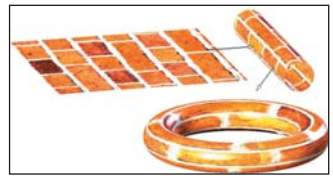


Figure 20.9: The sides of the square are identified to form a cylinder; the ends of the cylinder are then identified to make a torus.

1. The uv-plane, rather than having a single image placed in the unit square, is tiled with the image. Our texture coordinates define a map into this tiled plane (see Figure 20.8).
2. The edges of the unit square defined by the lines $u = 1$ and $u = 0$ are treated as identical; the square is effectively rolled up into a cylinder. Similarly, the lines $v = 1$ and $v = 0$ are identified with each other, rolling up the cylinder into a torus (see Figure 20.9). If you like, you can consider the

point (u, v) as corresponding to the point that's $2\pi u$ of the way around the torus in one direction, and $2\pi v$ around it in the other direction, with the texture stretched over the whole torus. The texture coordinates define a mapping from your object to the surface of a torus.

For either interpretation to make sense, the texture-map values on the right-hand side of the unit square must match up nicely with those on the left (and similarly for the top and bottom), or else the texture will be discontinuous on the $u = 0$ circle of the torus, and similarly for the $v = 0$ circle. Furthermore, any filtering or image processing you do to the texture image must involve a "wraparound" to blend values from the left with those from the right, and from top to bottom as well.

The second generalization was already suggested by the second version of bump mapping: The texture coordinates define a map of your object onto a surface in some higher-dimensional space. In the case of bump mapping, each object point is sent to a unit vector, that is, the codomain is the unit sphere within 3-space.

In general, the appropriate target for texture coordinates depends on their intended use.

Here are a few more examples.

- You can use the *actual* coordinates of each point as texture coordinates (perhaps scaling them to fit within a unit cube first). If you then generate a cubical "image" that looks like marble or wood, and use the texture value as the color at each point, you can make your object look as if it were carved from marble or wood. In this case, your texture uses a lot of memory, but only a small part of it is ever used to color the model. The codomain is a cube in 3-space, but the image of the texture-coordinate map is just the surface of your object, scaled to fit within this cube.
- A nonzero triple (u, v, w) of texture coordinates (typically a unit vector) is converted to $(u/t, v/t, w/t)$ where $t = 2\max(|u|, |v|, |w|)$; the result is a triple with one coordinate equal to $\pm\frac{1}{2}$, and the other two in the range $[-\frac{1}{2}, \frac{1}{2}]$, that is, a point on the face of the unit cube. Each one of the six faces of the cube (corresponding to u, v, or w being $+\frac{1}{2}$ or $-\frac{1}{2}$) is associated to its own texture map. This provides a texture on the unit sphere in which the distortion between each texture-map "patch" and the sphere is relatively small. This structure is called a **cube map,** and it is a standard part of many graphics packages; it's the currently preferred way to specify spherical textures. Alternatives, like the latitude-longitude parameterization of the sphere, are useful in situations where the high distortion near the poles is unimportant (as in the case of a world map, where the area near the poles is all white).
- In the event that the cube map needs to be regenerated often (e.g., if it's an environment map generated by rendering a changing scene from the point of view of the object), rendering the scene in six different views may be more work than you want to do. A natural alternative is to make two hemispherical renderings, recording light arriving from direction $\begin{bmatrix} x & y & z \end{bmatrix}^{\mathbf{T}}$ at position $(u, v) = \left(\frac{x+1}{2}, \frac{z+1}{2}\right)$ in one image for $y \geq 0$ and another for $y \leq 0$. Each of these renderings uses only $\pi/4 \approx 79\%$ of the area of the unit square, but they're very easy to compute and use. (An alternative two-patch solution is the **dual paraboloid** of Exercise 20.4.)

20.5 Assigning Texture Coordinates

How do we specify the mapping from an object to texture space? The great majority of standard methods start by assigning texture coordinates at mesh vertices and using linear interpolation to extend over the interior of each mesh face. We often (but not always) want that mapping to have the following properties.

- *Piecewise linear:* As we said, this makes it possible to use interpolation hardware to determine values at points other than mesh vertices.
- *Invertible:* If the mapping is invertible, we can go "backward" from texture space to the surface, which can be helpful during operations like filtering.
- *Easy to compute:* Every point at which we compute lighting (or whatever other computation uses mapping) will have to have the texture-coordinate computation applied to it. The more efficient it is, the better.
- *Area preserving:* This means that the space in the texture map is used efficiently. Even better, when we need to filter textures, etc., is to have the map be area-and-angle preserving, that is, an isometry, but this is seldom possible. A compromise is a **conformal** mapping, which is angle preserving, so that at each point, it locally looks like a uniform scaling operation [HAT+00].

The following are examples of some common mappings.

- *Linear/planar/hyperplanar projection:* In other words, you just use some or all of the surface point's world coordinates to define a point in texture space. Peachey [Pea85] called these **projection textures.** The wooden ball in Figure 20.10 was made this way: The texture shown at the right on the yz-aligned plane was used to texture the ball by coloring the ball point (x, y, z) with the image color at location $(0, y, z)$.
- *Cylindrical:* For objects that have some central axis, we can surround the object with a large cylinder and project from the axis through the object to the cylinder; if the point (x, y) projects to a point (r, θ, z) in cylindrical coordinates (r being a constant), we assign it texture coordinates (θ, z). More precisely, we use coordinates $u = \frac{\theta}{2\pi}$, $v = \text{clamp}(\frac{z}{z_{\max}}, -1, 1)$, where the $\text{clamp}(x, a, b)$ returns x if $a \leq x \leq b$, a if $x < a$, and b if $x > b$.
- *Spherical:* We can often pick a central point within an object and project to a sphere in much the same way we did for cylindrical mapping, and then use suitably scaled polar coordinates on the sphere to act as texture coordinates.
- *Piecewise-linear or piecewise-bilinear on a plane from explicit per-vertex texture coordinates (UV):* This is the method we've said was most common, but it requires, as a starting point, an assignment of texture coordinates to each vertex. There are at least four ways to do this.
  - Have an artist explicitly assign coordinates to some or all vertices. If the artist only assigns coordinates to some vertices, you need to algorithmically determine interpolated coordinates at other vertices.
  - Use an algorithmic approach to "unfold" your mesh onto a plane in a distortion-minimizing way, typically involving cutting along some (algorithmically determined) seams. This process is called **texture parameterization** in the literature. Some aspects of texture parame-

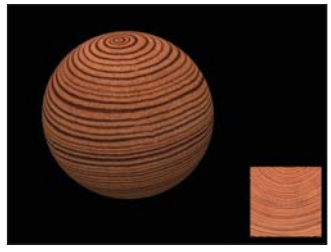


Figure 20.10: A wooden ball made with a projection texture shown at right (Courtesy of Kefei Lei).

terization may also be used in the interpolation of coordinates in the artist-assigned texture-coordinate methods above.

  - Use an algorithmic approach to break your surface into patches, each of which has little enough curvature that it can be mapped to the plane with low distortion, and then define multiple texture maps, one per patch. The cube-map approach to texturing a sphere is an example of this approach. Filtering such texture structures requires either looking past the edge of a patch into the adjacent patch, or using overlapping patches, as mathematicians do when they define manifold structures. The former approach uses textures efficiently, but involves algorithmic complications; the latter wastes some texture space, but simplifies filtering operations.
  - Have an artist "paint" the texture directly onto the object, and develop a coordinate mapping (or several patches as above) as the artist does so. This approach is taken by Igarashi et al. in the Chameleon system [IC01]. In a closely related approach, the painted texture is stored in a three-dimensional data structure so that the point's world-space coordinates serve as the texture coordinates. These are used to index into the spatial data structure and find the texture value that's stored there. Detailed textures are stored by using a hierarchical spatial data structure like an octree: When the artist adds detail at a scale smaller than the current octree cell, the cell is subdivided to better capture the texture [DGPR02, BD02].

- *Normal-vector projection onto a sphere:* The texture coordinates assigned to a point (x, y, z) with unit normal vector $\mathbf{n} = \begin{bmatrix} n_x & n_y & n_z \end{bmatrix}^{\mathbf{T}}$ are treated as a function of n_x, n_y, and n_z, either as the angular spherical polar coordinates of the point (n_x, n_y, n_z) (the radial coordinate is always 1), or by using a cube-map texture indexed by $\mathbf{n}$.

There are even generalizations in which the texture coordinates are not assigned to a point of an object, but instead are a function of the point and some other data. Environment mapping is one of these: In this case the texture coordinates are derived from the reflected eye vector on a mirrorlike surface.

There are also generalizations in which the Noncommutativity principle is applied: Certain operations, like filtering multiple samples to estimate average radiance arriving at a sensor pixel, can be exchanged with other operations, like the reflection computation used in environment mapping, without introducing too much error. If you want to environment-map a nonmirror surface, you'll want to compute many scattered rays from the eye ray, look up arriving radiance for each of them in the environment map, multiply appropriately by a BRDF value and cosine, and average. You can instead start with a different environment map that at each location stores the average of many nearby samples from the original environment map (i.e., a blurred version of the original). You can then push *one* sample of this new map through the BRDF and cosine to get a radiance value: You've swapped the averaging of samples with the convolution of light against the BRDF and cosine. These operations do not, in fact, commute, so the answers you produce will generally be incorrect. But they are often, especially for almost-diffuse surfaces, quite good enough for most purposes, and they speed up the rendering substantially. Approaches like this are grouped under the general term **reflection mapping.**

As an extreme example, for a diffuse surface the only characteristic of the arriving light that matters is irradiance—the cosine-weighted average radiance over the visible hemisphere [RH01]. From an environment map, one can compute the average for every possible direction, and use this **irradiance map** to rapidly compute light reflected from a diffuse surface, assuming that the light arrives from far enough away that the irradiance is a function of direction only. Figure 20.11 shows such an irradiance map, and Figure 20.12 shows a figure illuminated by it.

Inline Exercise 20.3: Describe the appearance of a pure-white, totally diffuse sphere, illuminated by the irradiance map of Figure 20.12, rendered with a parallel projection in the same direction as the view shown for the irradiance map. The answer is *not* that it looks exactly like the irradiance map!

20.6 Application Examples

Now that we've described the general idea of texture mapping (assigning texture coordinates to object points, followed by evaluating functions on these coordinates, often by interpolation of image values), let's look at a range of applications. Table 20.1 presents these, listing for each application the property that is being mapped, the map being used, and the name of the resultant technique. We use the name "UV" to indicate some kind of surface parameterization.

20.7 Sampling, Aliasing, Filtering, and Reconstruction

When we render a scene in which there is texture mapping, no matter how simple or complex the mapping scheme is, problems with sampling and aliasing can arise.

Table 20.1: Mapping applications.

Property	Map	Technique
k_d, diffuse reflectivity	UV	Diffuse detail mapping, like the upholstery pattern on a sofa
k_s, glossy reflectivity	UV	Glossy detail, like the part of a tarnished doorknob that's polished by constant use
L^{in}	Reflection	Environment mapping
L^{out}	UV	**Light mapping.** Texture mapping is used to specify the emissivity (typically diffuse) of an object like a neon lamp.
Position or normal vector	UV	Bump mapping or displacement mapping
Visibility of a light source	Perspective projection	**Shadow mapping** (see Chapter 15)
Artistic L^{out}	Various dot products	XToon shading in expressive rendering (see Section 33.8)

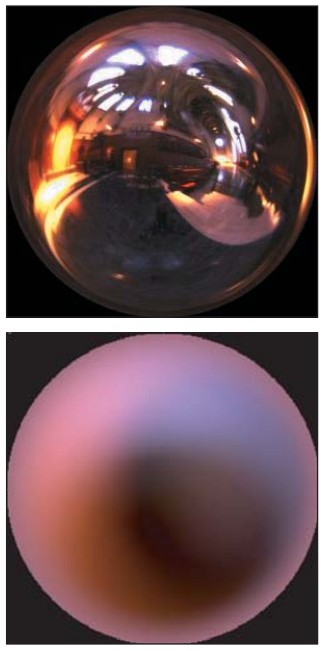


Figure 20.11: Top: A sphere map of light arriving at one point in Grace Cathedral (Photograph used with permission. Copyright 2012 University of Southern California, Institute for Creative Technologies.) Bottom: An irradiance map formed from that sphere map (Courtesy of Ravi Ramamoorthi and Pat Hanrahan, © 2001 ACM, Inc. Reprinted by permission.)

Figure 20.12: Several objects illuminated by the irradiance map, represented in a compact approximation (Courtesy of Ravi Ramamoorthi and Pat Hanrahan, © 2001 ACM, Inc. Reprinted by permission.)

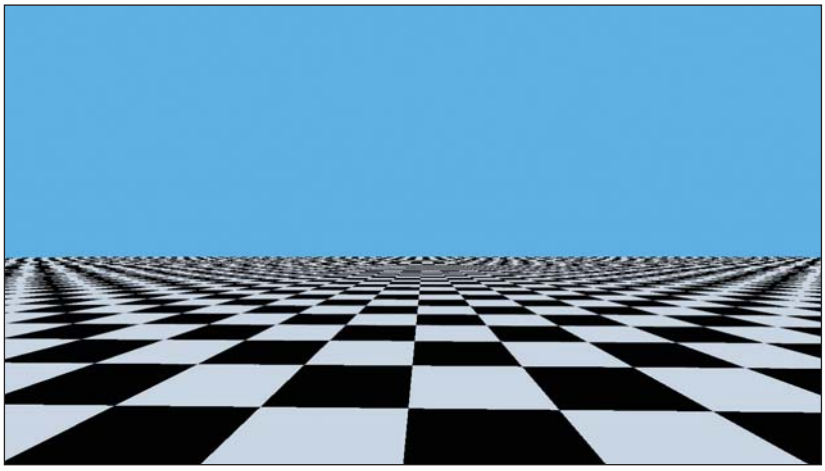


Figure 20.13: Aliasing arising from single-sample ray tracing of a checkerboard texture (Courtesy of Kefei Lei).

Figure 20.13 shows a very simple example, a ray-traced scene in which there's no lighting, but there is a single infinite ground plane and an infinite sky plane. The sky is blue; the ground has a checkerboard pattern. The color for a ray traced into the scene is therefore either blue, black, or white.

Even though the checkerboard texture is absolutely perfect, in the sense that it's represented by a function, namely, "Is floor(x) + floor(y) odd?" rather than interpolated from an image, it's clear that the picture doesn't look very good. The moiré patterns at the horizon are distracting and unnatural.

The reason for these patterns is clear: On any single horizon-parallel line, the texture looks like a square wave; the Fourier transform of the square wave contains arbitrarily high frequencies. And the image-space frequency of this square wave gets higher as we approach the horizon. Nonetheless, we're taking samples at a fixed spacing (one per output pixel, sampled at the pixel center). We're sampling a non-band-limited function, and the degree to which it's not band-limited gets worse as we get closer to the horizon. Naturally, aliases appear.

This doesn't mean that texture mapping is a bad idea by any means. It only tells us that we need to think about sampling before we use texture-mapped values. We need to band-limit the signal before we take samples.

How much must we band-limit? At the very least, to the Nyquist rate for the sampling, that is, a half-period per pixel spacing. On the ground plane itself, the upper limit on frequency varies with distance: If one checkerboard square at distance 5 from the camera projects to a span of 20 pixels left to right, then at distance 50 it will project to just two pixels and at distance 100 it will project to a single pixel, so a black-white cycle (two adjacent squares of the checkerboard) will project to exactly two pixels, which means that at distance 100 even the fundamental frequency of the square wave is at the Nyquist limit. For distances beyond 100, the best we can do is to replace the square wave by its average value (i.e., display a uniformly gray plane). A similar analysis applies in the projection of the checkerboard squares in the vertical direction. Evidently, the band-limit in each direction needs to decrease linearly with distance. Fortunately, this notion is now built into much hardware: It's easy to specify, when you use a texture map, that

the hardware should compute a MIP map for the texture image, and to select the MIP level during texture mapping according to the size of the screen projection of a small square. Nonetheless, we suggest that you play with texture band limits yourself (see Exercise 20.3) to get a feel for what's required.

In those situations where you need to implement texture mapping yourself (as in a software ray tracer, where you don't have the hardware support provided by a graphics card), and you want to do image-based texture mapping, two problems arise.

1. Sometimes you view an object from very close up, and even the texture map doesn't have enough detail: Two adjacent pixels in the texture map end up corresponding to two pixels in the final image that are perhaps ten pixels apart. You need to interpolate texture values in between. The usual solution is bilinear (for surface textures) or trilinear (for solid textures) interpolation.
2. Sometimes you view an object from a distance, and many texture pixels map to one image pixel. In this case, as we said, MIP mapping is the most widely used solution.

Note that the texture-sampling grid will rarely be aligned well with the screen-sampling grid, so merely having the texture resolution match the screen resolution (i.e., adjacent texture pixels project to points that are about one screen pixel apart) will still result in blurring. In practice, you need at least twice the resolution in each dimension for bilinearly interpolated texture values to look "sharp enough" when the texture's not exactly aligned with the final image pixel grid (or sampling pattern, if you're using something more complex than single-sample-at-the-pixel-center ray tracing).

20.8 Texture Synthesis

We've suggested gathering textures from photographs (for the soda can) or from data (mapping a world map onto a sphere) or from direct design (as in the 1D texture used for contour rendering). If you want to create a texture that's unlike anything seen before, you may want to use **texture synthesis,** a process for generating textures either *ab initio* or in some clever way from existing data. You might have a photo of part of a brick wall, and wish to make an entire brick building without the kind of obvious replication that wrapping from bottom to top and right to left might produce, or you might want to make a hilly area via a displacement map in which the displacements tend to produce rolling hills at a certain scale, but without repetition. We'll now discuss a few approaches to these problems.

20.8.1 Fourier-like Synthesis

For the rolling-hills problem, one solution is a procedural texture, where you assign a displacement in the form

$$d(x,y) = \sum_{i=0}^{n-1} c_i \cos(a_i x + b_i y + c), \tag{20.11}$$

where the numbers a_i, b_i, and c_i affect the orientation of the cosine waves, with c_i displacing the ith wave along its direction of propagation and with $\sqrt{a_i^2 + b_i^2}$

determining its frequency. If the points (a_i, b_i) are chosen at random from an annular region in the plane given by $r^2 \le a_i^2 + b_i^2 \le R^2$, then the resultant waves will all have roughly similar periods, producing a result like that shown in Figure 20.14. Geoff Gardner [Gar85] called this a "poor man's Fourier series," because of the use of only n terms (many of his applications used something like $n = 6$ terms).

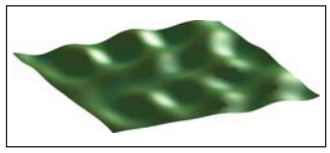


Figure 20.14: Displacement map synthesized with six terms.

By the way, Gardner also showed that this function could be used for other things; he used a three-dimensional version to represent the density of a cloud at a point (x, y, z), and the 2D version with thresholding to determine placement of vegetation in a scene: Anyplace where d was above some specified value, he placed a tree or a bush, generating remarkably plausible distributions.

20.8.2 Perlin Noise

Perlin [Per85, Per02b] took a different approach, aiming to directly produce "noise" whose Fourier transform was nonzero only within a modest band, or at least whose values outside that band were small, and where the noise values themselves were constrained to $[-1, 1]$. The main idea can be illustrated where the noise is a function of a single real parameter x. At each integer point we (a) want the value at that point to be 0, and (b) want the function to have some gradient, which for simplicity we choose to be 0, 1, or -1. If, at $x = 4$, we choose a gradient of $+1$, then we can build the function

$$y_4(x) = +1(x - 4), \tag{20.12}$$

which has the value 0 at $x = 4$, and the derivative $+1$ there. After picking gradients at each integer point, we get functions $y_1, y_2, \ldots$. The idea is that on the interval $4 \le x \le 5$, we can *blend* between $y_4(x)$ and $y_5(x)$ with a function that varies from 0 to 1 as x goes from 4 to 5. The fractional part of x, that is, $x - 4$, is such a function, which results in

$$y = y_4(x) \cdot (x - 4) + y_5(x) \cdot (1 - (x - 4)). \tag{20.13}$$

Unfortunately, the resultant graph has sharp corners at integer values. To resolve this, we need a nicer interpolation. We will use the fractional part of x, $x_f = x - \text{floor}(x)$, as the argument to our blending function, namely,

$$a(t) = 6t^5 - 15t^4 + 10t^3, \tag{20.14}$$

which varies from $a(0) = 0$ to $a(1) = 1$, and has $a'(s) = 0$ for $s = 0, 1$. The result is a second-order smooth interpolation between the successive linear functions. The result is shown in Figure 20.15.

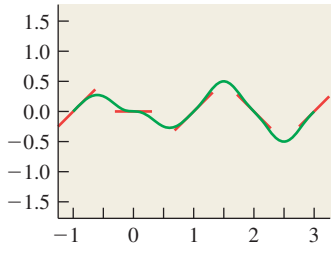


Figure 20.15: One-dimensional Perlin noise; the individual linear functions are shown as short red segments.

In practice, to create a larger version of Figure 20.15, we would generate, say, 20 random items from the set $\{-1, 0, 1\}$ and use these as the gradient values at locations $x = 0, 1, \ldots, 19$. For values outside this range, we reduce mod 20. That produces a repeating pattern, but it is much faster than invoking a random number generator lots of times. Of course, one can choose any fixed number of gradients—20 or 200 or 2,000—to avoid repetition in practice.

Note that because the control points had unit spacing, the resultant spline curve had features that were spaced about a unit apart in general, and hence they had a Fourier transform that was concentrated at frequency one. This idea can be generalized to 2D and 3D. To generalize to three dimensions (2D is left as an exercise for you), we must select (3D) gradients at each integer point. Perlin recommends

picking them from the set 12 vectors from the center of a cube to its edge midpoints, namely,

$$(1,1,0) \quad (-1,\ 1,0) \quad (1,-1,\ 0) \quad (-1,-1,\ 0) \tag{20.15}$$
$$(1,0,1) \quad (-1,\ 0,1) \quad (1,\ 0,-1) \quad (-1,\ 0,-1) \tag{20.16}$$
$$(0,1,1) \quad (\ 0,-1,1) \quad (0,\ 1,-1) \quad (\ 0,-1,-1), \tag{20.17}$$

to avoid a kind of bias in which the gradient of the noise function ends up preferentially aligned with the coordinate axes. By appending a redundant row to this list, namely,

$$(1,1,0) \quad (-1,\ 1,0) \quad (0,-1,\ 1) \quad (\ 0,-1,-1), \tag{20.18}$$

we end up with 16 vectors, which makes it easy to use bitwise arithmetic to select one of them.

Having assigned a gradient vector $\begin{bmatrix} u & v & w \end{bmatrix}^{\mathbf{T}}$ to the lattice point (i,j,k) we define a linear function

$$g_{i,j,k}(x,y,z) = u(x-i) + v(y-j) + w(z-k). \tag{20.19}$$

To find the noise value at a point (x,y,z), we let i_0, j_0, and k_0 be the floor of x, y, and z, respectively, so that $i_0 \le x < i_0 + 1$, and similarly for y and z, and the eight corners of the grid cube containing (x,y,z) have coordinates (i_0,j_0,k_0), (i_0,j_0,k_0+1), (i_0,j_0+1,k_0), $\ldots$, (i_0+1,j_0+1,k_0+1). For each corner (i,j,k), we evaluate the linear function associated to that corner at the point (x,y,z) to get a value $v_{i,j,k}$. We blend these values trilinearly using $a(x_\mathrm{f})$, $a(y_\mathrm{f})$, and $a(z_\mathrm{f})$. A direct but inefficient implementation is given in Listing 20.1 (the function a of Equation 20.14 must also be defined). Perlin [Per02a] provides an optimized implementation.

The result (in 2D) is shown in Figures 20.16 and 20.17.

We can use 3D Perlin noise to create a radial displacement map on a sphere, while also coloring points by their displacement, to get a result like that shown in Figure 20.18.

Perlin [Per85] describes far more complex applications, allowing him to produce things like the marble texture in Figure 20.19.

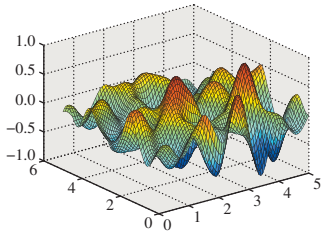


Figure 20.16: 2D Perlin noise on a 6×6 region of the plane.

20.8.3 Reaction-Diffusion Textures

A third approach to texture synthesis is due to an idea by Turing [Tur52] about the formation of patterns like leopard spots or snake scales in nature. He conjectured that such patterns could arise from evolving concentrations of chemicals called **morphogens.** Typically, one starts with two morphogens, with concentrations randomly distributed across a surface. They evolve through a combination of two processes: **diffusion,** in which high concentrations of morphogens diffuse to fill in areas of low concentration, and **reaction,** in which the two morphogens combine chemically to either produce or consume one or both of the morphogens. If the presence of A promotes the production of B, but the presence of B promotes the consumption of A, for instance, interesting patterns can arise. The appearance

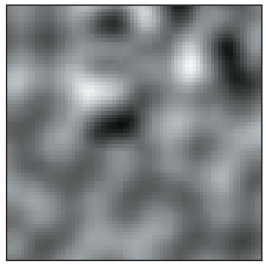


Figure 20.17: The same function, displayed as a grayscale image where -1 is black and $+1$ is white.

Listing 20.1: An implementation of Perlin noise using a $256 \times 256 \times 256$ cube as a tile.

```
grad[16][3] = (1,1,0), (−1,1,0), ...

noise(x, y, z)
  reduce x, y, z mod 256
  i0 = floor(x); j0 = floor(y);  k0 = floor(z);
  xf = x − i0;   yf = y − j0;    zf = z − k0;
  ax = a(xf) ;   ay = a(yf) ;    az = a(zf); // blending coeffs

  for i = 0, 1; for j = 0, 1; for k = 0, 1
    h[i][j][k] = hash(i0+i, j0 + j, k0 + k)
    // a hash value between 0 and 15
    g[i][j][k] = grad[h[i][j][k]]
    v[i][j][k] = (x − (i0+i)) * g[i][j][k][0] +
                 (y − (j0+j)) * g[i][j][k][1] +
                 (z − (k0+k)) * g[i][j][k][2]

  return blend3(v, ax, ay, az)

blend3(vals, ax, ay, az)
  // linearly interpolate first in x, then y, then z.
  x00 = interp(vals[0][0][0], vals[1][0][0], ax)
  x01 = interp(vals[0][0][1], vals[1][0][1], ax)
  x10 = interp(vals[0][1][0], vals[1][1][0], ax)
  x11 = interp(vals[0][1][1], vals[1][1][1], ax)
  xy0 = interp(x00, x10, ay)
  xy1 = interp(x01, x11, ay)
  xyz = interp(xy0, xy1, az)
  return xyz

interp(v0, v1, a)
  return (1−a) * v0 + a * v1
```

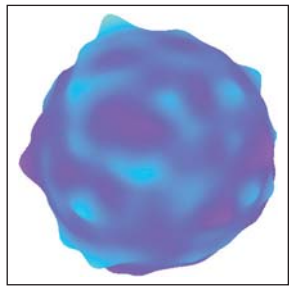


Figure 20.18: A sphere radially displacement-mapped with Perlin noise.

of these patterns is governed by many things: the particular differential equation representing the change in morphogen A as a function of the concentrations of A and B, the rate (and direction) of diffusion of the morphogens, and the initial distribution of concentrations. Turing's idea was that the steady-state concentration of one of the morphogens might control appearance so that, for example, a zebra's skin would grow white hair where the concentration of morphogen A was small, but black hair where it was large.

Turing lacked the computational power to perform simulations, but Turk [Tur91] and Kass and Witkin [WK91] took his ideas, extended them, and made it practical to run simulations to predict the steady-state concentrations resulting from some set of initial conditions, not only on the plane, but also on more general surfaces. Figure 20.20 shows some examples of the **reaction-diffusion textures** that they generated.

Figure 20.19: A marble vase with texture based on a complex combination of noise functions (Courtesy of Ken Perlin, © 1985 ACM, Inc. Reprinted by permission).

20.9 Data-Driven Texture Synthesis

As a final topic, we'll briefly discuss two methods for generating new texture from old. The first of these is Ashikhmin's texture synthesis algorithm [Ash01], which is based on ideas in papers by Efros and Leung [EL99] and Wei and Levoy [WL00], which in turn follow work of Popat and Picard [PP93]. The input

for the algorithm is a texture image (e.g., a photograph of a brick wall) that's called the **source,** and a size, $n \times k$, of an image to be created. The output is an $n \times k$ image we'll call the **target;** typically $n \times k$ is much larger than the size of the input image. The idea is that the target should look like it comes from "the same stuff as the source"; for example, like a photograph of a larger region of a brick wall. Figure 20.21 shows the idea. We fill in the target image row by row, moving left to right. At a typical point in the filling-in procedure, the blue-colored pixels have had their values determined and the yellow area is still to be processed. We select a small rectangle (2×3 in the figure) containing mostly known pixels, and one unknown (pink) pixel at the bottom right. (We'll call this set of six pixels a **template.**) The goal is to determine a value for this last pixel, and then advance the template one step to the right.

We take the five known pixels in the rectangle, and look in the source image for similarly shaped clusters that match these five known pixels. We then pick one with a good fit, and copy its *sixth* pixel into the appropriate spot in the target. We then move the template one step to the right, and proceed.

This description glosses over several important points, such as "How do we get started?" and "How do we move to a new row?" and "How do we find in the source image good-matching patches for the known template pixels without taking forever to do it?" One way to start is with random pixels from the source image filling the target. As we start at the upper left, finding a "match" for the first 5-pixel patch will be very difficult—we'll have to be happy with a not-very-good result. But as we move forward, things gradually start to work better. To improve matters, we may want to run the algorithm several times, working top to bottom, bottom to top, left to right, right to left, etc., to clean up the edges.

As for finding good candidates as matching patches, one useful observation is that if you move the template one step to the right, a good candidate for filling in the next pixel is exactly one step to the right of the source pixel you just used (i.e., the algorithm's very likely to favor copying whole rows of pixels). This can be generalized. For instance, right above the missing pixel is one that you filled in one row earlier. If we look at the corresponding source pixel, the one immediately below it is a good candidate for the missing pixel, too. In fact, the source locations of all five known template pixels similarly provide (with slight offsets) good candidate locations in which to find a source for the missing sixth pixel. The algorithm proceeds by picking a pixel from this candidate set. Occasionally (e.g., when a source pixel is near the edge of the image) this may not work, in which case we have to replace this candidate with another; there are many possible choices, and the details are not important.

The results are quite impressive. Figure 20.22 shows an example in which a few berries are used to synthesize many berries. The algorithm has another advantage: It's possible to start with the target image partly filled in! We can declare in advance that we want certain pixels to have certain colors, and when the algorithm reaches these pixels, it finds them already "filled in" and leaves them unchanged. But their presence affects how well the subsequent patches fit with one another, and hence the synthesis of subsequent pixels. Figure 20.23 shows an example.

The end result of the synthesis process is that long diagonal strips from the source tend to be copied whole, in such a way that they match up with neighboring strips. Figure 20.24 shows this structure.

In this situation, a small square of the target image around some pixel (i,j) tends to match a small square in the source image around some pixel (i',j'), although along the edges between strips this isn't an exact match. If we compute

Figure 20.20: Reaction-diffusion textures. Top: Textures synthesized by Kass and Witkin. (Courtesy of Michael Kass, Pixar and Andrew Witkin, © 1991 ACM, Inc. Reprinted by permission). Bottom: Texture synthesized by Greg Turk. (Courtesy of Greg Turk, © 1991 ACM, Inc. Reprinted by permission.)

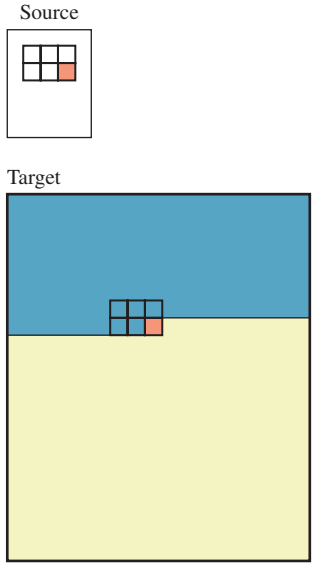


Figure 20.21: A partly synthesized image, and a highlighted region with one unknown pixel.

$\mathbf{v}_{ij} = \begin{bmatrix} i' - i & j' - j \end{bmatrix}^{\mathbf{T}}$, then we see that $\mathbf{v}$ tends to be locally constant as a function of i and j, precisely because of the approach that the algorithm takes, namely, using nearby pixels' $\mathbf{v}$-vectors as candidates for the $\mathbf{v}$-vector of the pixel being synthesized.

This notion can be generalized: Given an image A and an image B, we can extract, for each pixel (i,j) of A, a $p \times p$ square P centered there, and look at every $p \times p$ patch in B to find one "closest" to P. We then find the center of this B-patch, (i',j'), and set $\mathbf{v}_{ij} = \begin{bmatrix} i' - i & j' - j \end{bmatrix}^{\mathbf{T}}$. The resultant collection of vectors is called a **nearest-neighbor field** (for patches of size p). Computing this field can be very slow, but if the images A and B are similar, the field tends to have the same kind of coherence exhibited by the results of the Ashikhmin algorithm: Neighboring pixels tend to have similar vectors. Barnes et al. [BSFG09] use this observation to develop an approach to computing the nearest-neighbor field (or a very close approximation) for two images very quickly.

While the resultant field might be used for texture synthesis if B is an example texture and A is a large image that has some structural resemblance to the texture in B, Barnes et al. actually describe a great many *other* computational photography applications based on it, such as the "image shuffling" shown in Figure 20.25, in which the user annotates a region (in this case, the person in the photo) and a place to which to move the region. A combination of the nearest-neighbor field computation and an expectation-maximization algorithm that seeks to fill in the hole (and adjust pixels near the moved region) in a way that makes the patch matching optimal produces the image in which the person appears to have moved within the photo.

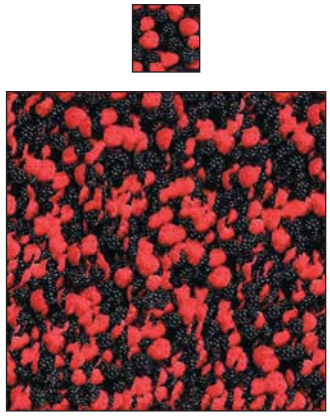


Figure 20.22: A tiny source image and a large image synthesized from it (Courtesy of Michael Ashikhmin, © 2001 ACM, Inc. Reprinted by permission.)

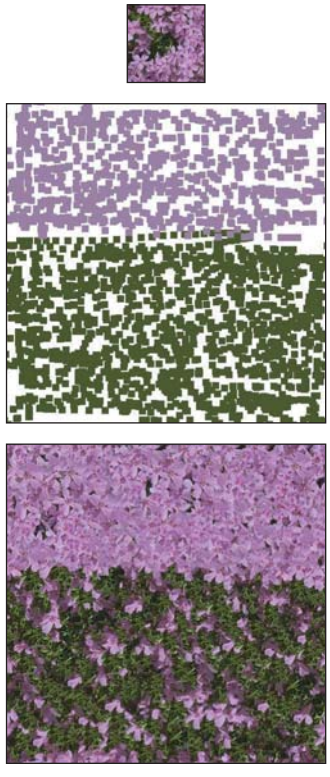


Figure 20.23: A source image, a hand-drawn "guide," and the resulting synthesized image after five rounds of synthesis (Courtesy of Michael Ashikhmin, © 2001 ACM, Inc. Reprinted by permission.)

20.10 Discussion and Further Reading

We've only touched on texture mapping here. Table 20.1 gives some hint of how powerful the method is, which should hardly be surprising, since at its essence, it's "indirection" or "function evaluation," both of which are at the very heart of computation.

With texture mapping, we're (generally) altering parameters to a lighting computation. But in Chapter 31, we'll see that what we actually want to compute for each sample in a rendering is an estimate of an integral, and the integrand has the general form of a lighting model. As we move from sample to sample, this integral varies. If the variation is not band-limited, we'll get aliasing errors. But texture mapping makes certain that the individual parameters within the integrand are each band-limited, rather than the value of the integral. In other words, we've band-limited and then integrated, rather than integrated and then band-limited. Since these two operations do not generally commute, we're producing potentially incorrect results. The degree to which they're incorrect has not yet been thoroughly analyzed.

Although texture mapping is used in rendering, it's also forced into service in more general computations. Both GL and DX shaders use texture maps as general-purpose RAM for storing arbitrary data structures. That is because the first graphics systems didn't support random access memory or anything beyond primitive floating-point types. The practice of tricky uses of texture memory is losing currency with new hardware that supports arbitrary read and write operations, often using sophisticated abstractions. DirectCompute and CUDA shaders already

just look like C programs with regard to memory access . . . but they still have the useful API for texture fetches that implement mappings and filtering.

Texture synthesis is also a rich and active area of research, from *ab initio* methods like Cook and DeRose's work on wavelet noise [CD05b], which generalizes Perlin noise, to work on detail hallucination, in which highly zoomed textures that would otherwise be blurry are enhanced with further detail that matches the neighboring texel values [SZT10, WWZ+07].

Figure 20.24: The striplike structure of a synthesized image. Coherent pixel regions are outlined in white. (Courtesy of Michael Ashikhmin, © 2001 ACM, Inc. Reprinted by permission.)

20.11 Exercises

Exercise 20.1: In varying parameters to the Phong model, we've used texture mapping to adjust the incoming light, the outgoing light, and the diffuse and glossy constants. But we haven't used it to modify the dot product. If we replace $\mathbf{v}\cdot\mathbf{w} = \mathbf{v}^{\mathbf{T}}\mathbf{w}$ with $\mathbf{v}^{\mathbf{T}}\mathbf{M}\mathbf{w}$, where $\mathbf{M}$ is some symmetric matrix with positive eigenvalues, then the results computed by the Phong model will change depending on the orientation of the eigenvectors and the magnitudes of the eigenvalues. Apply this idea to the unit sphere. Writing $\mathbf{M} = \mathbf{V}\mathbf{D}\mathbf{V}^{\mathbf{T}}$ where $\mathbf{V}$ contains the eigenvectors as columns, and $\mathbf{D}$ is a diagonal matrix with the eigenvalues on the diagonal, experiment with what happens when $\mathbf{V} = \begin{bmatrix}\mathbf{t}_1 & \mathbf{t}_2 & \mathbf{n}\end{bmatrix}$, where $\mathbf{t}_1$ and $\mathbf{t}_2$ are unit vectors tangent to the latitude and longitude lines, $\mathbf{n}$ is the surface normal, and $\mathbf{D} = \mathrm{Diag}(s,t,1)$, where $0 < s,t \leq 1$. You should be able to achieve the general appearance of brushed metal if you use this altered inner product in the glossy part of the Phong model.

Exercise 20.2: On the unit disk D in the plane, consider the vector field $\mathbf{n}(x,z) = \mathcal{S}(\begin{bmatrix}-z & 1 & x\end{bmatrix}^{\mathbf{T}})$. Show that there's no function $y = f(x,z)$ on D with the property that the normal vector to the graph of f at $(x,f(x,z),z)$ is exactly $\mathbf{n}(x,z)$, that is, that $\mathbf{n}$ is not the normal field of any surface above D. Hint: Assume without loss of generality that $f(1,0) = 0$. Now traverse the curve $\gamma(t) = (\cos t, 0, \sin t), 0 \leq t \leq 2\pi$ and see what you can say about the restriction of f to this curve.

Exercise 20.3: (a) Write a program to render a picture like the one shown in Figure 20.13. For the checkerboard itself, assuming 0 is black and 1 is white, make the light squares about 0.85 and the dark squares about 0.15, and be sure you can render a picture like the one shown.
(b) Figure out the size of the screen-space projection of a unit square at location $(x,0,z)$ on the ground plane, either algebraically or by projecting the four corners and computing numerically. From this, determine the vertical and horizontal band limits as a function of x and z.
(c) The texture color at location (x,z) on the plane can be written as $0.5 + 0.35S(x)S(z)$, where $S(x) = 1$ if $\mathrm{floor}(x)$ is even, and -1 otherwise. With methods like those of Chapter 18, you can compute the Fourier series for S; it's

$$S(x) = \frac{4}{\pi}\sum_{j=0}^{\infty}\frac{1}{2j+1}\sin(\pi(2j+1)x). \tag{20.20}$$

To band-limit this, you need only truncate the sum, and define

$$\bar{S}(x,\omega_0) = \frac{4}{\pi}\sum_{j=0}^{\mathrm{floor}(\frac{\omega_0-1}{2})}\frac{1}{2j+1}\sin(\pi(2j+1)x), \tag{20.21}$$

Figure 20.25: The man in the original image (a) is translated to the left in image (b). (Courtesy of Connelly Barnes)

which removes all frequencies ω_0 and above. Rerender your image using $\bar{S}$ with an appropriate band limit, based on z, and compare aliasing in the rerendered image.
(d) The aliasing may still seem excessive to you, and the Gibbs phenomenon artifacts may also be annoying. Experiment with other approximations to a band-limited version of S, and with band limiting to a lower frequency than might seem necessary, and evaluate the results.
◈ (e) Truncating the series at the band limit (i.e., multiplying by a box in the frequency domain) is not the only way to remove high frequencies. Try using the Fejer kernel (i.e., multiplying by an appropriately dimensioned tent in the frequency domain) to see if you can get more satisfying results.

Exercise 20.4: In two dimensions, imagine a parabolic mirror like the one shown in red in Figure 20.26, with the equation of the form $z = \frac{1}{2}(1 - y^2)$. Rays (in green) coming from a semicircle of directions are reflected to become rays (in blue) parallel to the z-axis (and vice versa).
(a) Show that the yz-vector $\mathbf{n} = \begin{bmatrix} y, 1 \end{bmatrix}^{\mathbf{T}}$ is normal to the red curve at the point $(y, z) = (y, \frac{1}{2}(1 - y^2))$.
(b) If light traveling in the direction $\begin{bmatrix} 0 & -1 \end{bmatrix}^{\mathbf{T}}$ strikes the mirror at (y, z) and reflects, in what direction $\mathbf{r}$ does it leave? Write out your answer in terms of y and z.
(c) Show that at $(\pm 1, 0)$ the outgoing ray is in direction $\begin{bmatrix} \pm 1 & 0 \end{bmatrix}^{\mathbf{T}}$.
(d) Show that incoming rays in direction $\begin{bmatrix} 0 & -1 \end{bmatrix}^{\mathbf{T}}$ whose y-coordinate is between -1 and 1 become outgoing rays in all possible directions in the right half-plane.
(e) If you spin Figure 20.26 about the z-axis, the red curve generates a paraboloid. Show that in this situation, incoming rays in the direction $\begin{bmatrix} 0 & -1 \end{bmatrix}^{\mathbf{T}}$ with starting points of the form $(x, y, 1)$, where $x^2 + y^2 \leq 1$, produce all possible outgoing rays in the hemisphere of directions with $z \geq 0$.
(f) If we take a second paraboloid defined by spinning $z = \frac{1}{2}(-1 + y^2)$, then arrows in direction $\begin{bmatrix} 0 & 1 \end{bmatrix}^{\mathbf{T}}$, arriving from points of the form $(x, y, -1)$, where $x^2 + y^2 \leq 1$, reflect into outgoing rays in the opposite hemisphere of directions. These two reflections establish a correspondence between (1) two unit disks parallel to the xy-plane, and (2) two hemispheres of directions. Write out the inverse of this correspondence.
(g) Explain how you can represent a texture map on the sphere (e.g., an irradiance map, or an environment map, etc.) by providing textures on two disks.
◈ (h) Estimate the largest value of the change of area for this "dual paraboloid" parameterization of the sphere to show that it uses texture memory quite effectively, even though $\pi/4$ of each texture image (the disk within the unit square) gets used for each half of the parameterization.

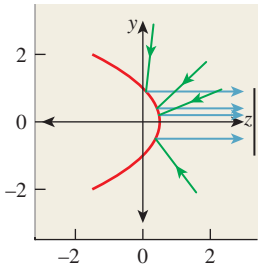


Figure 20.26: The red parabolic mirror reflects light from a semicircle of directions onto a line segment.

Chapter 21

Interaction Techniques

21.1 Introduction

While human-computer interaction is a field in itself, certain interaction techniques use a substantial amount of the mathematics of transformations, and therefore are more suitable for a book like ours than one that concentrates on the design of the interaction and the human factors associated with that design. We illustrate these ideas with a multitouch interface for 2D manipulation, and three 3D manipulators: the arcball, trackball, and Unicam. In each case we discuss the mathematics, but also the design choices made in creating the interaction technique.

We begin with a discussion of some basic ideas in interaction that everyone in graphics should know. Section 21.3 discusses an implementation of a simple multitouch photo-sorting application. We then discuss 3D transformation interfaces, both for rotating objects and for adjusting the camera in a scene. We conclude with some guidelines and example interfaces that demonstrate particularly useful ideas.

21.2 User Interfaces and Computer Graphics

Despite the advances in games and other technologies, the single biggest user of computer graphics is still and will continue to be the graphical user interface (GUI), by almost any measure except the number of pixels drawn, where games (or video display) undoubtedly dominate. This GUI is increasingly a combination of the **WIMP** (windows, icons, menus, pointers) **GUI** and post-WIMP developments like multitouch interfaces and 3D gestural interfaces.

There are two reasons for this. The first is the commoditization of hardware due to Moore's law and the superb engineering of interaction devices, displays, and wireless technologies. The second reason is the economics of computing: While it was once true that computers were expensive and users were not, the opposite is now true—processors are so cheap that an entry-level computer costs less than a week's salary at minimum wage. In this economic environment, it

makes sense to save time where it's expensive—the user!—rather than where it's cheap.

The discovery of an effective interface model—the WIMP GUI—to replace the cryptic mechanisms of the past not only enlarged the market for computing, but in doing so, enabled further progress by providing processor makers with large economies of scale: The cost of developing a new machine could be amortized over many more users.

> The modern GUI had its origins in Sutherland's Sketchpad system [Sut63], a CAD system that used a light pen and many (physical) buttons for its input, and an oscilloscope for output. It included direct-manipulation tools, selection by pointing, grouping, constraint-based interaction, and many other ideas that are being constantly reinvented even now. In the 1970s, at Xerox PARC, researchers developed the WIMP interface in a form that closely resembles the modern version (albeit in black and white), for a machine called the Alto. Much of this design was adopted in the design of the Apple Lisa (and later the Macintosh). In the decades since its introduction, it's come to dominate interaction, and has only recently begun to be challenged by new multitouch interaction and new interface devices like the Wii and Kinect.

While the framework provided by a GUI design like WIMP is a wonderful stepping stone, developing a good user interface is still extremely difficult. Although trial and error have their place in the exploration of possible designs, effective designs need testing and refinement, and having a model of the entire process of interaction, from the machine-dependent side (the pixel position of a pointer, filtering of pointer tracks to remove noise, etc.) to the human (the user's mental state, or his or her goals and sense of progress toward those goals, as in "I'm trying to move this paragraph, and I've succeeded in selecting it ...") is critical to both of these. The study of effective interaction is the field of **human-computer interaction** (**HCI**) [PRS02]. HCI is intensely multidisciplinary, involving hardware and software engineering, computer and mathematical sciences, design arts, ergonomics, and perhaps most important, human sciences (perception, cognition, and increasingly, social interaction), not to mention cultural and accessibility issues. It is, first and foremost, a design discipline, one where results are subject to experimentation and validation.

Such usability testing is surprisingly complex. Consider the problem of comparing two interface choices: one easy to learn but with limited expressive power, the other with great expressive power but difficult to learn. A good example is the choice of function keys versus a mouse for selecting menu items. Function keys are easy to learn, while using a mouse effectively requires several days of training. (If you doubt this, try using your mouse with your other hand for an hour. Even knowing all about the mouse, you'll soon find it's annoying you more than helping you.) Which is better? The function keys or the mouse? The answer is, naturally, that it depends: If you're going to be using the mouse for lots of other things as well, the eventual benefit may be large enough to make it worth learning (and the immediate benefit may be large enough to motivate you to do so). If doing this particular task is a one-time-only event, then the simpler interface is almost certainly better. As a concrete example, Adobe's Photoshop has an enormous user interface that takes quite a long time to learn completely. As a novice user, it sometimes seems that everything you do makes the picture worse! But when used

with a pen and tablet (for which much of the design is optimized), the interface supports such a wide array of operations relatively smoothly that it's become the dominant tool in its domain. By contrast, simpler image-editing programs like the Microsoft Office Picture Manager are easy to learn and use instantly, but this is in part because they support such a small range of operations. To be clear: A complex interface may be a necessity for expressive power, but not every complex interface is a good one. A common evolution pattern is **accretion,** in which new features are added to a program over time, each one added in the place that seems most convenient at that moment. The end result is a complex interface in which there's little logical organization at all, and the resultant program may be difficult to use, even for experts who use it every day.

These examples, though simple, make it clear that the testing of a user interface may depend on a larger context—not just the interaction process or device, but the entire user experience, which bundles the GUI together with its interaction and with the particular software functionality, and perhaps even with the context of use (shopping mall versus automobile versus office).

Before we leave the topic of complexity versus learnability, there are two more relevant aspects of GUIs. First, there's a general principle that recognition is faster than recall: It's easier to recognize a "yield" sign in the United States than to say whether its triangular shape points up or down, for instance. In the case of GUIs, this means that using familiar names and icons can help a new user make sense of a new interface almost instantly. For the same reason, placing menu items in expected places is generally a good idea. Second, you should, if possible, design a **gentle slope interface** [HKS+97], one in which it's easy to do something right away, but in which there's a smooth transition from novice to power user. Menus that display, next to each item, a keystroke that invokes that menu item are an example. Things like **tool trays,** which are buttons that can either be clicked (to invoke a standard operation) or be expanded into multiple buttons (to allow selection of closely related operations), provide easy access to richer functionality. (For example, a drawing program might have a button that selects line-drawing mode. When its tool tray is expanded, there might be options to draw solid, dotted, or dashed lines.) Such gentle slope interfaces provide a pathway between ease of initial use and ease of expert use.

As with software engineering, there are multiple design approaches that all share a common trait of needing to be user-centered, that is, to know the client and the domain. Two dominant ones are (a) a modified waterfall model[1] for software engineering, and (b) rapid prototyping, in which the evolving interface is always functional, but is gradually adapted from minimal function (clicking a button generates a "button clicked" message) to sophisticated interaction sequences. Some mixture of both of these processes is typical in the development of new kinds of interaction.

Abstraction boundaries can help you develop an interface effectively. These boundaries are the places where substitutions may make sense, whereas within a particular layer, there may be dependencies that make substitution less feasible. For instance, we may have a design in which a mouse is used to point at various

1. In the waterfall model, requirements determine design; the design determines the implementation. After implementation, the system is verified, and then maintained. Each step is completed before the next. In the modified waterfall, there is substantial feedback at all levels.

things; substituting a pen for the mouse's pointing functionality is often reasonable (although if clicking or double-clicking is part of the process, then the substitution may have to be more complex, with pen taps replacing button clicks). Replacing the pen with a Wiimote, or with your hand in a Kinect system, is similarly reasonable, although with each substitution, the details of the interaction must necessarily change. What *doesn't* change is the intent to identify or select certain objects in the scene through some interaction, which makes the separation of intent from implementation a natural boundary.

In interaction, there is communication between human and computer, typically in two languages: The user-to-computer direction involves various interaction devices, and the computer-to-user direction is primarily through the display to the eye, although there may also be audio or touch components. The meaning and form of each of these languages constitute natural abstraction boundaries: We must decide what things a user may communicate to the computer (meaning) and how each thing is communicated (form), and vice versa. There is also a third component: the relationship of interaction device to display, or the mathematics or algorithm required to transform the input into something meaningful in the output. But this is typically application-dependent and represents the *computation* rather than the communication between human and machine.

The two languages in turn break down into finer levels.

- **Conceptual design** is the model of the user's understanding of the application (e.g., a 3D modeling application), typically consisting of objects (shape, texture, control point), relationships among objects (textures are applied to shapes, splines are governed by control points), and operations on them (we can apply a texture to an object, or reshape a spline curve).
- **Functional design** is the specification of the interface to the operations of the conceptual design. It includes a specification of what information is needed for an operation, what errors may occur (and how they are to be handled), and what the results are. The functional design is an abstraction of the operations, but *not* of the user interface. We would specify that to apply a texture to a shape, we need the texture and the shape and the texture coordinates on the shape, but would leave the question of how the user communicates the texture or shape to a later stage. Conceptual and functional design together constitute the "meaning" part of the interaction language.
- **Sequencing design** describes the ordering of inputs and outputs, and the rules by which inputs may be assembled to generate meaning. A click and drag on a model may be meaningful (indicating screen-aligned translation of the model), while a click and drag on the empty part of a menu bar may be ignored as meaningless.
- **Lexical design** determines what constitutes the units of a sequence. For input, these are things like a single click, a double-click, a drag operation, etc. For output, they may be things like blinking, displaying a dialog box, the choice of font or text color for text display, etc.

Not all interaction is purely sequential; in two-handed multitouch interfaces, both hands may be doing things that, taken together over some period of time, have some meaning, but the precise ordering is irrelevant; nonetheless, a generalized notion of sequence design provides a good boundary even in these cases.

Interactions like the two-handed multitouch example above are the simplest cases of what are being called **natural user interfaces** (**NUIs**). These are interfaces that can involve multiple nondeterministically decoded channels of communication, leveraging our different senses (e.g., the ability to point with a finger while giving instructions by voice). Not surprisingly, the decoding of multiple streams of data into a coherent goal can be very challenging. One particular challenge is that in the WIMP interface, each interaction is purposeful and demarcated: We start an action by pressing a start button, for instance, and the meaning is completely clear. But for a camera-based interface that watches a user's face or hands for indication of an action to take, there's no clear delimiting of the action; the system must infer the start and end.

21.2.1 Prescriptions

We conclude these generalities with a few ideas that are important for anyone designing any kind of interface. There are no absolute prescriptions in interaction design except, perhaps, "You should test your design on real users." Designs must often satisfy the needs of both beginners and power users, and until the design is widely adopted, it's not certain that it will ever *have* power users. Designs must work within a budget: Interaction may be allocated only a tiny fraction of processor time, pixel fill rate, or other resources. As processor speed, fill rate, bandwidth, and other factors change, the sweet spot for a design can shift substantially.

For every design, some degree of responsiveness and fluidity is essential. When you click a button on a GUI, you need to know that the click was detected by the program: The button should change its appearance, and perhaps you should get audio feedback as well. It's essential that these happen apparently instantly—by the time there's a lag of even 0.2 sec, the interface begins to feel clunky and unreliable. The more "immediate" the GUI feels, the more critical prompt feedback becomes: When we feel separated from the computer, treating it as a device or machine, some delay is tolerable. The more we perceive it as "real," the more we expect things to behave as they do in the real world, that is, with instant feedback. With modern controllers—you use your hand to select from a menu in many Kinect-based games, for instance—the feeling of reality is substantially enhanced, and real-time feedback is essential. In fact, the separation of an interaction loop (something that receives and processes interrupts from interaction devices, with a high processor priority) into its own high-priority thread of execution is critical to maintaining a sense of hand-to-eye coordination, and a feeling of fluidity in the interface.

The need for instant feedback and fluidity is context-dependent: A WIMP desktop GUI may need smooth feedback, but a twitch game demands it—players get annoyed when their on-time interactions register too late to be effective! In a virtual reality environment, it becomes critical: Failure to update the interface (which may be the entire scene!) can lead to **cybersickness** (nausea due to inconsistent apparent motion). Thus, sufficiently rapid feedback becomes almost as severe a constraint as hard-real-time scheduling.

There are automobiles that seem "right" the moment you sit in them. You can tell instantly where all the controls are. As you grab the steering wheel, you notice that there are buttons nearly under your thumbs, in easy reach, but placed so that you won't trigger them accidentally. When you shift the transmission, the current gear is displayed clearly but subtly. When a display element changes discretely,

like the transmission indicator, it's because there was a change of state; continua like speed and coolant temperature are displayed with analog gauges. In the same way, there are interfaces that seem "right." There are some basic ideas that can help your interfaces be among these good ones.

First, use **affordances,** the way that objects disclose the possible actions that can be taken. We know to pick up a hammer by its handle because the handle is designed to fit the human hand. We know that something is a button in an interface because it looks like other buttons we've seen, either in the real world or in other interfaces. When we see visual elements, such as the draggable corner or side markers on a bounding rectangle in a drawing program, that seem to contrast with others (the bounding rectangle itself), we conjecture that they might have meaning. Such affordances make interfaces easy to learn through discovery. Objects that expose their manipulability in response to attention (or some proxy for attention) help as well: The spreadsheet column whose sides highlight as the cursor passes over them (with the cursor changing to a column-resizing icon) help us understand that columns are resizable; the position of the cursor is a proxy for the user's attention.

Note that many of the aspects of expert use of interfaces ignore affordances. There's nothing that tells you, as you select some text, that pressing CTRL-C will copy that text so that it can later be pasted. But it may well be worth it to you to know this so that you need not use the ever-apparent menu to perform the very frequent "copy" operation. Gestural interfaces, too, often lack affordances, except for those familiar from interaction in the real world (e.g., "If I drag something, it moves").

Second, use **Fitts' Law** to help your designs. Fitts' Law, proposed by Paul Fitts in 1954 [Fit54], describes how long it takes to move from rest to a point within a target at some distance (see Figure 21.1). In the case where the motion is one-dimensional (e.g., purely horizontal and the target is a vertical strip of width W, at a distance D from the starting point), the average time taken to move from the starting point to a target point in the strip obeys the rule

$$T = a + b \log\left(1 + \frac{D}{W}\right). \tag{21.1}$$

The b factor is an adjustment for units (the logarithm is unitless, but it needs to be converted to seconds) and for the base of the logarithm; the a term represents the minimum time for any task—it accounts for the time it takes to perceive and understand the task, to convert this understanding into a nerve activation, etc.

For most applications in interface design, the details of the law are unimportant. But a few general principles can be derived from the law.

- Large targets are easier to hit than small ones, especially when the "largeness" is in the direction of necessary motion.
- Closer targets are easier to hit than remote ones of the same size.

Furthermore, careful measurement shows that the constant b is device- and action-dependent: Moving a mouse pointer and moving a pen tip involve different constants; dragging with the mouse is slower than simply moving and then clicking.

As you think about a cursor-based interface design, with the cursor controlled by a pen, for instance, you should ask yourself, "What things am I most likely to do with the pen?" and "How can I make these things easy to accomplish?"

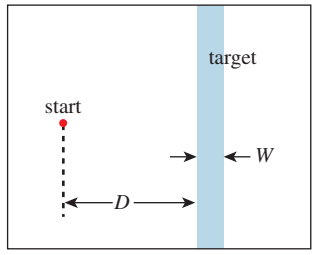


Figure 21.1: In the Fitts' Law experiment, the user must move a pointer (real or virtual) from the red start point at left to the blue strip at right as fast as possible.

The answer to the first question is application-dependent, but the answer to the second is more generic. For instance, we can make the simple observation that among all locations on the screen, the one most rapidly reachable by the cursor is the cursor's current location (see Figure 21.2). The next most reachable points are the four corners of the screen, because of the convention that the pen cursor never moves outside the screen: A motion to any point in the infinite quadrant associated to a corner requires no real precision in either the horizontal or vertical dimension. The four edge strips are similarly easy to reach, although they require some control in either the horizontal or vertical dimension.

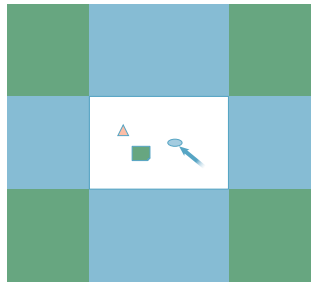


Figure 21.2: The quadrants (green) associated to corners are easy targets for cursor motion; the strips (blue) associated to screen edges are also good.

One consequence of the "point beneath the cursor is easy to reach" idea is that **pie menus** (menus that appear beneath the cursor, in which a drag into one of several sectors selects an option) are extremely easy to access (see Figure 21.3). Adjusting the sector sizes makes selecting common operations even easier, and muscle memory lets advanced users select from such menus without even looking at them.

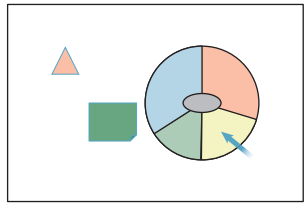


Figure 21.3: A pie menu. Different sector sizes make some options easier to choose than others.

A consequence of the "corners and edges of the screen are good targets" idea is that placing menus for all programs at the top of the screen may make interaction more efficient than locating them at the tops of individual windows. Of course, the initial interaction with a previously inactive program may be slower: The program must first be selected to activate it, and thus place its menu at the top of the screen. By contrast, in the "menus in windows" model, the program selection and menu selection may be combined into a single action.

By the way, generalizations of Fitts' Law give us estimates of the difficulty of reaching two-dimensional targets [GKB07], and of steering through a narrow (possibly winding) channel to a goal [AZ97], a result that's been discovered independently in several disciplines [Ras60, Dru71]. Fitts' Law also seems to extend quite naturally to multitouch devices [FWSB07, MSY07]. These extensions, too, can be used to guide your designs.

21.2.2 Interaction Event Handling

You've written programs in which clicking a button on the interface, or selecting a menu item, caused something to happen. The 2D test-bed program described in Chapter 4 contains examples of such interaction. The method used there is overriding methods. There's a `Button` class with a `buttonPressed` method that does nothing. We create a new class in which `buttonPressed` is overridden to do something useful for us. The system watches for events like a button press, and when they occur it invokes the appropriate method.

There are alternative approaches. In some object-oriented programming approaches, objects can respond to messages sent to them, rather than having methods that can be invoked. When a button is created in such a system, it's told what message to send and where to send it, in response to a button press.

In some non-object-oriented systems, you pass a function pointer to a procedure that creates a button. When the button is pressed, the function is called.

These are all just minor variations on a single theme. At a lower level, the fact that the mouse button was pressed at all must be noticed and handled. There are basically two approaches. In one, the button press generates an interrupt, and an interrupt handler is invoked to determine the location of the cursor and then dispatch the event to the appropriate button, for instance. In another, the button press enqueues an event on an event queue, which an interaction loop is constantly

polling—checking to see if there are new events to be processed. (The distinction is similar to that between preemptive and cooperative multitasking.)

Your choice of programming language, hardware, and operating system may influence which variety of system you end up using. But none of these substantially restrict general interface functionality: It's usually possible to get the same results in all cases.

In all the examples that follow, we use click-and-drag functionality: Some location(s) is/are selected, the location point is moved, and as a result something else is changed. Finally, the selection is released. In the 3D manipulation examples, the location selection comes from a mouse click, the move comes from a mouse drag, and the mouse-button release terminates the selection. In the photo-manipulation example, the selection comes from a finger contact, the move comes from contact motion, and the release comes when the finger is lifted from the interaction surface. But in all cases, there are multiple *states* of the system:

- The pre-interaction state
- The "selected" state
- The "dragging" state
- And the post-interaction state

In practice, we reduce this to two states: noninteraction and dragging. The course of a typical interaction can be described by a finite-state automaton (FSA) with these two states and four arcs (see Figure 21.4).

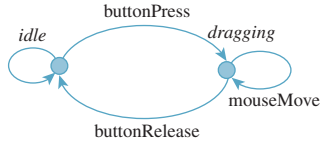


Figure 21.4: We are usually in the idle state. A click transitions to the dragging state; dragging remains there; a button release returns us to the idle state.

In general, FSAs provide a good structure for planning interaction sequences, which are seldom as simple as these. Unfortunately, as post-WIMP interactions evolve, the associated FSAs can become impossibly complex (imagine the FSA that might describe all possible interactions with your robotic butler in the future!), but for WIMP interactions, they can be a very useful tool.

21.3 Multitouch Interaction for 2D Manipulation

Multitouch interfaces are becoming increasingly common. We manipulate pictures on our smartphones using a thumb and index finger to translate and scale the pictures, for instance. Let's consider the implementation of this 2D manipulator, represented schematically in Figure 21.5.

Notice three things about the interaction.

1. The position of the touch points in the image remains approximately constant. In the first case, the initial touch was a little above and to the left of center; after the move, it remains in the same place.
2. In the move-and-scale interaction, the fingers widen more horizontally than vertically, but we have to choose a single scale amount. One alternative is to resize the image to accommodate the larger change. Another alternative is to average the horizontal and vertical widening fractions (i.e., a vertical stretch of 20% and a horizontal stretch of 30% would result in a uniform scale of 25%). A third possibility, and the one we choose, is to scale by the ratio of contact distances: If the distance between the contacts doubles, we scale by a factor of two.

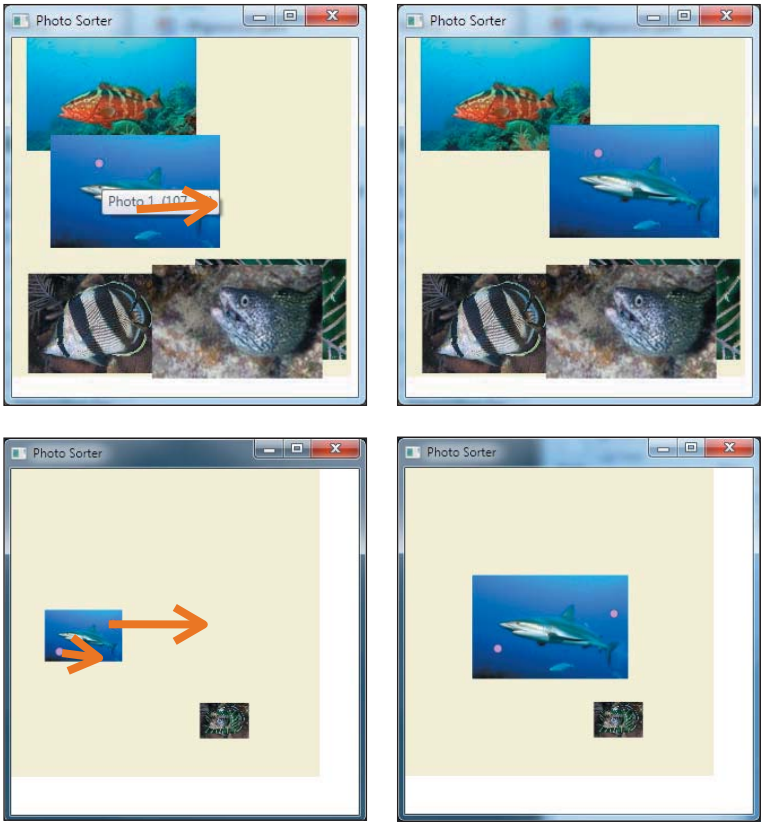


Figure 21.5: A photo-manipulation interface. (Top) A touch (shown by the pink dot in the shark photo) and drag (the large orange-brown arrow) moves a photo to a new location. (Bottom) Two contacts are spread apart to move and enlarge the photo. (©Thomas W. Doeppner, 2010.)

3. We've chosen to scale uniformly, even though nonuniform scaling of photos makes sense. That's because nonuniform scaling is so much less common, and it's so difficult to move your fingers in exactly proportional amounts, that it makes more sense to restrict to uniform scaling for convenience.

21.3.1 Defining the Problem

There are many possible ambiguous situations that still remain. What happens when the user starts by grabbing the upper-right and lower-left corners, and rotates these contacts to the upper left and lower right, respectively? According to rule 1 above, the picture should flip about its vertical axis to maintain contact-point correlation, but that's a nonuniform scale, which contradicts rule 2. We in fact choose rule 3 rule as the dominant one, since opening and closing the fingers is much easier than rotating the hand, and so the inconsistency of contact points isn't likely to be a problem in general.

By how much should we translate the photo during a two-finger interaction? We could translate the photo so that the first contact point remained underneath *its* finger, but the other perhaps did not. We could translate by the average of the

two contact-point translations. We could translate so as to preserve the lower-left contact point, whether it's the first or the second, on the grounds that for right-handed people, this is likely to be the thumb contact. We'll choose the second, but there's a good argument to be made for each of the others. The only way to decide conclusively is through user testing.

Now we have a complete problem definition: We'll translate the photo so that the midpoint of the two contacts moves as specified, and we'll scale it about that midpoint by the ratio of the contact distance after to the contact distance before.

The mathematical portion of the solution is now straightforward: We first scale the object about the initial midpoint, and then translate that midpoint to its new location.

21.3.2 Building the Program

To place our photo manipulator in context, we'll assume that there are several photos in a scene, represented by a very simple scene graph: a "background," representing an infinite canvas on which the photos are placed, with a global translate-and-scale transformation, and n photos, each with its own translate-and-scale transformation (see Figure 21.6). We "see" the parts of the photos that, after transformation, are visible in the unit square $0 \leq x, y \leq 1$. When we manipulate a particular photo (or the background), we will alter its transformation and none of the others.

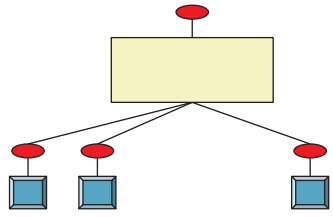


Figure 21.6: The background canvas (yellow rectangle) has its own scale-and-translate view transformation (the top red ellipse), and each photo (blue square) has a scale-and-translate as well.

We'll assume that the manipulation is to be done in the form of callbacks, one for each contact event, where a contact is the touch of a digit to the interaction surface: We get informed when there's a new contact, a contact drag, and a contact release. When two contacts move at once (as in the move-and-resize action), we'll get a callback for each one (in no particular order). Each callback will identify the contact with which it's associated. And at the start of the photo-manipulator application, the program will register with the operating system to receive callbacks for all such interactions.

When, for instance, a touch and drag begins, the application's new-contact callback will be invoked; it handles this by creating an `Interaction` object to handle the remainder of the interaction sequence. That interaction object registers for subsequent callbacks, and after receiving each and processing it, marks it as having been handled so that no other registrants like the application itself get that callback. When the interaction is completed (by a contact-release event), the interactor can unregister itself, and subsequent callbacks will once again go to the application (see Figure 21.7).

21.3.3 The Interactor

The interactor, at initialization, must do the following.

1. Identify which photo is being manipulated (and if the contact is not within a photo, record that the *background* is being manipulated).
2. Record the initial point of contact.
3. Record the initial transformation $\mathbf{T}_0$ for the photo or background.
4. Keep a reference to the transformation for the selected photo (or background) in the scene graph. (Because we do the same thing whether a photo or the background is selected, we'll refer to the selected photo from now on.)

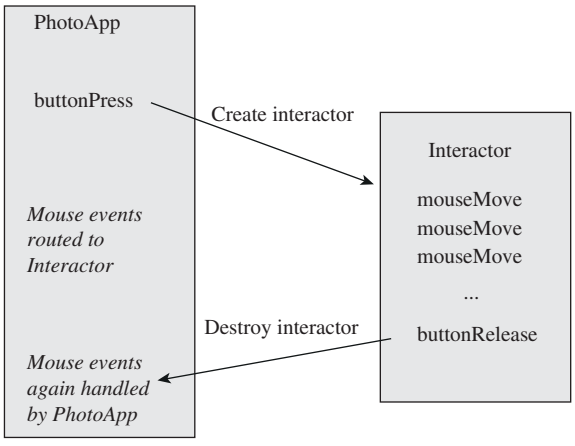


Figure 21.7: Callbacks to the application result in the creation of an interactor, which handles subsequent callbacks until done.

Let's imagine, for the time being, that the interaction is a simple single-finger click and drag, with no scaling involved. Then our strategy for implementing the interaction during dragging is as follows.

1. At each drag, compute the offset **d** between the current contact and the initial contact.
2. Let **T** be the transformation which is translation-by-**d**.
3. Replace the transformation for the selected photo with $\mathbf{T} \circ \mathbf{T}_0$ (i.e., first do whatever transformations were done previously, and then translate by **T**).

Notice that rather than accumulate incremental motions, we use the offset from the original point. Accumulating increments can also work, but numerical problems may make the sum of the increments different from the total motion for long drag sequences, making the photo appear to "slip" around the contact point that's being dragged. We discuss this further in the case of virtual sphere rotation.

Note that in either case—accumulated incremental motions, or a single translation determined from the start point and current point—the translation is composed with the existing transformations on the photo, and thus should be described as a "relative" transformation rather than an absolute one.

At the end of the interaction, when the contact is broken, we need only destroy the interactor.

The code outline, in an informal approximation of C#, is shown in Listing 21.1.

Listing 21.1: Outline of interaction code for photo manipulation application.

```
1  Application:
2     main()
3        build scene graph for photos and display the scene register newContact, dragContact,
4        releaseContact callbacks
5
6     public newContactCallback(Scene s, Contact c)
7        Interaction ii = new Interactor(c)
8
```

```
Interactor:
   private Contact c1
   private Transform2 initialXform
   private Point2 startPoint
   private FrameworkElement controlled
   private PhotoDisplay photoDisplay

   public Interactor(Contact c)
      c1 = c
      intialPoint = c1.getPoint()
      controlled = the photo (or background) that's at initialPoint
      initialXform = controlled.getTransform()

      register for all contact callbacks

   public dragContactCallback(Contact c)
      if c1 != c {signal an error}
      Vector2 diff = c.getPoint() - initialPoint
      Transform2 T = new Translation(diff)
      s.setTransform(o, initialXform*T)
      redisplay scene

   public releaseContactCallback(Contact c)
      if c1 != c {signal an error}
      unregister this interactor for callbacks
```

We're assuming here that we have point, vector, and transformation classes, and that composition of transformations is represented by the overloaded `*` operator, in which `S * T` is the transformation that applies *S* and *then T*. Furthermore, we assume that each object (photo or background) stores its own transformation, rather than the transformations being stored in a scene-graph object.

All of these assumptions hold in WPF, and a WPF implementation of this photo manipulator is available on this book's website. Rather than using actual multitouch contacts, which may not be available to all readers, the program simulates them by letting the user right-click to create or destroy a "contact" (shown as a small marker) and then left-click and drag to move contacts.

WPF also provides pick correlation—a report of which object in a scene is visible at the pixel where the user clicks, as needed at line 19 in Listing 21.1.

What changes must be made to allow for two-contact interaction? When the second contact happens, we'll treat the first contact's click-and-drag sequence as having terminated (i.e., we'll start from the current photo's current transformation, and forget that we ever had an initial transformation or contact point).

For a two-contact interaction, we'll (a) treat the *midpoint* of the two contacts as pinned to the photo so that when the midpoint moves, the photo moves, and (b) scale the photo relative to the distance between the fingers so that if the fingers move together the photo is unscaled, and if they widen the photo enlarges, etc. We'll record the midpoint and vector difference of the contacts at the start, and at each update we'll build an appropriate scale-and-translate transformation. In other words, we'll do just what we did for the single-contact click and drag, but now we'll do it by remembering the initial positions of *two* contact points, and we will include scaling.

Because the interaction sequence might look like "touch with one finger, drag to the right, touch with the thumb as well, drag farther to the right and widen the distance of the finger to the thumb," we must also track the number of contact

points. Whenever this number changes, we'll restart our tracking. Listing 21.2 shows the differences, except for what happens when a contact moves.

Listing 21.2: Handling the varying number of contact points.

```
Interactor:
   private Contact c1, c2;
   private Transform2 initialXform
   private Point2 startPoint
   private FrameworkElement controlled
   private PhotoDisplay photoDisplay
   private Vector2 startVector

   public Interactor(Scene s, Contact c)
      c1 = c; c2 = null;
      startPoint = c1.getPoint()
      initializeInteraction()
      ...

   // if there's only one contact so far, add a second.
   public void addContact(Contact c)
      if (c2 == null)
         c2 = newContact(e);
         initializeInteraction();

   private void initializeInteraction()
      initialFform = controlled.GetTransform();
      if (c2 == null)
         startPoint = c1.getPosition();
      else
         startPoint = midpoint of two contacts
         startVector = c2.getPosition() - c1.getPosition();


   public removeContact(Contact c)
      if only one contact, remove this interactor
      otherwise remove one contact and reinitialize interaction
```

When a contact point moves, we have to adjust the transformation for the relevant photo. Listing 21.3 gives the details.

Listing 21.3: Handling motion of contact points.

```
public void contactMoved(Contact c, Point p)
   if (c2 == null)
      Vector v = p - startPoint;
      TransformGroup tg = new TransformGroup();
      tg.Children.Add(initialTransform);
      tg.Children.Add(new TranslateTransform(v.X, v.Y));
      controlled.SetTransform(tg);
   else
      // two-point motion.
      // scale is ratio between current diff-vec and old diff-vec.
      // perform scale around starting mid-point.
      // translation = diff between current midpoint and old
      Point pp = getMidpoint(); // in world coords.
      Point qq = startPoint;
      pp = photoDisplay.TranslatePoint(pp, (UIElement) controlled.Parent);
```

```
16        qq = photoDisplay.TranslatePoint(qq, (UIElement) controlled.Parent);
17        Vector motion = pp - qq;
18
19        Vector contactDiff = c2.getPosition() - c1.getPosition();
20        double scaleFactor = contactDiff.Length / startVector.Length;
21        TransformGroup tg = new TransformGroup();
22        tg.Children.Add(initialTransform);
23        tg.Children.Add(new ScaleTransform(scaleFactor, scaleFactor, qq.X,qq.Y));
24        tg.Children.Add(new TranslateTransform(motion.X, motion.Y));
25
26        controlled.SetTransform(tg);
```

This code uses several WPF conventions that deserve explanation. First, a `TransformGroup` is a sequence of transformations that are applied in order; thus, in the `if` clause, we first perform the initial transformation to the photo, and then translate it. Second, the line

```
pp = photoDisplay.TranslatePoint(pp, (UIElement) controlled.Parent)
```

transforms the point `pp` from the world coordinate system (that of the `PhotoDisplay`) to the coordinate system of the parent of the current photo (the background canvas). In the case where the background canvas is being manipulated, it transforms the point to the coordinate system of background's parent, that is, the `PhotoDisplay`. Thus, the computed translation `qq - pp` is the one to apply after the photo has been scaled, but before it is further transformed by the transformation associated to the background. It's essential that the point `pp` start in world coordinates for this to work properly. If it were, say, in the coordinate system of the photo, we'd have to transform it to the photo's parent.

21.4 Mouse-Based Object Manipulation in 3D

The same general approach—build an `interactor` that handles a click-and-drag sequence by editing the transformation on a target object—works in 3D as well. A closely related idea is that the relationship of object to view is symmetric: In a view of a scene with only a single object, we can move the object to the right, or the camera to the left, and get the same change in the eventual image. Thus, a slightly modified version of the interaction we use for object manipulation can be used for camera manipulation.

21.4.1 The Trackball Interface

In the trackball model, we imagine that an object is suspended in a transparent solid ball with center C that can be rotated by the user; a click and drag on the ball's surface, from a starting point A to an endpoint B, defines a rotation: The ball is rotated in the plane of A, B, and C, with C as the center of rotation, so as to move A to B. (This is also called the **virtual sphere** model.)

Inline Exercise 21.1: In terms of A, B, and C, describe the axis of the rotation, and the angle.

Inline Exercise 21.2: Under what conditions on A, B, and C is the rotation ill-defined? Can you think of a situation in a typical interaction where this might be a problem, or will such a problem never arise?

Inline Exercise 21.3: ◈ We've specified the rotation quite carefully. If A and B are two distinct but nonantipodal points of a sphere, describe the set S of rotations of the sphere that take A to B. Is S a finite set? The space $\mathbf{SO}(3)$ of sphere rotations is three-dimensional. Is S a zero-, one-, two-, or three-dimensional subset of it?

In this interaction sequence, a right-click on the object (our demonstration example has only a single object) makes a transparent sphere appear surrounding the object; a first left-click on the sphere initializes a rotation action; and a drag to a new point defines a rotation, which is applied to the object so that it appears to be dragged within the transparent sphere. The mouse release makes the currently applied rotation permanent. By the way, **undragging** (i.e., returning to the starting click point) resets the transformation to its initial value.

In the implementation, we need to do three things.

1. Create the transparent sphere and respond to click, drag, and release events there.
2. Handle a click event by recording the current transformation on the object, and storing the initially clicked point. It's best to store this in the frame of reference of the object at the time of clicking.
3. Handle drag events by transforming the current mouse position into the frame of the object at initial-click time, and then computing the rotation that takes the initial click to the current mouse position. This rotation is applied to the object, followed by its pre-drag transformation.

There is one tricky problem: What happens when the drag leaves the sphere? For this, we project back onto the sphere: We find the sphere point closest to the eye-through-cursor ray, and pretend that the cursor is there.

With this in mind, let's look at the code. We start by creating a scene (see Figure 21.8) containing a single manipulable object, a cube. If pick correlation shows a right-click on the cube, we create an interactor to handle the subsequent interactions:

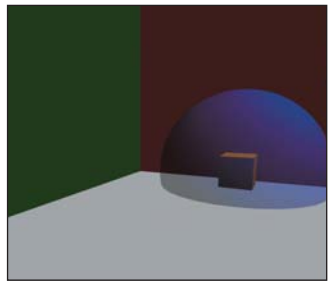


Figure 21.8: A floor and two walls, and a cube that can be rotated.

```
1  public partial class Window1 : Window
2      private RotateTransform3D m_cubeRotation = new RotateTransform3D();
3      private ModelVisual3D m_cube1;
4      private Interactor interactor = null;
5       public Window1()
6          // initialize, and build a ground and two walls
7          m_cube1 = a cube model
8          m_cube1.Transform = new TranslateTransform3D(4, .5, 1);
9          mainViewport.Children.Add(m_cube1);
10
11         this.MouseRightButtonDown +=
12                         new MouseButtonEventHandler(Window1_MouseRightButtonDown);
```

```
13          add handlers that forward left-button events to the interactor, if it's not null.
14
15      void Window1_MouseRightButtonDown(object sender, MouseEvent e)
16          // Check to see if the user clicked on cube1.
17          // If so, create a sphere around it.
18          ModelVisual3D hit = GetHitTestResult(e.GetPosition(mainViewport));
19          if (hit == m_cube1)
20              if (interactor == null)
21                  interactor = new Interactor(m_cube1, mainViewport, this);
22              else
23                  endInteraction();
24           // if there's already an interactor, delegate to it.
25          else if (interactor != null)
26              interactor.Cleanup();
27              interactor = null;
```

The interactor, just as in the photo-manipulation example, keeps track of the manipulated object (`controlled`) and the transformation for that object at the start of the manipulation. We also note the viewport from which the object is seen (which allows us to transform mouse clicks into rays from the eye). Initializing the interaction consists of recording the initial transformation on the controlled object, and creating a transparent sphere, centered at the object center. The corresponding cleanup procedure removes the sphere.

```
1  private void initializeInteraction()
2      initialTransform = controlled.Transform;
3      find bounds for selected object,
4      locate center and place a sphere there
5      viewport3D.Children.Add(sphere);
6
7  public void Cleanup()
8      viewport3D.Children.Remove(sphere);
9      initialTransform = null;
```

When the user left-clicks on the sphere, we record the current transformation associated to the controlled object and the location of the click. Just as in the photo-manipulation program, we record this position in the coordinate system of the *parent* of the controlled object. We also record that we are in the midst of a drag operation, and when the left button is released, we reset the drag status.

```
1  public void mouseLeftButtonDown(System.Windows.Input.MouseButtonEventArgs e)
2      ModelVisual3D hit = GetHitTestResult(e.GetPosition(viewport3D));
3      if (hit != sphere)
4          return
5      else if (!inDrag)
6          startPoint = spherePointFromMousePosition(e.GetPosition(viewport3D));
7          initialTransform = controlled.Transform;
8          inDrag = true;
9
10 public void mouseLeftButtonUp(System.Windows.Input.MouseButtonEventArgs e)
11     inDrag = false;
12
13 private Point3D spherePointFromMousePosition(Point mousePoint)
14     form a ray from the eye through the mousePoint
15     if it hits the sphere
16         return the hit point.
17     else // ray misses sphere
18         return closest point to ray on the sphere
```

Finally, just as before, the meat of the work is done when the mouse moves: We find the new location of the mouse (in the coordinate system of the controlled object's parent), build a rotation in that coordinate system, and append this rotation to the controlled object's initial transformation.

```
1  public void mouseMove(System.Windows.Input.MouseEventArgs e)
2     if (inDrag)
3        Point3D currPoint = spherePointFromMousePosition(e.GetPosition(viewport3D));
4        Point3D origin = new Point3D(0, 0, 0);
5        GeneralTransform3D tt = initialTransform.Inverse;
6        Vector3D vec1 = tt.Transform(startPoint) - tt.Transform(origin);
7        Vector3D vec2 = tt.Transform(currPoint) - tt.Transform(origin);
8        vec1.Normalize();
9        vec2.Normalize();
10       double angle = Math.Acos(Vector3D.DotProduct(vec1, vec2));
11       Vector3D axis = Vector3D.CrossProduct(vec1, vec2);
12       RotateTransform3D rotateTransform = new RotateTransform3D();
13       rotateTransform.Rotation = new AxisAngleRotation3D(axis, 180 * angle/Math.PI);
14
15       Transform3DGroup tg = new Transform3DGroup();
16       tg.Children.Add(rotateTransform);
17       tg.Children.Add(initialTransform);
18       controlled.Transform = tg;
```

Before leaving the trackball interface, let's examine some of the design choices and variants. First, initiating a rotation requires clicking on the object. That in turn requires moving the pointer so that it appears over the object. Fitts' Law tells us that this may be a somewhat costly operation if the object is far from the current pointer location. On the other hand, shifting our attention to the object happens at the same time, so we can perhaps regard some of the cost as amortized. Having selected the object with the first click, we then rotate it with a drag, which is ideal from a Fitts' Law perspective: The drag starts at the most easily accessible location, the current pointer position. How large a drag is required? That depends on the radius of the virtual sphere: A rotation of 90° will require a cursor motion equivalent to the sphere's projected radius. This suggests that a small radius is ideal. On the other hand, precisely placing the cursor within that small radius can be difficult; a larger sphere gives the user more precise control of the rotation. Depending on which is more important for the context, speed or precision, the designer should adjust the standard interaction-sphere size.

On the mathematical level, we've chosen to work with an integral form of the interface: The initial point is clicked, and the rotation of that point to the current point is recomputed for each bit of dragging. As an alternative, we could have used a differential version, in which the motion from the previous cursor point to the current one is used to generate a tiny rotation, and these tiny rotations are accumulated by multiplying them into the transform of the object. Unless the cursor moves along a great circle arc during the drag, the differential and integral forms give different results. In the differential version, making small circles about the initial click point generates a spin about that point; making circles in the opposite direction generates the opposite spin. Users sometimes find this useful. On the other hand, in the integral form, a drag that ends at the initial point always brings the object back to its starting orientation, which users may also find useful.

In the differential form, we "accumulate" many small rotations by multiplying them together in the form $\mathbf{R}_1\mathbf{R}_2\mathbf{R}_3 \ldots \mathbf{R}_k$, where k can be quite large. While each $\mathbf{R}_i$ may be a rotation matrix within the bounds of numerical precision, their

product may end up differing from a rotation by a large amount because of roundoff errors; the result can (and should!) surprise the user. A solution to this is to accumulate the rotations and then, after perhaps ten are accumulated, reorthogonalize the matrix with the Gram-Schmidt process.

Even with this reprojection onto the set of rotation matrices, the differential form has another drawback. The exact same cursor click-and-drag sequence, executed on two identical scenes, may produce different results. That's because the mouse motion is sampled by the operating system, and depending on other loads on the machine, the samples may not occur at exactly the same moments. Thus, the two sequences of points used to produce the two sequences of rotations may differ slightly, and the final results will generally differ as well. This is not usually a problem unless the load is rather high so that sampling occurs at a rate that fails to accurately represent the cursor path. For example, if the cursor is moved in a small circle over the course of a half-second, but only two position samples are taken during that time, the results will be very different than if ten samples are taken.

 In general, it's a bad idea to try to numerically integrate differentials, or even very small differences, for the reasons given above. There are two exceptions. First, such an integral may be the only practical way to compute a value. In studying light transport, for example, computing the light arriving at the eye amounts to evaluating an integral, one for which the only known methods are numerical (see Chapter 31). The second is where the summed quantity is known to be an integer; in this case, roundoff errors, if they're known to be small, can be removed by rounding. (For instance, if you sum four terms and get 3.000013, you can safely assume that the value is 3.)

21.4.2 The Arcball Interface

The arcball interface [Sho92] is exactly like that of the trackball, except that the sphere rotates *twice* as far as the drag would suggest. That is to say, if you drag from A to B, and they're 30° apart on the sphere centered at C, the object will rotate 60° in the plane of A, B, and C.

This has several practical implications. First, even though we can only see and click on the front half of the sphere, we can perform every possible rotation: Dragging the nearest point to the contour rotates it all the way to the farthest point, for instance. Second, dragging from P to Q, then Q to R, then R to P (where all three are points on the sphere) results in no rotation at all.

In evaluating the arcball, much of what we said about the virtual sphere still holds. If the interaction sphere is textured with some recognizable pattern, such as a world map, then there is some surprise for the user who clicks and drags London: During the drag, London slides out from under the cursor. With a transparent sphere, this effect is largely invisible, however, and the interaction feels quite natural. (If we were to implement a translation-by-dragging interface and translated by twice the drag vector, it would almost certainly be disconcerting to the user, however.)

21.5 Mouse-Based Camera Manipulation: Unicam

We now move on to the topic of manipulating the view of a scene. It's easy to imagine that this is just the same as manipulating an object; after all, the camera

transformation is part of the scene graph in exactly the same way that the transformations on objects are. In more basic terms, if we want to see the left-hand side of a box, we can either rotate the box to the right or move our eyes to the left. It's not very difficult to adapt the trackball or arcball interface to act on the scene as a whole rather than on a particular object, and thus achieve this effect. For a square viewport, we simply draw a manipulator sphere that touches all four sides of the viewpoint, thus giving maximal precision in control of the camera. Unfortunately, when we do so we find it's not very satisfactory: The camera keeps tilting away from "upright," and while being able to make a single object tilt is convenient, having a tilted camera is so rarely what's wanted that it's a constant annoyance. This is a situation where context (the traditional human experience of having the vertical almost always be "up" in our view of the world) should influence design.

Furthermore, camera control involves more than just the orientation of the camera: You may wish to look somewhere else (at some other object), or get closer to the object you're looking at. For the first of these **(panning),** there's a fairly natural interaction: You can click on the object you want to look at and drag it to the center of the screen. If it's off-screen, multiple panning steps may be needed. Of course, you need to do something to indicate that you're panning rather than rotating; that is, you need a notion of "mode." For the second **(dollying),** there's no obvious interaction, even once you've established you're in dollying mode.

Unicam [ZF99] is a camera-manipulation mechanism that allows for controlling the three rotational degrees of freedom and the three translational degrees of freedom in a virtual camera with a single integrated system. Other features common to virtual cameras (clipping, plane distances, view angle, and film-plane rotations for view-camera effects) are so rarely adjusted that they are not included, just as we omitted image rotation in our photo-manipulation application. The implementation is so very similar to that of the other manipulators we've described that we'll simply describe how the interface feels to the user. Unicam can actually be used for both perspective and orthographic cameras, but we'll only describe the more common perspective camera case here.

Because Unicam is designed for applications in which camera control is a frequent operation (e.g., solid modeling), a single mouse button is entirely allocated to it: All camera operations are performed by click and drag with this one mouse button. This reduces the transition time and effort when the user wants to switch between camera operations and other application operations controlled by other mouse buttons.

With Unicam, the viewing window is divided into two regions (see Figure 21.9): an inner rectangle in which interactions determine camera translations and a border where they determine rotations.

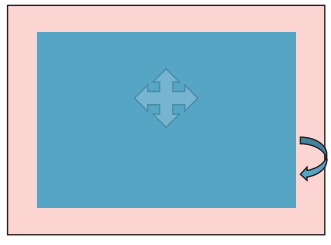


Figure 21.9: In the blue inner rectangle, mouse motions induce camera translations. In the pink border region, mouse motions determine rotations.

Unicam maintains a notion of a **hit point,** a place that represents the location of the user's focus of attention. Typically, this is the scene point under the cursor (i.e., the first point hit by tracing a ray from the eye through the cursor point on the film plane into the scene). In the event that this ray hits nothing in the scene, the hit point is the projection of the *previous* hit point onto this ray.

21.5.1 Translation

A click and drag in the translation area is initially classified as "horizontal" or "vertical" by examining the first few pixels of motion. (The authors suggest about 1% of the screen width as a reasonable distance to use in determining primary

direction, and say that this categorization must be done in the first 0.1 sec to avoid disturbing the user.) An initially horizontal motion introduces a camera translation in a direction parallel to the film plane in such a way that the hit point remains beneath the cursor. Thus, a click and drag to the right causes the camera to move to the left in the scene so that the hit point moves the appropriate distance to the right in the resultant image.

Inline Exercise 21.4: How would a user move the scene up and down rather than left and right?

An initially vertical cursor motion indicates a different mode of interaction. Left-right motion continues to act as before, performing film-plane-parallel motion to the left or right, but vertical motion translates the camera along the ray from the camera to the hit point. The authors make an interesting choice for how cursor motion is converted to translation toward the object: The conversion is linear, with a motion from the bottom to the top of the interaction window corresponding to the distance from the camera to the hit point. This makes it impossible to "overshoot" the hit point, but makes it easy to approach the hit point with a kind of logarithmic interaction: Multiple half-screen vertical cursor motions each divide the distance to the hit point by two.

The assignment of *vertical* cursor motion to dollying is an apparently arbitrary choice; the authors could have chosen to use horizontal motion. But they report that users find the vertical motion far more natural, perhaps because we are familiar with scenes like that shown in Figure 21.10, in which the horizontal layout of the terrain makes the correspondence between vertical position and distance obvious. (Try to think of a situation in which there's a similarly strong relationship between horizontal position and distance; is it a commonplace or familiar situation?)

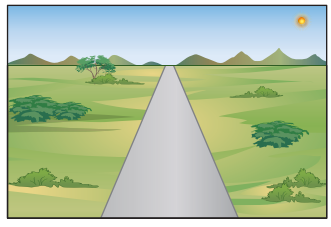


Figure 21.10: Vertical position, in this scene, corresponds to distance from the viewer.

21.5.2 Rotation

While there are three rotational degrees of freedom, such rotations must have a center of rotation. (A rotation about one center can be converted, by a translation, into a rotation about any other center, but we need a particular center to start from.) The camera location itself is one possible center of rotation, and it corresponds well to our physical structure, in which you can bend your neck to look up or down, and can rotate it to look left and right. But when your attention is focused on some object, "orbiting" around the object feels more natural than turning your head and then stepping to the side to bring the object back into view. In Unicam, a click and release on a scene object places a small blue sphere (the **focus dot**) at the hit point, and subsequent rotations are all interpreted as rotations about this focus dot.

Alternatively, the user can click in the border area to invoke rotation about the **view center,** a point on the ray from the camera through the center of the view. The distance *along* that view ray is determined by the current hit point: The perpendicular projection of the hit point onto the view ray is the view center.

In the case of a focus dot, a subsequent click and drag anywhere on the view begins a rotation; in the case of a view-centered rotation, the initial click in the border area initiates the rotation, and subsequent drags determine the amount

of rotation. In each case, rather than using virtual-sphere or arcball rotations, the *x*- and *y*-coordinates of this mouse displacement from its initial click determine, respectively, rotation about the world "up" vector (usually *y*) and about the camera's "right" vector (i.e., the vector pointing to the right in the film plane). Full-screen-width horizontal motion corresponds to 360° rotation about the up vector; full-screen-height motion corresponds to 180° rotation about the right vector, although this rotation is clamped to prevent ever arriving at a straight-up or straight-down view. The rotations are implemented sequentially: first a rotation about the up vector, then about the right vector.

21.5.3 Additional Operations

The focus dot also serves as a focus for further interactions: Clicking and releasing on the focus dot moves the camera to an oblique view of the underlying object, seen from slightly above the object. A click and drag up and to the right saves the current view into a draggable icon that can later be clicked to restore the view. Dragging down and to the right temporarily scales the focus sphere by enlarging its radius to the drag distance; upon release, the camera dollies inward until this enlarged sphere fills the view, at which point the focus sphere returns to its normal size. This allows the user to easily specify a region of interest. A drag in any other direction aborts the gesture.

21.5.4 Evaluation

Unicam presents the user with very easy access to the most common camera operations. By having many of the gestures start at the current cursor location, it takes greatest advantage of Fitts' Law. By associating actions with a direct-manipulation "feel" (translation by dragging feels as if you are dragging the world with the cursor, and dollying by vertical motion feels like you're moving along a train track toward its vanishing point), the designers make the operations easy to use and remember.

On the other hand, there are no affordances in the system. There's nothing that tells you that the view's border area can be used for rotation, or that its center can be used for translation. For an often-used feature like camera control, this is probably appropriate. Through constant use, the user will rapidly memorize its features. For controls that are used less often, some visual representation would be appropriate.

21.6 Choosing the Best Interface

We've seen two object-rotation interfaces and a camera-control interface. Many games provide camera controls that simply let you look left or right (by fixed increments) or up and down (by fixed increments), often controlled by keyboard keys. Architecture walkthrough programs let the user move through a building, by typically constraining the eye height to something near 1.8 m, and prevent motion that passes through walls, etc. Which interface is best? The answer is that among well-designed interfaces (e.g., ones that pay attention to matters of affordance and Fitts' Law), the best choice almost always depends on context. In an architecture walkthrough application, the camera-control interface should restrict the eye height and prevent passing through walls; in a CAD/CAM system for designing

an aircraft, being able to view places that are inaccessible to humans (e.g., the cable-routing channels in the airframe) is essential, and eye height and collision-prevention elements in an interface would be annoying.

21.7 Some Interface Examples

In this section, we briefly describe some of our favorite interaction work. The results range from items you'll want in your toolbox of ideas to ones that are single-application interfaces where the interface is enabled by new underlying graphics technology. Other good ideas, like pie menus, tool trays, and Unicam, have already been described elsewhere in this chapter, and there are so many good ideas that we cannot possibly be exhaustive here. This is an idiosyncratic list of ideas we've found important, useful, or inspiring.

21.7.1 First-Person-Shooter Controls

These FPS controls provide keyboard control of view and camera motion in many video games. They make a nice addition to any other camera control mechanism you have in your program: They're easy to learn and widely applicable. In one form, they use the arrow keys: The up and down keys move the viewer forward and backward; the left and right keys typically "strafe" to the left and right, although they can also be used to turn the view to the left or right. If you want to have nearby keys perform related functions (Fitts' Law applies to the keyboard as well as the mouse), the arrow keys are less convenient. Instead, it's typical to use W and S for forward and backward motion, Q and E to rotate the view to the left or right, and A and S for strafing (which, in nonshooting games, can be remapped to "peeking" to the left or right—the view is shifted somewhat to the left or right for the duration of the keypress, and it returns to a forward view when the key is released).

21.7.2 3ds Max Transformation Widget

The ViewCube [KMF+08] is a 3D view manipulation widget (see Figure 21.11). It was developed by Autodesk and has been deployed in all of its 3D modeling products, which include AutoCAD, 3ds Max, Maya, and Mudbox. This makes it one of the most significant 3D user-interface elements in use today. The ViewCube was designed to address a long-standing problem in 3D modeling that has only grown as the popularity and importance of CAD and digital content creation have brought more designers in from 2D tools: user disorientation. The often-ambiguous third-person view of an untextured and often unfinished scene can easily leave the user without a sense of orientation or broader context for content creation applications. This is less problematic in applications like games, where a polished surrounding environment and strong lighting cues provide intuitive orientation cues.

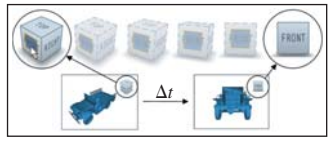


Figure 21.11: The basic modeling widget from 3ds Max (Courtesy of Azam Khan, ©2008 ACM, Inc. Reprinted by permission.)

The ViewCube always sits in the upper-right corner of the screen. It both provides intuitive orientation feedback and acts as a camera control widget. The orientation feedback is in the form of a subtle drop shadow indicating vertical orientation and explicitly labeled faces. The researchers who developed the ViewCube experimented with several alternatives to the text labels, such as embedding a small 3D view of the current object within the cube, but they found that the text

was most effective. The 8 corners, 12 edges, and 6 faces of the cube each correspond to specific views. A user can click on zones near any of those with a mouse to warp to the predefined viewpoints relative to the center of the cube, or click and drag to rotate the cube to an arbitrary orientation (in the style of the arcball [Sho92]). The outlines of the cube are stroked as solid paths when the cube is at one of the 26 canonical views and dashed for intermediate views. In addition, small arrows (not shown in Figure 21.11) point to the four peripheral faces (which may not be visible) and support 90° roll rotations in the plane of the current view.

21.7.3 Photoshop's Free-Transform Mode

When you are in free-transform mode in Photoshop and you select an image, its bounding box is shown with small square "handles" at the corners and edges. As the cursor moves over these handles, it changes to a double-headed arrow, disclosing that you can click and drag the handles. Corner drags reshape the bounding box (and its content) in both x and y; a shift-key modifier makes the changes in width and height be proportional. A control-key modifier lets the corner (or edge) be moved to any position, so the image is no longer rectangular. Edge drags move the selected edge; a shift modifier makes the opposite edge adjust as well so that a shift edge drag on the top scales the image around its horizontal centerline; a control-shift modifier lets the user shear the image (i.e., move the edge center along the line containing the edge).

When the cursor is slightly outside the bounding box, it becomes a curved double-headed arrow, indicating that you can rotate the box and its contents (see Figure 21.12). Finally, if you click on a corner and press appropriate modifier keys, you can apply a perspective transform to "keystone" the bounding box in either the horizontal or vertical direction, giving the appearance of perspective (see Figure 21.13).

Figure 21.12: Rotating an image in free-transform mode.

21.7.4 Chateau

Chateau [IH01] is a system for rapidly creating highly symmetric forms. User input is processed to search for symmetries. For instance, if the user recently created a cylinder of length 5 and radius 1, and begins the gestures to create a new cylinder, indicating a length of approximately 5, the system offers up a completed cylinder in a thumbnail view, which the user can click to confirm that it's what's wanted. If there are multiple possible completions, the most likely (according to some heuristics) are offered. Once the second cylinder is placed somewhere, the system may propose a third cylinder, offset from the second in the same way the second is offset from the first, thus making it easy to create a row of columns, for instance.

While the particulars of this program are not especially relevant, the notion of a **suggestive interface,** in which candidate completions of actions are offered, leverages the "recognition is faster than recall" idea: The user can recognize the correct completion rapidly. Similar ideas are used in keyboard input for Asian character sets, where each character is represented by a quadruple of ASCII characters, but once the user types one or two ASCII characters, several "likely" choices are offered as completions, with likelihood being determined by things like recent use in the document, or even surrounding vocabulary or sentence structure.

Figure 21.13: The image has been keystoned by dragging a corner along the left side; the bounding box remains unchanged, however.

The auto-completion used in text-messaging systems on mobile devices is similar, offering multiple completions. In the T9 input system, using the conventional "2 = ABC, 3 = DEF, 4 = GHI . . . " mapping, a user types "432" and the system recognizes that the most likely word containing one of GHI, followed by one of DEF, followed by one of ABC, is "head" and offers it as a completion. The user can continue to type numbers ("54") to select a longer word like "healing." And in a radically different approach to text entry, the Dasher system [WBM00] (see Figure 21.14) displays text in boxes that approach a user-controlled point. When the point is moved to the right, the boxes move to the left, at a speed proportional to the displacement. As the point is moved up or down, the user can arrange for the point to pass through a particular box. Doing so produces the "keystrokes" shown in the box (typically a single letter). Using the statistics of the input language, the system places likely boxes near the middle, and unlikely ones at the top and bottom. In some cases, sequences of two or more characters may be very likely, and boxes containing those sequences end up "in line" so that it's easy to pass through all of them. (For instance, if the user starts by selecting a "T," the easiest two boxes to draw through are "h" followed by "e.") By training the system (thereby altering its notion of likelihood) or introducing a custom vocabulary, a user can make it even more effective. This is a suggestive interface that can reasonably be used by the severely disabled.

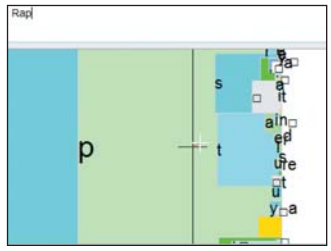
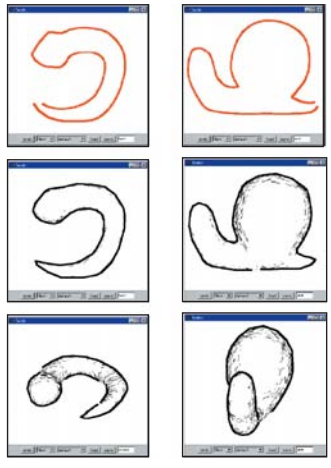


Figure 21.14: The user has chosen the characters R-a-p, shown in the upper left. The user will move the cursor upward so that the box labeled "t" passes over it, completing the word "Rapt."

21.7.5 Teddy

Teddy [IMT99] is a system for the informal creation of smooth or mostly smooth 3D shapes. The user makes gestures that are interpreted as 3D modeling commands. For instance, at the start, if the user draws a simple closed curve, it is interpreted as the silhouette of a smooth shape; an "inflation" algorithm converts the silhouette into 3D. A stroke drawn across a shape cuts off part of the shape, as if it had been sliced with a sword. If the user draws a closed curve on the surface, then rotates the object so that this is near the silhouette, and draws a curve starting and ending on the first one, the system creates an "extrusion" from the base shape using the first curve as the cross section, and the second to determine the shape of the extrusion. This allows the rapid creation of interesting shapes (see Figure 21.15).

The system is made possible by various mesh-construction and editing operations, but it is more notable for the coherence and simplicity of its interface design. By providing just a few simple operations, and making intuitive gestures to represent them, Teddy hits a sweet spot in shape creation. Not long after it was introduced, it was used as an avatar-creation interface in a video game, with thousands of users.



(a) Snake (a) Snail

Figure 21.15: Examples of inflation of a 2D stroke by Teddy (Courtesy of Takeo Igarashi, ©1999 ACM, Inc. Reprinted by permission.)

21.7.6 Grabcut and Selection by Strokes

Another example of a technology-enabled interface is Grabcut [RKB04], a system for automatically dividing an image into foreground and background portions, given a user's input—a closed curve that mostly surrounds foreground and not too much background. The system then creates a statistical model of each set (foreground and background) of pixels, based on the kinds of colors that appear in each one. From this model, one can ask, for a given pixel color, "How likely is it that this pixel was drawn from the foreground distribution? From the background distribution?" Doing this for every pixel in the image, one can find large areas that

are "likely to be background" and large areas that are "likely to be foreground," and some pixels that are ambiguous. The system then tries to find a partition of the image into foreground and background regions with two goals.

1. Pixels that are more likely to be foreground than background are generally labeled as foreground, and similarly for background.
2. Adjacent pixels tend to have the same labels.

These goals allow the system to assign a score to a partition, which in turn means that finding the best partition is an optimization problem. The optimization can be framed as an instance of the min-cut problem, for which approximation algorithms have recently been developed [BJ01]. The system finds an optimal partition, rebuilds the foreground and background models based on the new partition, and repeats the operation until the result stabilizes. (The algorithm also handles subpixel partitioning through local estimates of mixtures of foreground and background, but those details are not important here.)

The end result is that the user need only express rather general intent ("separate stuff like *this* from stuff like *that*") to accomplish a rather difficult task. (Actually drawing outlines around foreground elements in programs like Photoshop with more basic tools, or even "smart scissoring" tools, is remarkably time-consuming.)

The Grabcut approach has been improved upon with a "scribbling" interface, in which the user scribbles over some typical background regions, then changes modes and scribbles over some typical foreground regions. The scribbled-on pixels are used to create the foreground and background statistical models. In situations where making a close outline of the foreground may be difficult (e.g., a gray octopus on a coral bed), it may still be easy to mark a large group of representative pixels (e.g., by scribbling on the octopus body rather than its arms).

Grabcut has its own advantages, however: If the foreground object is one person in a crowd, the enclosing curve in Grabcut can help prevent other people with similar skin tones from being included in the foreground, as they might be with the scribbling interface.

21.8 Discussion and Further Reading

While the techniques we've discussed in this chapter provide nice illustrations of the use of linear algebra and geometry in the manipulation of objects and views, they are merely a starting point. There are other camera-manipulation approaches (such as allowing the user to control pitch, yaw, and roll about the camera itself), which, while easy to understand, can easily lead to disorientation; in a sparsely populated world—a geometric modeling system in which you're crafting one object, for instance—it's easy to rotate the camera to "look at nothing," and then have trouble refinding the object of interest. Similarly, it can be easy to zoom or dolly so far out that the object of interest is subpixel in size, or so far in that the entire view is covered by a single tiny part of the object, rather like standing with your nose against the outside of a building. Fitzmaurice et al. [FMM+08] describe a suite of tools intended to assist with "safe" navigation in a 3D CAD environment, navigation in which natural camera motions avoid the look-at-nothing and excessive-zoom problems and a host of others as well. Khan et al. [KKS+05] describe the **HoverCam,** a camera manipulator that maintains a constant distance from an object of interest (see Figure 21.16).

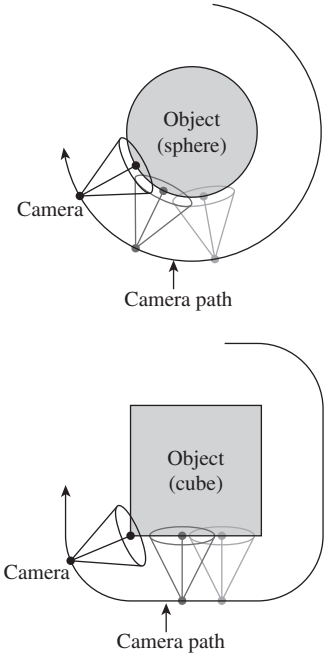


Figure 21.16: The hovercam moves the camera in a way that maintains constant distance from an object of interest. (Courtesy of Azam Khan, ©2005 ACM, Inc. Reprinted by permission.)

Glueck et al. [GCA+09] demonstrate how to improve the *output* side of interaction to make it easier for the user to understand the placement of objects in a 3D modeling system (see Figure 21.17) by showing their relationship to a ground plane (with a multiscale grid to assist in understanding size). This, in turn, is related to work of Herndon et al. [HZR+92], in which shadows of an object are projected on three walls, and the user can drag the shadow on any wall to induce a corresponding motion of the object (see Figure 21.18).

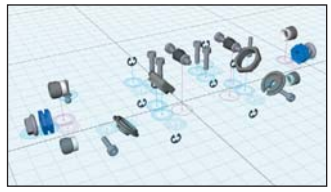


Figure 21.17: Position pegs give cues about the vertical position of objects. Transparent peg bases indicate objects below the plane. Pink pegs, like the one closest to the central grid-crossing, represent assemblies rather than individual objects. (Courtesy of Michael Glueck and Azam Khan ©2009 ACM, Inc. Reprinted by permission.)

All of these techniques show off possibilities for improved navigation and manipulation in a particular context (3D CAD and modeling); for navigation in a 3D environment (e.g., in video games) rather different approaches make sense. In CAD, for instance, you may want to be able to pass through surfaces to reach hidden surfaces on which you will then perform further operations, while in video games, it's typical to prevent players from passing through walls, for instance, and the control is often primarily 2D (forward-back and turn-left-or-right), with height above the floor determined by typical human dimensions. While generally understood camera and motion and object controls may evolve (just as some standard controls have evolved in 2D), we anticipate that application- or domain-specific controls will continue to be developed.

The form of interaction is also dependent on the device you're using: A user in a virtual reality system typically adjusts the view by moving his/her head and body, although there are many alternatives, like the World-in-Miniature approach [PBBW95], in which the VR user can hold in one hand a miniature version of the world, and move a miniature camera with the other, thus establishing a new point of view for the full-size world.

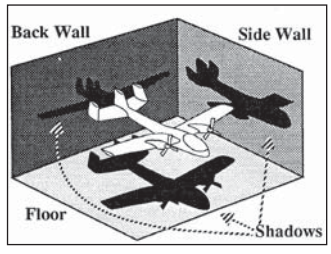


Figure 21.18: Dragging any one of the three "shadows" of the airplane makes the airplane itself move. (Courtesy of the Brown Graphics Group, ©1992 ACM, Inc. Reprinted by permission.)

In some contexts, camera control can be inferred from other aspects of the application. He et al. [HCS96] describe a "virtual cinematography" tool that uses various film idioms to automatically choose views of scenes containing multiple interacting people. For instance, in a film, when two people begin talking to each other, we typically see them both in profile; as the conversation proceeds, we typically see jump cuts between reciprocal over-the-shoulder views. Idioms like this can be used to automatically place the virtual camera in a scene with interacting people, or to assist in virtual storytelling, etc.

In general, the success of these methods can be characterized by *context integration* and *expression of intent* and the integration of expert knowledge into interfaces. For camera control, for instance, the viewer typically doesn't really want to dolly the camera. Instead, she wants to get a closer look at something; dollying the camera is a means to an end. The Unicam system provides a gesture to say, "Give me an oblique view of this object from slightly above it," for instance, and generates the camera transition to that view automatically. Similarly, the virtual cinematography system incorporates expert knowledge into the design of view transitions so that the user need not consider anything except "which person to look at." In general, there's a cognitive advantage to interfaces that let a user express *intent* rather than the *action* needed to achieve that intent.

Surprisingly often, the *technology* of interaction is closely tied to the rest of graphics. Pick correlation, for instance, is most easily implemented with a ray-scene intersection test, the very same thing we optimized for making efficient ray-casting renderers. Keeping a virtual camera from passing through walls by surrounding it with a sphere that's constrained to lie in empty space uses the underlying technology of collision detection and response to ensure that the sphere doesn't pass through any scene geometry.

If you are interested in making interfaces for video games, the best source we know is Swink's *Game Feel* [Swi08]. It discusses the problem of how to make an interface that "feels good," both analyzing successes and failures, and giving prescriptive guidelines for design.

For inspiration about user interfaces and how humans approach them, read Norman's *The Design of Everyday Things,* which takes Gibson's notion of affordances [Gib77] and applies it in the context of human-machine interaction.

For 3D spatial interaction techniques that go beyond those described in this chapter, Bowman et al. [BKLP04] give good coverage. Olsen [Ols09] discusses interactive system design that's *not* restricted to games, and is firmly hands-on, with good algorithmic and mathematical details. The classic text for those who want some grounding in user-interface design without making a career of it is by Schneiderman et al. [SPCJ09].

21.9 Exercises

Exercise 21.1: A variant of the trackball interaction works like this: The initial click is at some point P of the image plane; the mouse is currently at some point Q. The center of the object is at C, which is assumed to not be on the image plane. The vector $(Q - C) \times (P - C)$ serves as an axis for the rotation, with the amount of rotation made proportional to $\|Q - P\|$. It's nice if the proportionality constant is chosen so that for small drags, at least those that start on the line between the eyepoint and C, the rotation resembles the rotation provided by the virtual sphere interaction. The advantage is that there's no special-case handling needed if the drag goes off the transparent sphere. Implement it to see if there are any obvious disadvantages. Can you easily spin the object around the eye-to-object axis?

Exercise 21.2: The arcball has the property that a sequence of drags from A_1 to A_2 to A_3 to $\ldots$ A_n has the same net effect as a drag from A_1 to A_n. We could therefore treat each mouse-drag step as its own operation and update the controlled object's transformation at each instant, with no need to remember the `initialTransform`. Can you think of any disadvantages of this approach?

Exercise 21.3: Think about your favorite map-viewing software. Imagine that you have a route between your home and that of a friend who lives 500 miles away. You want to follow that route. Near your home, the route may involve several small streets, but soon you'll get on a major highway and remain there for some time, eventually doing some small-scale navigation again at the end as you approach your friend's home.

(a) Design an interface that lets you follow the route conveniently from end to end.

(b) Suppose that you've located your home on the map, and have marked it, and now you want to locate your friend's home and mark that so that the route-finding software can find a good route. You could enter your friend's address, but navigating visually could be faster, especially on a mobile device. You'll probably want to zoom out, find your friend's city, focus on it, zoom in, etc. Can you design a cursor-based interface to achieve this without separate steps (i.e., zoom out, then translate, then zoom in)? Hint: Consider adapting the scale of the view depending on the magnitude of the current motion.

Exercise 21.4: Adapt the photo manipulator so that if the two contacts are moved sufficiently, the photo is rotated clockwise or counterclockwise by 90° about the midpoint of the contacts. You might, for instance, check whether the

vector difference between the contacts is rotated more than 60° from the initial vector difference, and treat this as a cue to rotate. Having made this rotation, when should you rotate back to the original position? Why does a 60° threshold make more sense for starting a manipulation than a 45° one?

Exercise 21.5: Implement the translation and rotation parts of Unicam, but replace the dollying adjustment with one where each unit of vertical cursor movement multiplies your distance to the object by some constant $\rho < 1$. You'll have to decide how close you should come to the object if the user drags from the bottom to the top of the view. Compare this "logarithmic" version to the linear version of Unicam, and discuss which is preferable, and why.

Exercise 21.6: Enhance the photo-manipulation application so that the user can place two fingers on the photo, and when she moves her fingers, the photo translates and (nonuniformly) scales to maintain the contacts at the same point of the photo. Contrast this to the uniform scaling operation we described.

Exercise 21.7: Consider a basic drawing program, in which the user may draw points, lines, rectangles, ellipses, etc. How would you design the interaction with elements for a multitouch environment? In conventional drawing programs, one can resize a rectangle by dragging any corner, but with a modifier key (like CTRL) held down, the resizing is restricted to preserve the rectangle's aspect ratio. Do you think such control-limiting operations are more or less important in a multitouch context? Explain.

Exercise 21.8: The photo-sorting application has a front-to-back order on the photos: The last one loaded from the photo directory is on top. Describe some approaches to adding the ability to reorder the photos front to back in a seamless way.

Exercise 21.9: We implemented the virtual trackball by comparing the current position to the initial position and computing a rotation based on that difference. We could instead have implemented an *incremental* version, in which each cursor motion is interpreted as representing a separate tiny rotation from the prior cursor position to the current one, and these tiny rotations are accumulated. Implement this, and click on the frontmost point of the interaction sphere, then drag a small circle around that frontmost point, and finish by returning to the frontmost point. Does the cube return to its initial position? Do you personally prefer the differential or the integral version of this interaction?

Exercise 21.10: Write down, in as much detail as possible, the conceptual, functional, sequencing, and lexical design for the virtual sphere interaction.

Exercise 21.11: Write a first-person game controller. The game-play area consists of a large room populated by cylindrical poles of various radii, and you're playing "tag" with several other players, each of whom has a controller like yours, and an avatar that's a colored sphere. One player is "it," and tries to tag another player. Tagging a player happens when the avatar spheres touch. (They cannot interpenetrate, or pass through walls or poles.) When the player who is "it" tags another player, that player becomes "it" and the former "it" becomes untaggable for two seconds. Your challenge is to make an effective controller using whatever device you have: a keyboard, a mouse, a touchpad, etc. You should justify your design decisions. The game setting is loosely sketched here so that you are not too constrained: You can create the game in a small room with fat poles to make navigation difficult (because the avatar spheres barely fit between them), or in a room with no poles at all. Construct a world, and then design your controller and discuss how your controller design is influenced by the game-play world.

Chapter 22

Splines and Subdivision Curves

22.1 Introduction

In this chapter and the next, we turn very briefly to the topics of splines and subdivision, which are closely related. Both are used in **geometric modeling** (representing geometric shapes of the sort that we want to animate and render). Splines are also used in image processing, animation, data fitting, and a host of other applications. The web materials for these chapters provide a far more thorough treatment of splines. In this chapter, we provide only the briefest outline of some of the most common splines and subdivision curves; in the next, we will discuss surfaces.

22.2 Basic Polynomial Curves

We begin with two widely used ways to specify a curve. You can think of these as analogous to two ways to specify a line segment: You could specify the endpoints, P and Q, or one endpoint, P, and a vector $\mathbf{v}$ to the other endpoint (which is therefore $Q = P + \mathbf{v}$). Each form of specification has its uses, and both specify the same geometric entity. Both are instances of the Coordinate-System/Basis principle: By choosing the correct basis in which to work (in this case a basis for the vector space of cubic curves in the plane or space), we make our work simpler.

22.3 Fitting a Curve Segment between Two Curves: The Hermite Curve

Imagine that you're animating a car that's driving up the y-axis with velocity $\begin{bmatrix}0 & 3\end{bmatrix}^{\mathbf{T}}$, and arrives at the point $(0,4)$ at time $t = 0$. You need to animate its

motion as it turns and slows down so that at time $t = 1$ it's at position $(2, 5)$, with velocity $\begin{bmatrix} 2 & 0 \end{bmatrix}^{\mathbf{T}}$, as shown in Figure 22.1.

You need a way to "glue together" the two parts of the car's path to get a smooth motion. What can you do? (We're just looking for a smooth way to connect the "traveling along the y-axis" part of the path to the "traveling along the line $y = 5$" part, that is, the translational part of the car's motion. We can then, at each instant, rotate the car to align it with the tangent to our interpolating path.

First, we generalize the problem: Given positions P and Q, and velocity vectors $\mathbf{v}$ and $\mathbf{w}$, find a function $\gamma : [0, 1] \to \mathbf{R^2}$ such that $\gamma(0) = P$, $\gamma(1) = Q$, $\gamma'(0) = \mathbf{v}$, and $\gamma'(1) = \mathbf{w}$. The solution is given by

$$\gamma(t) = (2t^3 - 3t^2 + 1)P + (-2t^3 + 3t^2)Q + (t^3 - 2t^2 + t)\mathbf{v} + (t^3 - t^2)\mathbf{w} \tag{22.1}$$

$$= (1 - t)^2(2t + 1)P + t^2(-2t + 3)Q + t(t - 1)^2\mathbf{v} + t^2(t - 1)\mathbf{w}. \tag{22.2}$$

To check that $\gamma(0) = P$, we need only evaluate the four polynomials at $t = 0$; their values are $1, 0, 0$, and 0.

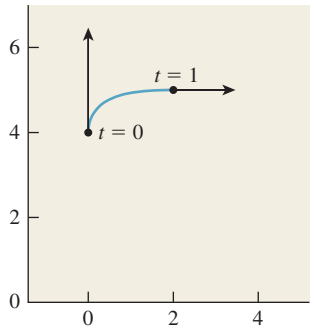


Figure 22.1: Animating a car's motion. Given the initial and final points and velocity, we want to find a path like the magenta curve.

Inline Exercise 22.1: Convince yourself that in fact the curve defined by γ satisfies $\gamma(1) = Q$, $\gamma'(0) = \mathbf{v}$, and $\gamma'(1) = \mathbf{w}$.

The resultant curve is called the **Hermite** (pronounced "airMEET") curve for the data $P, Q, \mathbf{v}$, and $\mathbf{w}$. The four polynomials in Equation 22.1 are called the **Hermite functions,** or **Hermite basis functions.**

Everything in this example works equally well if $P, Q, \mathbf{v}$, and $\mathbf{w}$ are in $\mathbf{R^3}$, or in $\mathbf{R}$: It's a dimension-independent construction. That'll be true for all our subsequent curve types, too, and we won't mention it again.

The four cubic polynomials in Equation 22.1 tell us how the inputs are combined to make the curve γ. In particular, the factors of t and $(1 - t)$ in the polynomials for $\mathbf{v}$ and $\mathbf{w}$ tell us that these inputs have no influence on the locations of the endpoints $\gamma(0)$ and $\gamma(1)$, while the factor of t^2 in the polynomial for Q shows that Q has no influence either on the location of $\gamma(0)$ *or* on the tangent vector $\gamma'(0)$ (see Exercise 22.1 at the end of this chapter). The other polynomials can be read similarly. The graphs of these four polynomials, shown in Figure 22.2, reveal the same information.

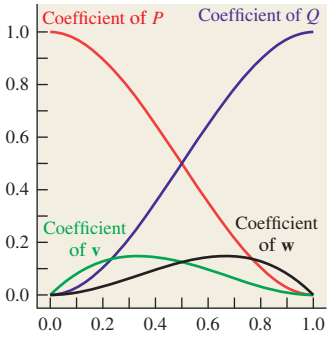


Figure 22.2: The four Hermite polynomials.

This is, as we said, an illustration of the Basis principle. In the Hermite basis, if we want to alter the starting point, we need only adjust the coefficient of the first polynomial; doing so will not alter the starting velocity, the ending point, or the ending velocity. If we had instead expressed the curve as a linear combination of the functions $t \mapsto t^3$, $t \mapsto t^2$, $t \mapsto t$, and $t \mapsto 1$, then adjusting our solution in response to a change of starting point would have altered *all* the coefficients. The so-called "power basis" consisting of powers of t is the wrong choice for this problem; the Hermite basis is the right one.

We'll generally use lowercase Greek letters (often γ) to name parametric curves, and we'll generally use t as a parameter. Sometimes, however, we'll need to relate two different curves, and in that case we will use s as well.

If the original problem had not been so nicely posed—if the original contact with the hill was at $t = a$, with velocity $\mathbf{v}$, and the straight-line motion began at time $t = b$, with velocity $\mathbf{w}$—we could still use an Hermite curve to solve it. We let $c = b - a$, and find the Hermite curve ζ for $P, Q, c\mathbf{v}$, and $c\mathbf{w}$. The gluing curve γ that we're seeking is then given by

$$\gamma(t) = \zeta\left(\frac{t-a}{b-a}\right). \tag{22.3}$$

Inline Exercise 22.2: Verify that in Equation 22.3, $\gamma(a) = P, \gamma(b) = Q, \gamma'(a) = \mathbf{v}$, and $\gamma'(b) = \mathbf{w}$.

This kind of substitution works in great generality: If we find a function of t on $[0, 1]$ with nice properties, we can transform it to a function of s on $[a, b]$ with the substitution $s = a + t(b - a)$, or $t = \frac{s-a}{b-a}$.

The Hermite basis functions are all cubic polynomials. We can, by a small change in notation, write the polynomial $a_0 + a_1t + a_2t^2 + a_3t^3$ using matrix multiplication:

$$a_0 + a_1t + a_2t^2 + a_3t^3 = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \end{bmatrix} \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix}. \tag{22.4}$$

Letting $\mathbf{t}(t)$ denote a vector containing powers of t, or $\mathbf{t}(t) = \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix}^{\mathbf{T}}$, we can write

$$\gamma(t) = [P; Q; \mathbf{v}; \mathbf{w}] \cdot \begin{bmatrix} 1 & 0 & -3 & 2 \\ 0 & 0 & 3 & -2 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \cdot \mathbf{t}(t). \tag{22.5}$$

The first factor is a matrix, called the **geometry matrix** for the curve, and is denoted **G**. Its columns are the coordinates of P, Q, $\mathbf{v}$, and $\mathbf{w}$, respectively (we'll use the semicolon notation for this in the future as well). The middle matrix, called the **basis matrix** and denoted **M**, contains the coefficients of the polynomials for the Hermite curve, from lowest to highest degree. ◈ In effect, it represents the change from the basis for cubic polynomials consisting of the four Hermite polynomials to the $\{1, t, t^2, t^3\}$ basis.

Inline Exercise 22.3: (a) Multiply out, by hand, the second and third factors in the expression for $\gamma(t)$; you should get a column vector of four polynomials. Confirm that these are the Hermite polynomials.
(b) Suppose that we had defined $\mathbf{t}$ to be the vector $\begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix}^{\mathbf{T}}$ instead; how would the second matrix in the expression for $\gamma(t)$ have to change to make the formula correct in this case?

Inline Exercise 22.4: Suppose that $\zeta(t) = (1 - t)P + tQ$. Write ζ in a matrix form like that of Equation 22.5. Your vector $\mathbf{t}(t)$ will be just $\begin{bmatrix} 1 & t \end{bmatrix}^{\mathbf{T}}$.

Thus, in brief, the Hermite curve can be written

$$\gamma(t) = \mathbf{GMT}(t). \tag{22.6}$$

All our subsequent curve formulations will have the form of Equation 22.6, namely, a geometry matrix **G** (which usually contains four points rather than two

points and two vectors), a basis matrix $\mathbf{M}$ that lists the coefficients of some polynomials, and the vector $\mathbf{T}(t)$. The differences in various curve types are (a) the contents of the geometry matrix, and (b) the polynomials specified by the basis matrix.

22.3.1 Bézier Curves

Our second curve type is the **Bézier curve.** (Bézier is pronounced "BAY-zee-ay.") It's built from four points $P_1, \ldots, P_4$. The curve starts at P_1, finishes at P_4, and has initial velocity $3(P_2 - P_1)$ and final velocity $3(P_4 - P_3)$, as shown in Figure 22.3.

The Bézier curve is given by

$$\gamma(t) = [P_1; P_2; P_3; P_4] \begin{bmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{T}(t) \tag{22.7}$$

so that this time the geometry matrix contains the four *points,* and the basis matrix contains different coefficients.

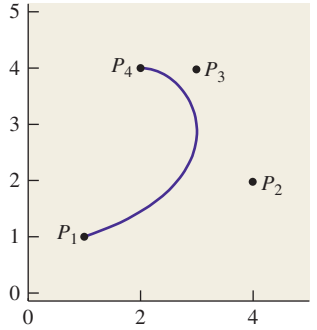


Figure 22.3: A Bézier curve starts at P_1, heading toward P_2, and ends at P_4, coming from the direction of P_3.

It may seem that the Bézier specification is less natural than the Hermite form. The role of the points P_2 and P_3 is a little vague compared to that of the initial and final tangents. The advantage of the Bézier form is that all the specified items are *points,* so when we want to transform a Bézier curve we can simply transform the points. With an Hermite curve, we have to be careful about the distinction between transforming points and vectors, which you'll recall from Chapter 12.

Inline Exercise 22.5: Suppose that P_2 and P_3 are evenly spaced between P_1 and P_4.
(a) Show that this means that $\mathbf{G}$ can be written
$\mathbf{G} = [P_1; P_4] \begin{bmatrix} 1 & \frac{2}{3} & \frac{1}{3} & 0 \\ 0 & \frac{1}{3} & \frac{2}{3} & 1 \end{bmatrix}$.
(b) Use the result of part (a) to show that in this case, $\gamma(t)$ simplifies to just $(1 - t)P_1 + tP_4$, that is, a constant-speed, straight line from P_1 to P_4. This property is one reason why the factor of 3 is included in the definition of the Bézier curve.

Exercise 22.2 shows that there's really very little difference between the two curve types.

22.4 Gluing Together Curves and the Catmull-Rom Spline

Suppose that we have a sequence of points $P_0, P_2, \ldots, P_n$ and associated vectors $\mathbf{v}_0, \mathbf{v}_2, \ldots, \mathbf{v}_n$ as shown in Figure 22.4, and we want to find a curve $\gamma : [1, n] \to \mathbf{R^2}$ that passes through these points with the given vectors as velocities. We can certainly use the Hermite formulation to find a curve $\gamma_0 : [0, 1] \to \mathbf{R^2}$ that starts at P_0, ends at P_1, and has initial and final tangents $\mathbf{v}_0$ and $\mathbf{v}_1$. We can also use it to find a curve $\gamma_1 : [0, 1] \to \mathbf{R^2}$ that starts at P_1, ends at P_2, and has $\mathbf{v}_1$ and $\mathbf{v}_2$ as its initial and final tangents, and similarly can find curves $\gamma_3, \ldots \gamma_{n-1}$. We can then define

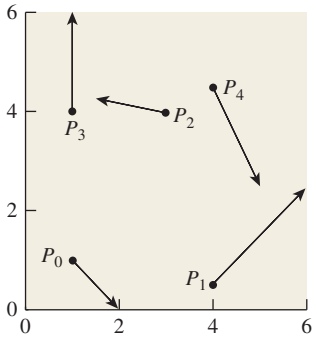


Figure 22.4: A sequence of points and vectors; we want a curve that passes through the points with the given vectors as velocities.

$$\gamma(t) = \begin{cases} \gamma_0(t) & 0 \le t \le 1 \\ \gamma_1(t-1) & 1 \le t \le 2 \\ \gamma_2(t-2) & 2 \le t \le 3 \\ \cdots & \\ \gamma_{n-1}(t-(n-1)) & n-1 \le t \le n. \end{cases} \tag{22.8}$$

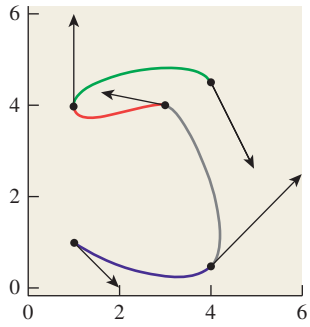


Figure 22.5: A collection of segments forming a curve that solve the problem.

The resultant assembly of curves γ (Figure 22.5) is a continuous differentiable curve that passes through each point with the specified tangent. The individual pieces, $t \mapsto \gamma_i(t-i)$, are referred to as **segments;** the whole assembly is a **spline,** and the points and vectors are what we'll call **control data:** They are the inputs that control the shape of the curve. In the most common case, we have just a sequence of points as our control data, and we call the points **control points.**

The Catmull-Rom spline is an example of a curve defined by a sequence of control points. It's the solution to the problem, "Given a sequence of points $P_0, \ldots, P_n$, find a smooth curve that passes through point i at time $t = i$, with the property that if the points are equispaced, the resultant curve is just a straight-line interpolation between the first and last points."

The idea is simple: If we can just pick a tangent at each P_i, we can use Hermite curves as before. Thus, at each control point P_i, we need to pick a tangent. The Catmull-Rom idea is to use the previous and next control points as guides, that is, to pick the tangent vector at P_i to be in the direction $P_{i+1} - P_{i-1}$ from the previous to the next control point. To satisfy the equispacing condition, we need to scale down this vector somewhat. We use the tangent vector $\mathbf{v}_i = \frac{1}{3}(P_{i+1} - P_{i-1})$, $(i = 1, \ldots, n-1)$.

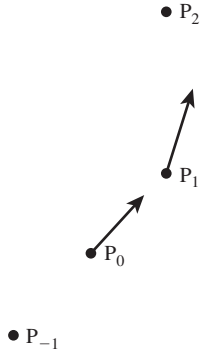


Figure 22.6: If we place a fictitious control point P_{-1} symmetric to P_1 about P_0, then we can define $\mathbf{v}_0 = \frac{1}{3}(P_1 - P_{-1})$. Notice that the tangent at P_1 is parallel to the line from P_0 to P_2.

At the endpoint P_0, this formula doesn't work, because we don't have a point P_{-1}. So, for P_0, we use $\mathbf{v}_0 = \frac{2}{3}(P_1 - P_0)$, which is what we'd get with the general formula if there were a control point P_{-1} placed symmetric to P_1 about P_0, as shown in Figure 22.6.

Similarly, for P_n we use $\mathbf{v}_n = \frac{2}{3}(P_n - P_{n-1})$. The result, after a lot of algebraic shuffling that's described in the web material for this chapter, can be written in the form

$$\gamma(t) = \sum_{i=0}^{n} P_i b_{CR}(t-i), \tag{22.9}$$

where b_{CR} is the Catmull-Rom curve shown in Figure 22.7 and defined by

$$b_{\mathrm{CR}}(t) = \begin{cases} \vdots & \\ 0 & t < -2 \\ p_4(t+2) & -2 \le t \le -1 \\ p_3(t+1) & -1 \le t \le 0 \\ p_2(t) & 0 \le t \le 1 \\ p_1(t-1) & 1 \le t \le 2 \\ 0 & 2 < t \\ \vdots & \end{cases}. \tag{22.10}$$

where

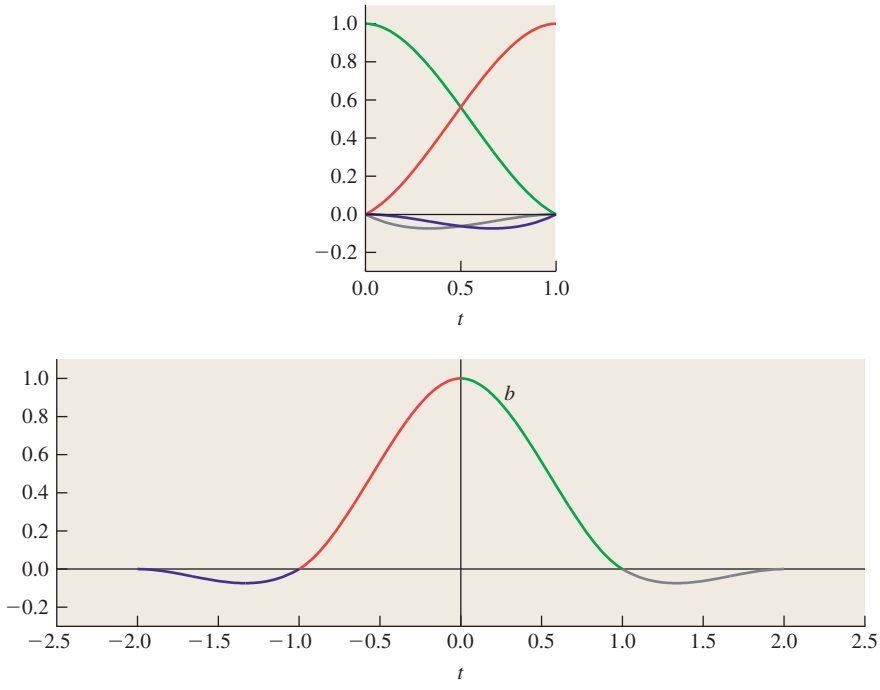


Figure 22.7: The four Catmull-Rom basis functions, plotted on a single coordinate system, and then shifted and assembled to form the function b_{CR} defined on the interval $[-2, 2]$. Because b_{CR} is continuous and is C^1 smooth, so is the Catmull-Rom spline. Because $b_{CR}(0) = 1$, while $b_{CR}(i) = 0$ for all other integers i, the Catmull-Rom spline is interpolating.

$$p_1(t) = \frac{1}{2}(-t^3 + 2t^2 - t)$$
$$p_2(t) = \frac{1}{2}(3t^3 - 5t^2 + 2)$$
$$p_3(t) = \frac{1}{2}(-3t^3 + 4t^2 + t), \text{ and}$$
$$p_4(t) = \frac{1}{2}(t^3 - t^2).$$

The form of the Catmull-Rom curve given in Equation 22.9 is convenient for studying the properties of Catmull-Rom splines. Note that, although the sum appears to have n terms, for any particular value of t there are, at most, four nonzero terms. This means that it's easy to write code to rapidly determine points on a Catmull-Rom spline. The function b_{CR} goes below 0 at some times. This means that the sum in Equation 22.9 is not a *convex* combination of the control points: The interpolating curve for control points $P_o, \ldots, P_n$ may go outside the convex hull of these points. Figure 22.8 shows a simple example. It's a sad fact that if you want a smooth **interpolating** curve (i.e., one that passes *through* the control points rather than near them), this failure to stay within the convex hull is unavoidable.

Note that the function b_{CR} is infinitely differentiable at most points (because it's polynomial), but at the joint points ($x = -2, -1, 0, 1, 2$) it's only *once*

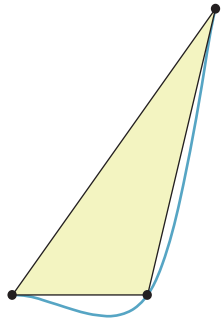


Figure 22.8: The Catmull-Rom spline for three control points lies almost entirely outside the yellow triangular convex hull of the three points.

differentiable. This is fine in some simple modeling applications, but you would not want to use a Catmull-Rom spline curve as a path for a dollied camera, for example: The resultant camera motion would seem very jerky to the viewer.

22.4.1 Generalization of Catmull-Rom Splines

We can generalize and say, "Given $P_0, \ldots, P_n$ and a sequence of parameter values $t_0 < \ldots < t_n$, find a curve γ such that $\gamma(t_i) = P_i$ for $i = 0, \ldots n$." These parameter values t_i are called **knots,** and the sequence of knots is denoted by the letter $T = t_0, t_1, \ldots, t_n$. Figure 22.9 shows a spline with knots at $t = 0, 1, 3, 4.3, 3.8, 5.5$ and control points drawn as red circles; the large blue dots are the fictitious control points. The 50 green dots are equispaced from 1 to 5.5 to show the velocity of the curve.

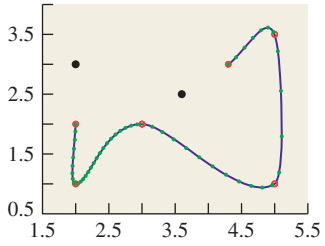


Figure 22.9: A generalized Catmull-Rom spline.

The basic Catmull-Rom spline has its knots at $t = 0, 1, 2, \ldots$; the constant inter-knot spacing leads to the name **uniform** for this kind of spline. In this section, we're describing the generalization to a **nonuniform Catmull-Rom spline.**

To solve the general problem of finding the nonuniform Catmull-Rom spline for a given sequence of knots and control points, we once again add a fictitious pre-start point $P_{-1} = P_0 - (P_1 - P_0)$, but we also add a pre-start knot, $t_{-1} = t_0 - (t_1 - t_0)$, and corresponding post-end point and knot. The ith segment of the curve is controlled by points $P_{i-1}, P_i, P_{i+1}, P_{i+2}$ and is defined for $t \in [t_i, t_{i+1}]$, but it is influenced by t_{i-1} and t_{i+2} as well. The four blending functions for this ith segment, $t \mapsto p_{i,1}(T,t), t \mapsto p_{i,2}(T,t), t \mapsto p_{i,3}(T,t)$, and $t \mapsto p_{i,4}(T,t)$, which are used to blend P_{i-1}, P_i, P_{i+1}, and P_{i+2}, are given by

$$p_{i,1}(t) = \frac{-(t-t_i)(t-t_{i+1})^2}{(t_{i+1}-t_{i-1})(t_i-t_{i+1})^2} \tag{22.11}$$

$$p_{i,2}(t) = \frac{(t-t_{i+1})^2(t_i-t_{i+1}+2(t_i-t))}{(t_i-t_{i+1})^3} - \frac{(t-t_{i+1})(t_i-t)^2)}{(t_{i+2}-t_i)(t_i-t_{i+1})^2} \tag{22.12}$$

$$p_{i,3}(t) = \frac{(t-t_i)^2(t_{i+1}-t_i+2(t_{i+1}-t))}{(t_{i+1}-t_i)^3} + \frac{(t-t_i)(t_{i+1}-t)^2)}{(t_{i+1}-t_{i-1})(t_{i+1}-t_i)^2} \tag{22.13}$$

and

$$p_{i,4}(t) = \frac{(t-t_{i+1})(t-t_i)^2}{(t_{i+2}-t_i)(t_{i+1}-t_i)^2}. \tag{22.14}$$

For the special case $t_i = i$, these agree with the Catmull-Rom basis functions given in the previous section.

To make this concrete, suppose we specify a nonuniform Catmull-Rom spline in the plane with the data

i	t_i	P_i
0	1	(1, 3)
1	1.2	(2, 3)
2	1.7	(3, 4)
3	2.5	(3, 6)

and we wish to evaluate the spline curve at many t-values so as to make a polyline that closely approximates it.

First, we add the two fictitious control points to the table:

i	t_i	P_i
-1	0.8	(0, 3)
0	1	(1, 3)
1	1.2	(2, 3)
2	1.7	(3, 4)
3	2.5	(3, 6)
4	3.3	(3, 8)

Now, to evaluate the curve at a particular t-value like $t = 2.6$, for example, we determine that $t_3 \leq 2.6 \leq t_4$. This means that we're in the ith segment, where $i = 3$. We'll therefore need to evaluate $p(3,1)(t), \ldots, p(3,4)(t)$.

$$p(3,1)(t) = \frac{-(t-t_3)(t-t_4)^2}{(t_4-t_2)(t_3-t_4)^2}, \text{ so} \tag{22.15}$$

$$p(3,1)(2.6) = \frac{-0.1 \cdot 0.8^2}{1.6 \cdot (0.8)^2}. \tag{22.16}$$

In evaluating the other three polynomials, the expressions $t - t_i$ and $t_{i+1} - t$ appear repeatedly. To write efficient code, you'd want to evaluate these just once and reuse them often.

22.4.2 Applications of Catmull-Rom Splines

Suppose that you're doing an animation in which you have a moving object, and you want it at position P_i at time t_i for $i = \ldots$; the Catmull-Rom spline is a natural choice. Now suppose you've got an object that's controlled by some parameter such as a pinwheel whose rotation is specified in degrees at several key times—say, $R(0) = 45$, $R(1) = 360$, and $R(3) = 720$—and you want to provide values for R at intermediate times. Again, the Catmull-Rom spline is a natural choice. Note, however, that if the rotations were specified by $R(0) = R(1) = 0$ and $R(3) = 90$, then the Catmull-Rom interpolant, at $t = 0.5$, would be *negative:* The pinwheel would start to spin backward before rushing to spin forward. While this might give the feeling of anticipation of motion in a cartoon-like animation, it would be inappropriate for an animation that was supposed to be physically realistic.

22.5 Cubic B-splines

Cubic B-splines (there are also linear, quadratic, quartic, etc., B-splines, but cubics are widely used) are similar to Catmull-Rom splines. The key differences are that the cubic B-spline (a) is C^2 smooth, that is, both its first and second derivatives are continuous functions, and (b) is noninterpolating, that is, it passes *near* the control points, but not *through* them in general.

Cubic B-splines come in two flavors: uniform and nonuniform. We'll start with the uniform B-spline. The formula for the cubic B-spline with control points $P_0, \ldots, P_n$ is

$$\gamma(t) = \sum_{i=0}^{n} P_i b_3(t-i), \text{ where} \tag{22.17}$$

$$b_3(t) = \begin{cases} \frac{1}{6}t^3 & 0 \le t \le 1 \\ \frac{1}{6}(-3(t-1)^3 + 3(t-1)^2 + 3(t-1) + 1) & 1 \le t \le 2 \\ \frac{1}{6}(3(t-2)^3 - 6(t-2)^2 + 4) & 2 \le t \le 3 \\ \frac{1}{6}(-(t-3)^3 + 3(t-3)^2 - 3(t-3) + 1) & 3 \le t \le 4 \\ 0 & \text{otherwise.} \end{cases} \tag{22.18}$$

The domain of the curve γ is $0 \le t \le n-2$. Because of the structure of the function b_3, it turns out that for $j \le t \le j+1$, the point $\gamma(t)$ lies in the convex hull of the four points $P_j, \ldots, P_{j+3}$. This **convex hull property** is useful in computing the intersection of a ray with a B-spline: If the ray misses the convex hull of four sequential control points, it also must miss the corresponding segment of the B-spline curve. If the ray *hits* the convex hull, then further computation is needed. The web material for this chapter gives details.

Just like the Bézier and Hermite curves, a segment of a B-spline can be expressed in a matrix form, which can make evaluation more efficient. Recall that the form for the Bézier and Hermite curves was

$$\gamma(t) = \mathbf{GMT}(t), \tag{22.19}$$

where $\mathbf{T}(t)$ is the vector $\begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix}^{\mathbf{T}}$ of powers of t. Because a B-spline curve is made of many segments, defined for $0 \le t \le 1$, $1 \le t \le 2$, etc., we'll end up using $\mathbf{T}(t-j)$ for the jth segment, that is, a vector of powers of the fractional part of t.

For the jth segment, defined for $j \le t \le j+1$, and influenced by control points $P_j, \ldots P_{j+3}$, we define a geometry matrix

$$\mathbf{G}_B = \begin{bmatrix} P_j; P_{j+1}; P_{j+2}; P_{j+3} \end{bmatrix} \tag{22.20}$$

which is 2×4 for curves in the plane, or 3×4 for curves in space, and contains the coordinates of the control points as columns of the matrix. We multiply this by the **B-spline basis matrix, $\mathbf{M}_{\text{Bs}}$**:

$$\frac{1}{6}\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 3 & 3 & -3 \\ 4 & 0 & -6 & 3 \\ 1 & -3 & 3 & -1 \end{bmatrix}. \tag{22.21}$$

The uniform B-spline curve is then

$$\gamma(t) = \mathbf{G}_B \mathbf{M}_{\text{Bs}} \mathbf{t}(t-j), \tag{22.22}$$

where $j = \lfloor t \rfloor$ so that $t-j$ is the fractional part of t.

Although B-splines don't pass through their control points, their extra degree of continuity makes them attractive in many applications. The tradeoff between controllability (does the curve interpolate its control points?) and continuity (how smooth is it?) is one that must be managed on a case-by-case basis in applications.

22.5.1 Other B-splines

While B-splines (and other cubic or piecewise-cubic curve formulations) are very popular, they do have limitations. One is that with a finite set of control points, you cannot make a B-spline curve traverse a unit circle. Since circles are important in manufacturing and many other applications, this is a severe limitation.

The solution is to include an extra coordinate, w, in your B-spline. You then take $(x(t), y(t), w(t))$ and treat it as defining $\left(\frac{x(t)}{w(t)}, \frac{y(t)}{w(t)}\right)$; the resultant curve is called a **rational B-spline,** and it happens that with a rational B-spline, you *can* traverse a circle and other conic sections.

The uniform spacing of B-splines is a convenience . . . unless you have data that happens to have nonuniform spacing (e.g., you know the position of an object in an animation at times $t = 0, 1, 2$, and 10). For this situation, there's a generalization of the B-spline called the **nonuniform B-spline,** and the rational version of this—the **nonuniform rational B-spline** or **NURB**—is one of the tools of choice in many CAD systems. One advantage of nonuniform B-splines is that by repeating knots (i.e., by having both t_3 and t_4 have the same value), you can reduce the continuity of the curve at t_3, allowing a user to put sharp corners into an otherwise smooth piecewise cubic curve, for instance. The web materials describe the uses of repeated control points and repeated knots in shaping NURBS curves.

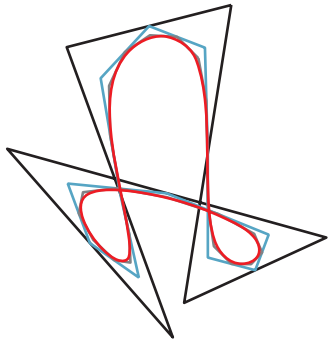


Figure 22.10: A polygon (black) subdivided three times (colors) to approach a smooth limit curve.

22.6 Subdivision Curves

As you saw in Chapter 4, repeated subdivision of a polygonal curve can lead to a smooth curve. There's one particular subdivision rule with some great properties. The new polygon is derived from the old one by doing the following:

- Using the midpoint of the edge from v_i to v_{i+1} as the vertex we'll call e_i (for "edge")
- Replacing v_i with $w_i = \frac{1}{2}v_i + \frac{1}{4}e_i + \frac{1}{4}e_{i+1}$ (which is only defined for $0 < i < n$)
- Creating the new polygon $e_1, w_1, e_2, w_2, \ldots, e_{n-1}$

Figure 22.10 shows an example of several levels of subdivision, where the rule has been extended to $i = 0$ and $i = n$ using indices modulo n. The limit curve (with this subdivision scheme) turns out to be smooth. Figure 22.11 shows an advantage of subdivision as a modeling approach: You can draw the general shape of a curve with a first polygon, subdivide a couple of times, and then move a single control point to introduce a finer-scale feature, and continue subdividing.

Even though subdivision is easy to perform, it's nice to know a parametric form for the limit curve. Figure 22.12 shows that if we take the polyline with vertices $\ldots, (-2, 0), (-1, 0), (0, 1), (1, 0), (2, 0), \ldots$ and subdivide it, the successive curves rapidly approach the B-spline curve b_3 of Equation 22.18, which we've drawn as a solid red curve at the bottom. With a good deal of linear algebra, you can show that this apparent limit is in fact exact: Subdivision is just another way of describing cubic B-spline curves.

What makes subdivision important (aside from its simplicity) is that it generalizes very nicely to surfaces, which we'll discuss in the next chapter.

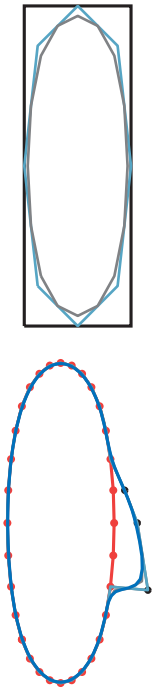


Figure 22.11: The large-scale shape of a face is drawn at the top as a black rectangle; after two levels of subdivision (shown as a red oval at the bottom), three control points (in black) are moved to the right to make a nose, and further subdivision generates a smooth curve (blue).

22.7 Discussion and Further Reading

The web material for this chapter includes a much-expanded version of the material you've just read, and includes topics such as spline paths that are "circular" (i.e., that start and end at the same point, with the same tangent vector, so you can use them for describing repeated motions). It also contains pointers to other literature on the subject.

One of the early reasons for developing splines was to approximate other functions with comparatively simple ones. This idea leads naturally to the use of splines for **compression:** If you have a sequence of many data points that lie on a fairly smooth curve, you can probably approximate that curve with a spline curve defined by just a few control points, thus generating a lossy compression of the data. This is really just a generalization of the idea of approximating data by fitting lines to them, but it's quite powerful.

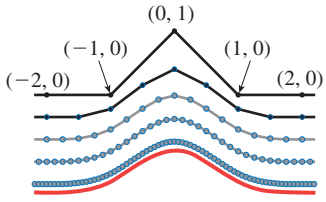


Figure 22.12: The control polygon at the top, when subdivided, approaches the graph of b_3 (in red), the cubic B-spline function. The subdivision levels are drawn vertically offset for clarity.

22.8 Exercises

Exercise 22.1: The four Hermite polynomials of Equation 22.1 control the shape of the Hermite curve.
(a) Compute the derivative of each polynomial.
(b) Evaluate the derivatives at $t = 0$ and $t = 1$.
(c) Explain why only $\mathbf{v}$ and $\mathbf{w}$ affect the direction of the Hermite curve at the start and end, while P and Q have no effect on these directions.

Exercise 22.2: An Hermite curve specification has a geometry matrix $\mathbf{G}_H = \begin{bmatrix} P, Q, \mathbf{v}, \mathbf{w} \end{bmatrix}$ containing the two endpoints and their associated tangents. A Bézier curve specification contains four points $\mathbf{G}_B = \begin{bmatrix} P_1 & P_2 & P_3 & P_4 \end{bmatrix}$. They each, too, have associated basis matrices, which we'll call $\mathbf{M}_H$ and $\mathbf{M}_B$, respectively.
(a) Show that if we pick

$$P_1 = P \qquad P_2 = P + \frac{1}{3}\mathbf{v} \qquad P_3 = Q - \frac{1}{3}\mathbf{w} \qquad P_4 = Q, \tag{22.23}$$

then the Hermite curve defined by $P, Q, \mathbf{v}$, and $\mathbf{w}$ is identical to the Bézier curve defined by $P_1, \ldots, P_4$.
(b) Show that in this situation, in matrix form, we have

$$\mathbf{G}_B = \mathbf{G}_H \mathbf{S} \tag{22.24}$$

$$= \mathbf{G}_H \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \frac{1}{3} & 0 & 0 \\ 0 & 0 & -\frac{1}{3} & 1 \end{bmatrix}. \tag{22.25}$$

(c) Using the fact that $\mathbf{G}_B \mathbf{M}_B = \mathbf{G}_H \mathbf{M}_H$, and part (b), show how to determine $\mathbf{M}_H$ from $\mathbf{M}_B$. ◈ (The equality of the two products holds because if $\mathbf{A}\mathbf{t}(t) = \mathbf{C}\mathbf{t}(t)$ for every t for two 4×4 matrices $\mathbf{A}$ and $\mathbf{C}$, then $\mathbf{A} = \mathbf{C}$; that's because the vectors $\mathbf{t}(0), \mathbf{t}(1), \mathbf{t}(2)$, and $\mathbf{t}(3)$ are linearly independent.)

Exercise 22.3: In the development of the Catmull-Rom spline, we talked about placing a fictitious control point P_{-1} that's symmetric to P_1 about P_0.
(a) Show that the point P_{-1} is given by $P_0 - (P_1 - P_0)$, and simplify.

(b) Show that if we apply the rule $\mathbf{v}_0 = \frac{1}{3}(P_1 - P_{-1})$, then the formula from part (a) lets us simplify this to $\mathbf{v}_0 = \frac{2}{3}(P_1 - P_0)$.

Exercise 22.4: In the Catmull-Rom spline, we placed a fictitious control point at each end, placing it so that the last three control points at each end were symmetrical. What would happen if we set $P_{-1} = P_0$ and $P_{n+1} = P_n$ instead? The resultant spline will still interpolate all the original control points, but thinking of the spline as describing the position of a moving point at time t, we'll see its motion change at the ends. How will it change?

Exercise 22.5: Show that the Catmull-Rom spline is in general *not* C^2. On each segment, the second derivative is a linear function (as it is for any cubic spline). Show that this function need not be continuous between segments.

Chapter 23

Splines and Subdivision Surfaces

23.1 Introduction

Both spline curves and subdivision curves can be generalized, creating spline and subdivision *surfaces*. In this chapter, we show how to make a **Bézier patch:** a small piece of surface, parameterized by $[0, 1] \times [0, 1]$, for which $u \mapsto S(u, v_0)$ is a Bézier curve for each value of v_0 and $v \mapsto S(u_0, v)$ is a Bézier curve for each value of u_0 (i.e., "it's Bézier in both directions"). Just as Bézier curves can be joined together into longer curves, Bézier *patches* can be assembled together into a "quilted" surface, although making the patches meet up smoothly at the edges and corners is more complex than in the curve case. The quilt, in this situation, generally has the form of a grid: squares meeting four at a corner. The web material for this chapter describes the creation of spline surfaces in more detail.

If we want to make a shape where adjacent patches meet three at a corner or five at a corner, the conditions for continuity are much messier and the resultant shape is not as controllable. One popular solution in this case is to shift to **subdivision surfaces,** which start from an arbitrary polyhedron and, through repeated subdivision, converge to a surface that's generally very smooth. In the subdivision scheme we present in this chapter we can start from any polygonal mesh, but after subdivision all faces of the mesh become quadrilateral, and after repeated subdivision most vertices meet exactly four faces. Faces whose vertices all have valence four can be shown to be the same as cubic spline surfaces, which meet their neighbors with C^2 smoothness. At the "exceptional" vertices, where three or five or six or more faces meet, the surface is generally C^1 smooth, but it may have curvature discontinuities. The web material for this chapter describes various further subdivision schemes and their implementation.

23.2 Bézier Patches

Just as a Bézier curve was defined by a sequence of four points P_1, P_2, P_3, and P_4, we'll describe a Bézier **patch** by a mesh of 16 points, P_{ij}, where i and j range from 1 to 4. Writing

$$b_i(t) = \binom{3}{i}(1-t)^i t^{3-i} \tag{23.1}$$

for the ith Bézier basis function, the Bézier curve based on P_{11}, P_{12}, P_{13}, and P_{14} is

$$\gamma(t) = \sum P_{1i}\, b_i(t). \tag{23.2}$$

There's nothing special about the "1" in this formula—a Bézier curve can be constructed based on any row of points in the mesh. The same goes for the columns. If we combine the two, we can build a parameterized *surface:*

$$S(u,v) = \sum_{i,j=1}^{4} b_i(u) P_{ij}\, b_j(v). \tag{23.3}$$

The function S is defined for $0 \le u, v \le 1$. If we hold v fixed—say, at $v = 0$—we get

$$S(u,0) = \sum_{i,j=1}^{4} b_i(u) P_{ij}\, b_j(0). \tag{23.4}$$

Since $b_j(0) = 0$ for $j = 2, 3, 4$, and $b_1(0) = 1$, this simplifies to

$$S(u,0) = \sum_{i=1}^{4} b_i(u) P_{i0}, \tag{23.5}$$

so $S(u,0)$ traverses a Bézier curve as u varies from 0 to 1, with control points $P_{i1}, i = 1, 2, 3, 4$. Similarly, $S(u,1)$ traverses a Bézier curve with control points P_{i4}, and $S(0,v)$ and $S(1,v)$ traverse Bézier curves with control points along the other two edges of the mesh.

In fact, for any fixed value of v—say, v_0—$S(u,v_0)$ traverses a Bézier curve: The surface defined by S is made up of a family of Bézier curves, one for each value of v_0, as Figure 23.1 shows. But the same is true in the other direction: As we hold u fixed and vary v, we also get a Bézier curve.

Steven A. Coons did some of the early work on applications of spline patches to computer-aided geometric design. He particularly studied ways to represent surfaces by multiple patches "glued together" along their boundaries in ways that were fairly easy to control. In 1967, he wrote a book on geometric design [Coo67] that profoundly influenced the development of the field.

The highest honor in computer graphics, the Steven Anson Coons Award for Outstanding Creative Contributions to Computer Graphics, is named for him.

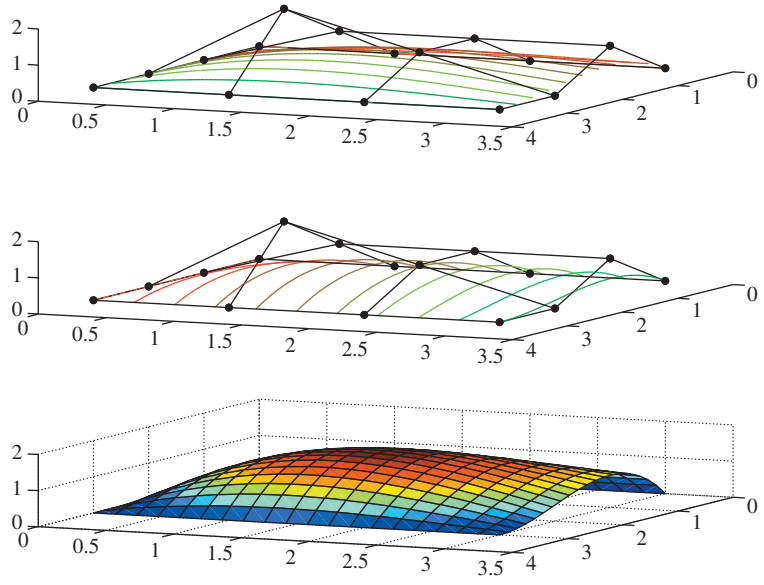


Figure 23.1: (Top) A Bézier patch drawn with the collection of curves $t \mapsto S(s,t)$ for several values of s between 0 and 1. (Middle) The same patch, drawn with the curves $s \mapsto S(s,t)$ for several values of t. (Bottom) The same surface, drawn colored by height. In the top two drawings, the control mesh $Q_{ij}(0 \leq i,j \leq 3)$ is shown.

The shape of the surface patch we've just described is controlled by the locations of the control points P_{ij}. The surface passes through the four corner points P_{11}, P_{14}, P_{44}, and P_{41}. The points on the interior of each edge, like P_{21} and P_{31}, control the shapes of the edges of the patch. For instance, the tangent plane to the patch at P_{11} contains both the vector $P_{21} - P_{11}$ and the vector $P_{12} - P_{11}$, so the cross product of these two vectors is the surface normal at P_{11}. The four interior control points determine the shape of the center region of the patch without influencing the patch boundary. They do, however, affect the direction in which the patch meets its boundary. If you plan to work with patches like this, you should write a small interactive application in which you can manipulate each control point to see its effect on the surface shape.

The surface we've just described is called a **bicubic tensor product patch,** because it's made by using products of basis functions, each of which is a cubic. If, in the expression $b_i(u)P_{ij}b_j(v)$ of Equation 23.3, we replaced $b_i(u)$ with $c_i(u)$, where c_i is the ith basis function for the Hermite curve, or the ith Catmull-Rom basis, or the ith cubic B-spline basis, we'd get different kinds of tensor product patches: The effects of the control points on the eventual shape would depend on the kinds of basis functions used. You could make a patch that used Bézier curves in one direction and Hermite in the other, for instance.

Just as we glued together curve segments to get longer curves, we can do similar things to get larger surfaces. We can try to place two surface patches next to each other so that they match up along a single edge. In the case of the Bézier-based patches described above, the rightmost column of control points for one patch must match the leftmost column of control points for the other, for instance. This will guarantee that the surfaces join up (their joining edges consist of a single

Bézier curve), but not that they join up smoothly. For a smooth join, without a crease along the joining curve, further conditions on the adjacent two columns of control points are needed.

Arranging a gridlike "quilt" of patches requires that a substantial collection of constraints be met; the web material for this chapter describes some of these. But when we try to make rectangular patches glue together in a pattern that has them meet three at a vertex, as in Figure 23.2, the constraints become overwhelming. There are several solutions: We can deal with the overwhelming constraints and continue to use rectangular patches, or we can shift to something like triangular patches, where gluing together is a little easier, or we can, as we did with curves, move to subdivision as a way to create shapes. We'll now briefly discuss this third approach.

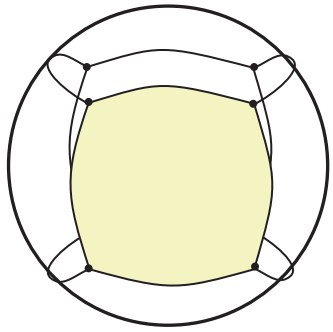


Figure 23.2: A spherical blob made from six "rectangular" patches that meet three at a vertex.

23.3 Catmull-Clark Subdivision Surfaces

Subdivision surfaces don't start with individual patches to be joined: They start from a polygonal mesh, which is repeatedly modified to approach a usually smooth limit surface. This simplifies matters a good deal.

In the Catmull-Clark subdivision scheme [CC98, HKD93], we start with a mesh, typically in $\mathbf{R}^3$ (although the process works in any dimension). The vertices of each face need not actually be coplanar, although it's easiest to visualize the subdivision process if they're nearly coplanar, so we'll start with an example where this is true (see Figure 23.3). The faces of the initial mesh may be triangles, quads, pentagons, etc., but after one level of subdivision all faces will be quads, so we've drawn an example where they are all quads.

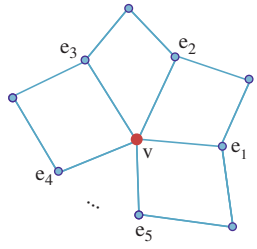


Figure 23.3: A mesh where one vertex **v** *has n adjacent vertices* $\mathbf{e}_1$, $\mathbf{e}_2$, $\ldots$ $\mathbf{e}_n$, *at the ends of the edges, leaving* **v**.

Just as with subdivision curves, we'll describe subdivision surfaces in terms of a **neighborhood** of a vertex **v**, that is, a set of vertices near **v** in the graph structure of the mesh.

The first step of subdivision is to compute the centroid $\mathbf{f}'_i$ (the average of the vertices of the ith face). (We'll follow the convention that primes denote points of the subdivided mesh, and unprimed symbols denote points of the mesh before subdivision.)

We next compute the edge points $\mathbf{e}'_i$ by the formula

$$\mathbf{e}'_i = \frac{\mathbf{v} + \mathbf{e}_i + \mathbf{f}'_{i-1} + \mathbf{f}'_{i+1}}{4}. \tag{23.6}$$

All subscripts are taken modulo n.

Finally, we compute a new location for the vertex **v**:

$$\mathbf{v}' = \frac{n-2}{n}\mathbf{v} + \frac{1}{n^2}\sum_i \mathbf{e}_i + \frac{1}{n^2}\sum_i \mathbf{f}'_i. \tag{23.7}$$

These new locations are connected as shown in Figure 23.4.

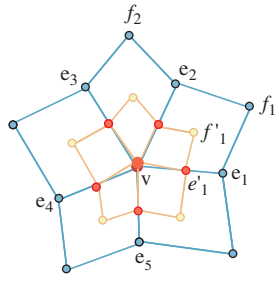


Figure 23.4: The new vertex is connected to each new edge point; the new edge points are connected to the face points for adjacent faces.

After subdivision, there are approximately four times as many faces as before subdivision. After just a few levels of subdivision, we'll have a great many faces.

Inline Exercise 23.1: Convince yourself that after one level of subdivision any mesh becomes a quad mesh, and that each newly introduced edge vertex $\mathbf{e}_i'$ has degree four. Then show that in further subdivision, each newly introduced face vertex also has degree four.

The special case $n = 4$ (which is the most common, as shown by the preceding exercise) is worth examining.

In this case, Equation 23.7 becomes

$$\mathbf{v}' = \frac{1}{2}\mathbf{v} + \frac{1}{4}\frac{\sum_i \mathbf{e}_i}{4} + \frac{1}{4}\frac{\sum \mathbf{f}_i'}{4}, \tag{23.8}$$

which says that $\mathbf{v}'$ is a weighted average of $\mathbf{v}$, the average of the adjacent edge points, and the average of the adjacent face points, just as for curve subdivision the new vertex location was an average of the old vertex location and the average of the adjacent edge points.

This situation (after the first level of subdivision) is shown in Figure 23.5.

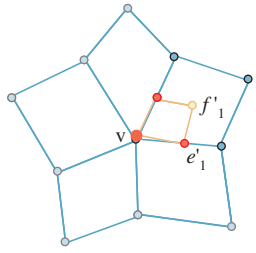


Figure 23.5: For a quad mesh, the face vertices from the previous level of subdivision are opposite $\mathbf{v}$ *in each quad.*

In this case, we can rewrite the subdivision formula for $\mathbf{v}'$ in terms of $\mathbf{v}$, $\{\mathbf{e}_i\}$, and $\{\mathbf{f}_i\}$ instead of using $\{\mathbf{f}_i'\}$; all we need to do is substitute

$$\mathbf{f}_i' = \frac{\mathbf{v} + \mathbf{e}_i + \mathbf{e}_{i+1} + \mathbf{f}_i}{4} \tag{23.9}$$

to get

$$\mathbf{v}' = \frac{1}{2}\mathbf{v} + \frac{1}{4}\frac{\sum_i \mathbf{e}_i}{4} + \frac{1}{4}\frac{\sum \mathbf{f}_i'}{4} \tag{23.10}$$

$$= \frac{1}{2}\mathbf{v} + \frac{1}{4}\frac{\sum_i \mathbf{e}_i}{4} + \frac{1}{4}\frac{\sum (\mathbf{v} + \mathbf{e}_i + \mathbf{e}_{i+1} + \mathbf{f}_i)/4}{4} \tag{23.11}$$

$$= \frac{1}{2}\mathbf{v} + \frac{1}{4}\frac{\sum_i \mathbf{e}_i}{4} + \frac{1}{4}\left[\frac{\mathbf{v}}{4} + \frac{\sum_i \mathbf{e}_i}{8} + \frac{\sum_i \mathbf{f}_i}{16}\right] \tag{23.12}$$

$$= \frac{9}{16}\mathbf{v} + \frac{3}{32}\sum_i \mathbf{e}_i + \frac{1}{64}\sum \mathbf{f}_i. \tag{23.13}$$

Corresponding formulas for $\mathbf{e}_i'$ and $\mathbf{f}_i'$ in terms of the pre-subdivision vertices are

$$\mathbf{e}_i' = \frac{1}{16}\left[6\mathbf{v} + 6\mathbf{e}_i + \mathbf{e}_{i-1} + \mathbf{e}_{i+1} + \mathbf{f}_{i-1} + \mathbf{f}_i\right] \text{ and} \tag{23.14}$$

$$\mathbf{f}_i' = \frac{1}{4}\left[\mathbf{v} + \mathbf{e}_i + \mathbf{e}_{i+1} + \mathbf{f}_i\right]. \tag{23.15}$$

If we list the coordinates of the central vertex $\mathbf{v}$, then the edge vertices $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \mathbf{e}_4$, and then the face vertices $\mathbf{f}_1, \ldots, \mathbf{f}_4$ in a 9×3 matrix, $\mathbf{V}$, we can summarize by saying that

$$\mathbf{V}' = \frac{1}{16}\begin{bmatrix} 9 & \frac{3}{2} & \frac{3}{2} & \frac{3}{2} & \frac{3}{2} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ 6 & 6 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 6 & 1 & 6 & 1 & 0 & 1 & 1 & 0 & 0 \\ 6 & 0 & 1 & 6 & 1 & 0 & 1 & 1 & 0 \\ 6 & 1 & 0 & 1 & 6 & 0 & 0 & 1 & 1 \\ 4 & 4 & 4 & 0 & 0 & 4 & 0 & 0 & 0 \\ 4 & 0 & 4 & 4 & 0 & 0 & 4 & 0 & 0 \\ 4 & 0 & 0 & 4 & 4 & 0 & 0 & 4 & 0 \\ 4 & 4 & 0 & 0 & 4 & 0 & 0 & 0 & 4 \end{bmatrix} \mathbf{V}. \tag{23.16}$$

More generally, for a vertex of degree n, there are $2n+1$ rows in $\mathbf{V}$ and the subdivision matrix is a $(2n+1)\times(2n+1)$ matrix. Letting $\mathbf{V}_n$ denote the neighborhood coordinates for a vertex of degree n, we can write

$$\mathbf{V}'_n = \mathbf{S}_n \mathbf{V}_n, \tag{23.17}$$

where the matrix $\mathbf{S}_n$ is of the appropriate size. Everything about Catmull-Clark subdivision can be determined by studying the matrix $\mathbf{S}$. Halstead et al. [HKD93] carry this out in some detail.

As an example of what one can derive, if we look at some vertex $\mathbf{v}$ in a mesh, during repeated subdivision $\mathbf{v}$ will approach some point $\mathbf{v}^\infty$. (This can be proved by looking at powers of $\mathbf{S}$; the web materials for this chapter do so explicitly.) The limit point point $\mathbf{v}^\infty$ for $\mathbf{v}$ under repeated subdivision is

$$\mathbf{v}^\infty = \frac{n^2\mathbf{v}' + 4\sum_j \mathbf{e}'_j + \sum_j \mathbf{f}'_1}{n(n+5)}, \tag{23.18}$$

where the primes indicate one level of subdivision. (This is determined by looking at eigenvectors of $\mathbf{S}$.) Notice that $\mathbf{v}$ need not be a vertex of the original mesh. If, after three levels of subdivision, we insert a new face point, we can call this point $\mathbf{v}$, find its neighboring face and edge points, and apply the limit formula above.

We can now make three observations.

First, the formula for computing the limit point does exactly the same computation with the x-, y-, and z-coordinates. In fact, it works in any dimension at all, and we can think about subdivision surfaces one coordinate at a time.

Second, if the initial mesh consists of the integer lattice in the xy-plane (i.e., all integer points are vertices, and the unit-length vertical and horizontal segments joining them are the edges), then a single subdivision operation produces the half-integer lattice, two subdivisions produce the quarter-integer lattice etc.

Third, if we alter the integer lattice by replacing $(0,0,0)$ with $(0,0,1)$, the limit surface shown in Figure 23.6 is very simple. In fact, just as for subdivision *curves,* it turns out that this limit surface is actually identical to the cubic B-spline basis function. In particular, if (x,y,z) is on the limit surface, then $B(x,y) = z$, where B is the basis function.

This means that, at least in areas where the mesh has standard lattice connectivity, the limit subdivision surface is the same as the B-spline surface defined by those control points. Since code for computing and rendering such surfaces is widely available, this is very convenient. In any neighborhood of an exceptional vertex (i.e., one whose degree is different from four), the limit surface is *not* similar to a B-spline, and we have to compute the surface by repeated subdivision. Stam [Sta98] discusses the limit shape both at and away from exceptional points.

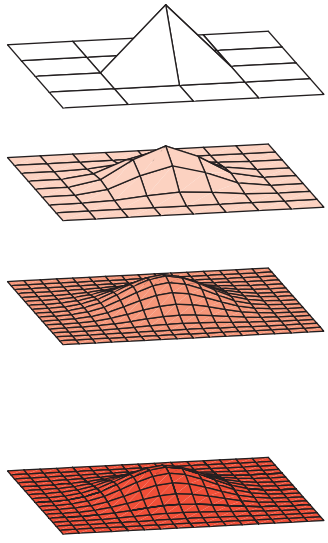


Figure 23.6: (Top) The initial mesh consists of the integer lattice with one point raised. (Middle) The first and second levels of subdivision. (Bottom) The cubic B-spline basis function whose control points are the vertices of the original control mesh, for comparison.

Analysis like that for the limit points also shows how to compute the normal vector at the limit point [HKD93], at least for vertices in sufficiently general position. (There are configurations for which the limit surface does not have a well-defined normal, just as there are B-spline curves that have geometric cusps, despite being parametrically smooth.)

Since both the limiting position and the surface normal are expressible as linear combinations of a few neighboring vertices, one can solve problems like "Find me initial vertex positions for this mesh with the property that the limit surface passes through *these* points and has *these* normal vectors at them." This is the central problem addressed by Halstead et al.

While the limit surface for subdivision is generally smooth, at exceptional vertices it may not be curvature-continuous (i.e., the curvature at the limit of an exceptional vertex may be undefined, or it may not be continuous in a neighborhood of that vertex). One might try to address the surface to be curvature-continuous through a kind of "surgery": Remove a small disk around the offending point and replace it with a smooth surface. But guaranteeing continuity at the seam is problematic. As an alternative, one can *blend* between the limit surface and a smooth replacement, using a blending function that's a sufficiently smooth function of the distance from the exceptional point and that varies from 0 at the seam to 1 at the center, with all derivatives being 0 both at the seam and at the center. Such an ad hoc approach requires choosing a radius for the disk and choosing a smooth shape to blend with, but it can be made to work quite well in practice [Lev06]. A different approach, based on improving the *appearance* of the exceptional point rather than its geometry, is to use the "approximate Catmull-Clark subdivision surfaces," or ACC surfaces, developed by Loop and Schaeffer [LS08]. Starting from the mesh to be subdivided, they develop, for every quad, a bicubic patch $(u, v) \mapsto S(u, v)$ that represents the geometry of the limit surface, and two associated patches $\mathbf{t}_u(u, v)$ and $\mathbf{t}_v(u, v)$ where the value of $\mathbf{t}_u(u, v)$ approximates the tangent vector to S in the u direction at (u, v), that is, $\frac{\partial S(u,v)}{\partial u}$, and similarly for $\mathbf{t}_v$. These approximate tangent-vector functions can be used to compute an approximate normal that varies smoothly over the surface, that is, that gives the limit surface the *appearance* of smoothness when this approximate normal vector is used in rendering it.

23.4 Modeling with Subdivision Surfaces

We've indicated, in our discussion of Catmull-Clark surfaces, how subdivision surfaces can be fitted to point and normal data. But when you have a shape in mind and want to create a model, point and normal data is not available. One approach is to make a physical model of your shape, scan it, and then fit a surface to it; this approach is used by many production studios. But as an alternative, one can model directly with subdivision surfaces.

A typical modeling session starts with a coarse mesh that the user adapts to have the general shape of the object of interest. The user then subdivides this mesh and adjusts the locations of some of the resultant vertices. Often the subdivision process puts the new mesh just about where the user expects, and new vertices need adjustment only to add detail. After another level of subdivision, the user adds more detail, etc. At some point, the surface shape is satisfactory and the limit

surface is computed (or perhaps a few more stages of subdivision are performed to generate an effectively smooth mesh).

Adjusting vertex positions after a level of subdivision is acceptable because the resultant mesh is an acceptable input to the subdivision algorithm. Indeed, one can go further: One can actually edit the topology of a mesh, adding a hole at some level, etc. The data that needs to be recorded for such a modeling session consists of the original vertex positions, plus any edits made at each level. In the event that several vertices at level three, say, were all moved in the same direction, there may be a level-two edit that would have achieved the same effect, or most of it. Rewriting the representation to include this level-two edit, and then smaller level-three edits, may make the representation more compact.

This editing approach has the advantage that the user can "browse" through different levels, adjusting the shape at higher levels and then returning to lower levels. One problem that arises is that details added at a lower level may not make sense after a high-level edit. In a face model, for instance, a nose might be drawn out in the x-direction. If at a higher level, the face is rotated by 90° in the xy-plane, the low-level edit will make a nose that's dragged to the side of the face rather than in front of it. It therefore is useful to express low-level edits in a coordinate system that's tied to the result of higher-level subdivision so that the nose is described as being drawn out along the normal to the face, rather than "in the x-direction." Such multiscale editing is described by Cohen et al. [MCCH99], and the condensation of multiscale editing representations, together with other multiscale editing techniques, is described by Zorin et al. [ZSS97].

23.5 Discussion and Further Reading

As with the preceding chapter, the web material for this chapter contains a much-expanded version of the material presented here, together with pointers to the literature. Spline and subdivision surfaces are at the heart of most of today's CAD packages, and CAD long ago became its own area, largely separate from computer graphics. Introductory CAD texts will help you grasp the main ideas (and some of the sometimes-complex indexing schemes!). Loop's Master's thesis [Loo87] is a gentle introduction. Despite the separation of graphics and CAD, there continues to be cross-fertilization.

Chapter 24

Implicit Representations of Shape

24.1 Introduction

We introduced implicit functions in Chapters 7 and 14 as a means for defining shapes. Implicitly defined shapes, like the circle defined by $x^2 + y^2 = 1$, or the sphere defined by $x^2+y^2+z^2 = 1$, or far more general shapes defined by equations of the form $F(P) = c$ for some complicated function F, serve several roles in graphics. First, for a wide class of functions, computing ray-surface intersections with such shapes is fairly easy. Second, it's sometimes convenient to represent surfaces like "the boundary between water and air" in a simulation implicitly, because it's very easy to change the topology of an implicitly defined surface (by changing either F or c), while it's generally difficult to do so for parametrically defined surfaces. Third, in many applications we find ourselves with data defined on a grid of points (the temperature at each point in a nuclear reactor, for instance, or the material density at each point in a CAT scan of a brain) and we wish to visualize this data; often, seeing the level surface (the set of points where the function has a particular value) for a function that's consistent with the observed data can help us understand the data. In this chapter, we introduce implicit curves and surfaces and discuss how they are used to model shapes, how they can be used in ray tracing and animation, and how they can be converted to polyhedral meshes.

The main advantages of implicit representations are the general smoothness of the shapes defined this way, the simplicity of creating quite general shapes, the ease of defining shapes whose topology changes over time, and the ability to exactly compute surface normals and other geometric properties (many of which are difficult to estimate for polyhedral surfaces). The disadvantages are that converting an implicit representation to a polygon mesh suitable for most renderers can be very expensive, and that the ability of implicits to represent multiple topologies can also make it difficult to *control* the topology of an implicitly defined shape.

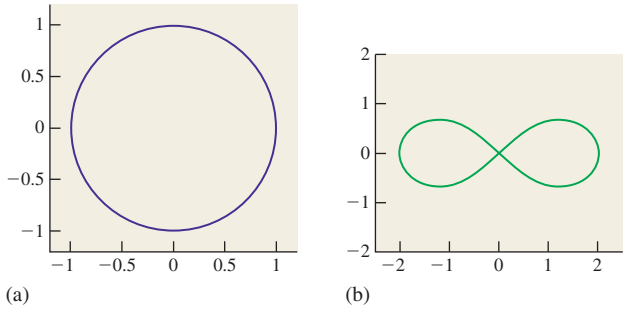


*Figure 24.1: Two implicit curves in the plane. (a) The circle is defined by $x^2 + y^2 = 1$, or by $x^2 + y^2 - 1 = 0$. (b) The **lemniscate of Bernoulli** is defined by $(x^2 + y^2)^2 = 2c(x^2 - y^2)$; changing c adjusts the angle at which the lines cross at the center of the figure eight.*

24.2 Implicit Curves

In Chapter 7 we discussed two ways to describe a line in the plane: either parametrically (writing $P + t\mathbf{d}$, for values $t \in \mathbf{R}$) or implicitly (in a form like $Ax + By + C = 0$, with A and B not both zero; or in vector form, $(X - P) \cdot \mathbf{n} = 0$, where $\mathbf{n}$ is a nonzero vector in the plane and P is a point of the line—the set of points X satisfying this equation forms a line containing P and perpendicular to $\mathbf{n}$). In addition, we observed that it was particularly easy to find the intersection of a parametric line with an implicit line. Similarly, in 3-space we could define a plane implicitly, and ray-plane intersections were easiest when the ray was parametric and the plane implicit.

We further generalized to talk about implicitly defined curves in the plane that were more general than lines, like the circle, defined by $x^2 + y^2 = 1$, or more general curves (see Figure 24.1). More generally, if we have any function $z = F(x, y)$ defined on the plane,[1] such as the height of the terrain at each location (x, y) in some hilly area, then the sets of points defined by $F(x, y) = c$ are called **contour lines** and are an example of a **level set** of a function (in the sense that they are the points at the same level on the graph of F). Level sets are sometimes also called **isocurves** of F. Mathematics books often discuss level sets by only considering the case $c = 0$; that's because the level set where $F(x, y) = c$ is the level set where $G(x, y) = 0$ if we define the function G by

$$G(x, y) = F(x, y) - c. \tag{24.1}$$

Thus, you'll sometimes encounter the term **zero set** rather than "level set."

Because the points of the curve are defined indirectly—we simply have the function F which tells us whether a point is on the curve or not—we also say that the curve is an **implicit curve.** If the formula for F is sufficiently complicated, it may not even be clear whether the set defined by $F(x, y) = c$ is empty or not.

1. We are following the mathematics convention that the xy-plane is horizontal and that the z-direction is vertical, because the xy-plane is of primary interest to us for the time being; were we to follow the graphics convention, we'd have to describe a circle by an equation like $x^2 + z^2 = 1$ instead of $x^2 + y^2 = 1$; the familiarity of the xy-formulation seems worth the inconsistency in the choice of axes.

In the cases we've discussed so far—the line, circle, and lemniscate—the first two implicit curves are very smooth, but the third has a self-crossing. The distinction among them is the nature of the functions defining them. In general, if C is the level set $F(x,y) = c$, then C consists of disjoint simple closed curves *if* at every point P of C, the gradient $\nabla F(P)$ is nonzero.

In the case of the line, the function $F_L(x,y) = Ax + By + C$ has gradient $\nabla F_L(x,y) = \begin{bmatrix} A \\ B \end{bmatrix}$, which is nonzero everywhere. For the circle, the function $F_C(x,y) = x^2 + y^2$ has gradient $\begin{bmatrix} 2x \\ 2y \end{bmatrix}$, which is zero only at $(x,y) = (0,0)$, which is not a point of the circle. But for the lemniscate,[2] where

$$F_B(x,y) = (x^2 + y^2)^2 - 2c(x^2 - y^2), \tag{24.2}$$

we have

$$\nabla F_B(x,y) = \begin{bmatrix} 4x(x^2 + y^2) - 4cx \\ 4y(x^2 + y^2) + 2cy \end{bmatrix}, \tag{24.3}$$

which, at $(x,y) = (0,0)$, is the zero vector. At places where the gradient is zero, an implicit curve can have singularities (self-intersections, sharp corners, tangencies). This is not, however, an if-and-only-if condition. For instance, the circle can also be defined by the equation

$$F(x,y) = ((x^2 + y^2) - 1)^2 = 0, \tag{24.4}$$

which has gradient zero at every point of the circle. In short, a nonzero gradient ensures that the curve is nice, but the curve's niceness tells us nothing about the gradient.

The preceding example also shows that the function that defines an implicit curve is by no means unique: Many functions can define the same curve. That's another drawback of implicits.

How common are zeroes in the gradient? A back-of-the-envelope argument says they're fairly common. If we set the first term of the gradient to zero, we've got one equation in two variables (which defines a curve in the plane); if we set the second to zero as well (defining a second curve in the plane), we've got two equations in two variables. If they were *linear* equations, we'd generally have a solution; because they may be nonlinear, we can merely say that we might well expect to find isolated solutions to the two equations (i.e., points where the two curves intersect). If we chose a level c at random, we would not expect $F(x,y) = c$ to hit any of these gradient zeroes, but if we were to vary c, we might well expect that for certain values of c, the level set for c contains a gradient zero. This can be thought of in terms of a physical analogy, as shown in Figure 24.2: If we take our function to be the height of the terrain above or below sea level, then when the sea level is c, the level set for c is the shoreline. As the tide rises, c changes, and the shape of the shoreline changes. For example, two adjacent islands may be separated by water at high tide (so that the level set consists of two closed curves—the shorelines of each island); as the tide drops, the islands may become joined by an isthmus so that at low tide, the shoreline is one long curve. For some

2. The subscript "B" is for Bernoulli.

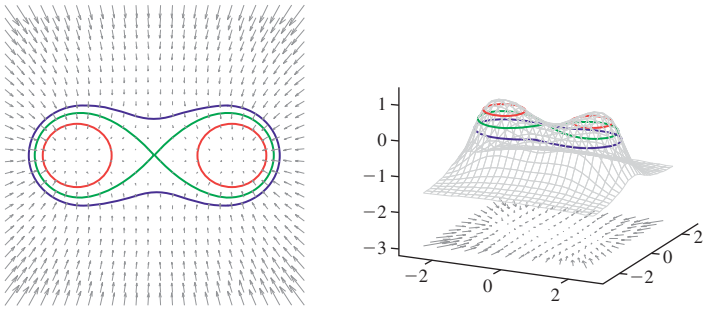


Figure 24.2: A topographical map and side view of two islands. At high tide (two almost-circular red curves) the islands are separated and the shoreline has two parts; at low tide (large blue curve) the islands are joined by an isthmus, and the shoreline is a single curve. At mid-tide (green figure-eight curve), the shoreline has a singular point, where the gradient of the height function is zero. In both figures, the faint gray arrows are scaled versions of the gradient of the implicit function.

value of c, the level set changes from two curves to one curve; at the point where the curves join, the gradient is zero.

The gradient of an implicit curve, when nonzero, has another important function: It always points in the direction of the normal vector to the curve. (This, and the claim that when the gradient is nonzero the curve is smooth, are consequences of the implicit function theorem [Spi65].) We can see this in the case of the unit circle: At the point (x, y), the gradient is $\begin{bmatrix} 2x \\ 2y \end{bmatrix}$, which is indeed parallel to the normal, which is $\begin{bmatrix} x \\ y \end{bmatrix}$.

We can also see the kinds of problems that arise when the gradient is zero by looking at the function $F(x, y) = 1 + x^3 - y^2$, shown in Figure 24.3 and the level

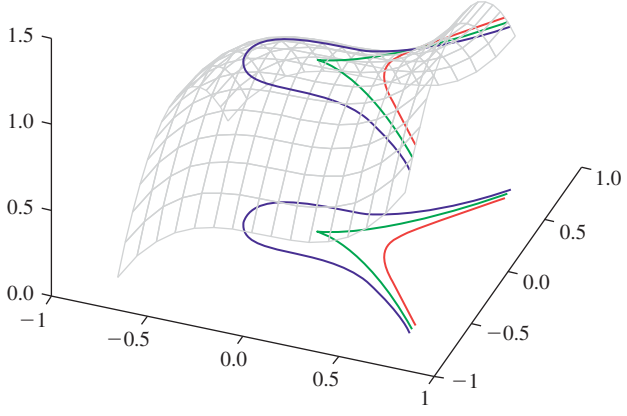


Figure 24.3: The graph of $F(x, y) = 1 + x^3 - y^2$ and its associated level curves for $c = .95, 1$, and 1.05. The level curve for $c = 1$ has a cusp at the origin, where the gradient is zero. This example shows that the topology of the level set need not change at a place where the gradient is zero.

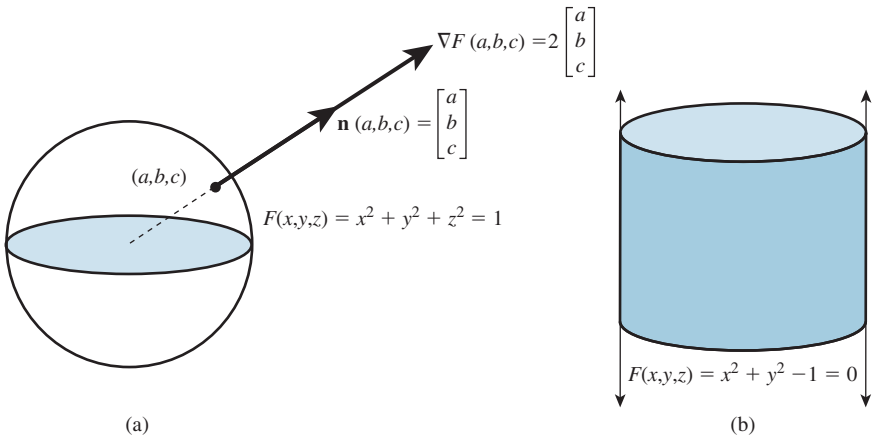


Figure 24.4: (a) The sphere is defined as the zero set of the implicit function $F(x, y, z) = x^2 + y^2 + z^2 - 1$; at a typical point $P = (x, y, z)$ of the sphere, the gradient is parallel to the ray from the origin to P, hence parallel to the normal vector to the sphere at P. (b) The cylinder can be defined implicitly by $x^2 + y^2 = 1$.

set defined by $F(x, y) = 1$: At the point $(x, y) = (0, 0)$, the level set has a sharp cusp, even though nearby level sets are completely smooth.

24.3 Implicit Surfaces

The notions of the preceding section all generalize to three dimensions quite simply: If we have a function $w = F(x, y, z)$ defined on 3-space (e.g., the temperature at each point in a room), we can find the set of points

$$\{(x, y, z) \ : \ F(x, y, z) = c\} \tag{24.5}$$

at which F takes on the value c; in general, this is a smooth surface in 3-space. As a concrete example, if F is the function defined by

$$F(x, y, z) = x^2 + y^2 + z^2, \tag{24.6}$$

then the level set for $c = 1$ is the unit sphere in 3-space, as shown in Figure 24.4. (In three dimensions, level sets are sometimes called **isosurfaces** or **level surfaces.**)

Just as in the two-dimensional case, if $P = (x, y, z)$ is a point of some level surface, then the gradient $\nabla F(x, y, z)$ is parallel to the normal vector to the surface at P. And if the gradient is nonzero everywhere, then the surface is actually smooth. On the other hand, if the gradient is zero at some point of a level surface, there may be a self-intersection there, or a corner of the surface, or a sharp point.

Again, as in the curve case, a randomly chosen level surface of a smooth function F is unlikely to contain any gradient zeroes, but if we continuously vary the level (or the function F), we should expect to encounter some gradient zeroes.

Finally, the intersection of a ray defined by a point P and a direction $\mathbf{d}$ with an implicit surface defined by $F = c$ can be computed by solving $F(P + t\mathbf{d}) = c$ (see Figure 24.5).

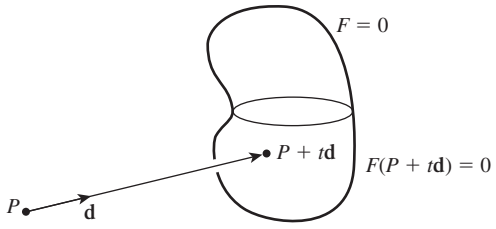


Figure 24.5: The intersection of a ray (defined by a point P and a direction $\mathbf{d}$) and an implicit surface defined by $F(x, y, z) = c$ must occur at a point $Q = P + t\mathbf{d}$ (for some value of t) which satisfies $F(Q) = 0$. So to find the intersection, we can solve $F(P + t\mathbf{d}) = c$ for the unknown t; the intersection point is then $P + t\mathbf{d}$.

For instance, if $F(x, y, z) = x^2 + y^2 + z^2$, and we consider the intersection of the ray with $P = (-2, 0, 0)$ and $\mathbf{d} = (1, 1/3, 0)$ with the level set $F = 1$ (the unit sphere), we must solve

$$F(P + t\mathbf{d}) = 1, \tag{24.7}$$

that is,

$$F(-2 + t, t/3, 0) = 1. \tag{24.8}$$

Applying the formula for F, this gives

$$(-2 + t)^2 + (t/3)^2 + 0^2 = 1, \tag{24.9}$$

which is a quadratic in t, namely,

$$10t^2 - 36t + 27 = 0, \tag{24.10}$$

whose solutions are

$$t = \frac{36 \pm \sqrt{36^2 - 4 \cdot 10 \cdot 27}}{2 \cdot 10} \approx 1.065, 2.535; \tag{24.11}$$

these correspond to the points

$$Q_1 \approx (-0.935, 0.355, 0) \text{ and } Q_2 \approx (0.535, 0.850, 0) \tag{24.12}$$

on the sphere.

Inline Exercise 24.1: The intersections we just computed depended on the coordinates (P_x, P_y, P_z) of P and the coordinates (d_x, d_y, d_z) of $\mathbf{d}$. Express the intersection points in terms of these coordinates rather than their particular values, and determine under what conditions an intersection exists.

With these generalities on implicit curves and surfaces in mind, we can now move on to discuss the ways in which implicit functions are most often represented.

24.4 Representing Implicit Functions

While the examples in the preceding section were given in terms of explicit polynomial formulas, such an approach becomes impractical when we want to use implicit surfaces for modeling particular shapes: What polynomial in three variables, for instance, has a level set that has the shape of a dolphin? It's clear that searching for the appropriate degree and coefficients is an intractable task.

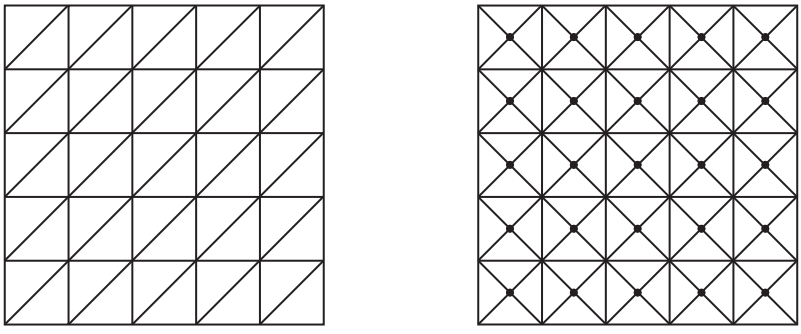
Instead, an implicit function is often represented by **samples,** the values of the function on a fixed grid of points such as the integer points of the plane or 3-space. (Such a representation arose naturally from the gathering of regularly spaced data in scientific experiments or surveying). Of course, knowing a function's values at integer points does not tell us the values at noninteger points. Indeed, between any pair of integer points, a function can take on any values at all. It's conventional to assume that the samples are so closely spaced that between samples, the function "doesn't do anything funny" so that, for instance, one might assume that between samples, the function takes on values that are determined by simple combinations of the values at the sample points (the same way we took values at points of a polygon mesh in Chapter 9 and extended them to define a function on the entire mesh). If we consider the plane, for instance, as a polygon mesh (with each polygon being a square), with values known at the vertices, we could interpolate over the interiors of squares. The methods of Chapter 9 don't help us, because they assumed that the mesh was made of triangles.

24.4.1 Interpolation Schemes

There are several approaches to extending a function defined on the integer grid in the plane to a function defined on the whole plane.

24.4.1.1 Conversion to Triangles

A first approach is to convert the mesh of squares into a mesh of triangles, as shown in Figure 24.6(a), by adding a diagonal to each square. Because there are two choices for the diagonal (and there's no particular *a priori* reason for



(a) (b)

Figure 24.6: (a) The mesh of squares defined by the integer points of the plane can be converted to a mesh of triangles by drawing a diagonal in each square. (b) It can also be done in a more symmetric way by adding a vertex at the center of each square (shown as a small dot) and breaking the square into four triangles.

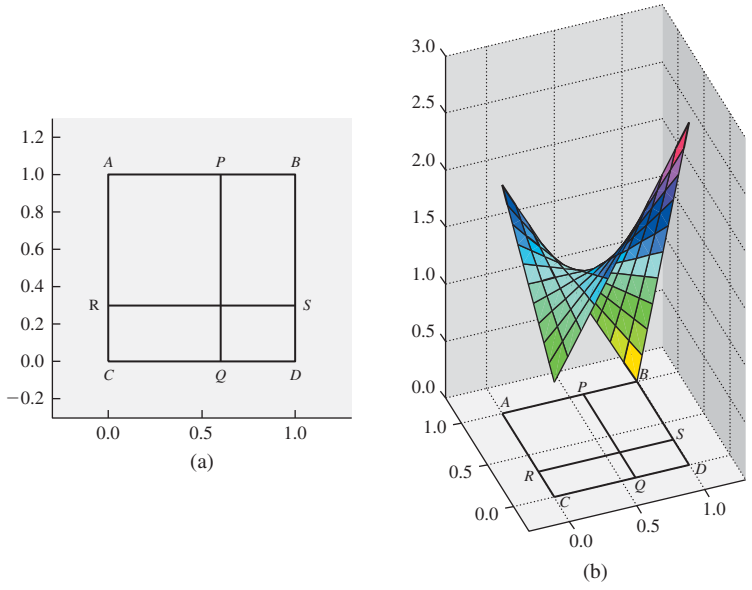


Figure 24.7: (a) If we know the values v_A, v_B, v_C, and v_D of a function at the points $A = (0,0)$, $B = (1,0)$, $C = (0,1)$, and $D = (1,1)$, we can compute a value at the point (x,y) inside the unit square by first interpolating values linearly at the points P and Q of AB and CD, respectively, and then interpolating between these; alternatively, we could interpolate the values at the points R and S of the edges AC and BD, and then interpolate between those. In either case, the resultant value is $(1-x)(1-y)v_A + x(1-y)v_B + (1-x)yv_C + xyv_D$. (b) The graph of the resultant function when $v_A = 1, v_B = 3, v_C = 2$, and $v_D = 0$. Notice that constant-x and constant-y cross sections of the graph are linear.

the choices to be the same for every square), this approach seems unsatisfactory, although for finely sampled data it's often quite adequate. Alternatively, as Figure 24.7(b) shows, we can break each square into four triangles by adding a point at the center. We typically assign this center point a value that's the average of the four corner values; we can then interpolate using the method of Chapter 9.

24.4.1.2 Bilinear Interpolation

A different approach is to insist that the interpolation, along each edge of the square, should be linear. With this in mind, we can take a square in our grid, as shown in Figure 24.7, and determine the value at the point (x,y) in the square by linear interpolation along a pair of parallel edges to get the values $v_P = (1-x)v_A + xv_B$ at P and $v_Q = (1-x)v_C + xv_D$ at Q, and then interpolate linearly between these to get $(1-y)v_P + yv_Q$ as the value at the interior point (x,y) of the unit square. (For any other square, we must use the fractional parts of the coordinates of x and y in place of x and y).

Writing this out in terms of the four corner values, we get

$$v = (1-x)(1-y)v_A + x(1-y)v_B + (1-x)yv_C + xyv_D \tag{24.13}$$

as the value at the point (x,y) of the unit square. Because the blending functions are all bilinear in x and y, this is called **bilinear interpolation.**

24.4.2 Splines

Bilinear interpolation can be seen as a blending of values at the four corners by certain polynomials, suggesting that any interpolating spline would also work, and indeed this is true: If we take any function h that's 0 everywhere except in the range $-1 \leq x, y \leq 1$, where it's nonnegative, and is 1 at $(0,0)$, we can define a function

$$F(x,y) = \sum_{i,j} v(i,j)\, h(x-i, y-j) \tag{24.14}$$

that takes on the known values $v(i,j)$ at each integer point (i,j). If, in addition, the function h has the property that F is everywhere 1 when all the $v(i,j)$ values are 1, then in general the value $F(x,y)$ will lie between the minimum and maximum values known at the four corner points. Examples of such functions are

- The box function on the unit square $-\frac{1}{2} < x, y < \frac{1}{2}$
- The bilinear basis function

If we weaken the requirement that $h(0,0) = 1$, other functions like the bicubic B-spline basis function can also be used.

Even more general functions can be chosen to play the role of h, but the key idea is simple: h represents how the effect of the value at each vertex fades as we move away from that vertex. In doing so, h encodes something of our belief about the implicit function that we're representing by samples.

24.4.3 Mathematical Models and Sampled Implicit Representations

As the previous sections show, given samples of a function on the integer grid, there's no single answer to the question, "What function do these samples come from?" And without an answer to that question, there's no hope of answering, "What implicit curve (or surface) do these samples define?" In the case of data gathered in an experiment, there may be little knowledge on which to base our choice of function, but it's clear that if the variation of the function over a grid cell is so large that the values at the corners of the grid cell fail to represent this variation faithfully, any interpolation and level-set finding is bound to give a wrong answer. It's therefore common to assume that the function being sampled is band-limited (i.e., its Fourier transform contains no frequencies higher than some specified frequency ω), and that the samples are spaced close enough to ensure that we can accurately reconstruct any such function from its samples. Indeed, if the samples are spaced twice as close as needed for reconstruction, then simple linear interpolation serves to approximate the function quite well, as we saw in Chapter 18. Unfortunately, approximating the true function F_0 by a function F whose value at each point is very near the value of F_0 does not ensure that a level set of F resembles the corresponding level set of F_0. To understand this, consider a very gradually sloping beach. A very small change in tide level can create a drastic shift in the shoreline; alternatively, a beach shaped only slightly differently can have a drastically different-looking shoreline. Thus, the level sets of F and F_0 need not be very similar at all.

This apparent contradiction—the defining functions are similar, but the implicit curves or surfaces are different—can be resolved, in part, by scaling: If we insist that we consider only functions F and level sets $F = c$ with the property that the gradient, at each point of the level set, has magnitude at least 1, then an alteration of F by some small enough amount δ results in a motion of the level surface that's $O(\delta)$. For acquired data, guaranteeing this property of the gradient may be infeasible. For cases where we are building implicit functions ourselves, it *may* be feasible. But if our interest in implicits is in their ability to represent changing topologies as the level value changes, then at the topology-changing level, we must have a point where the gradient is zero (so the assumption that the magnitude is greater than one is violated). In short, although it's possible to make guarantees of correctness for certain classes of implicit functions, in practice the hypotheses may be unenforceable or impractical.

24.5 Other Representations of Implicit Functions

Implicit surfaces are sometimes referred to as "blobbies," because it's so easy, with functions like $F(x, y, z) = x^2 + y^2 + z^2$, to create small blobs. Indeed, radially symmetric functions, translated to various points and summed, allow one to create multiple blobs. If $z = f(r)$ is a rapidly decreasing function of r with $f(0) = 1$, then we can define

$$F(P) = \sum_i f(\|P - P_i\|), \tag{24.15}$$

which will be a function with maxima at or near the points P_i (assuming that they're far enough apart), and the level set at level $c = 0.9$, for instance, will consist of approximately spherical blobs around the P_i. If two of the points P_i are very close, then their associated blobs will merge into a single larger blob, and this idea is the basis for modeling shapes with implicit functions: By choosing the points P_i carefully, we can build up a shape as a sum of blobs. This approach to modeling has been very thoroughly investigated [BW90, WGG99]; Bloomenthal's book [BW97b] provides a great many details. One approach, in which blobs blend in a very predictable way, was developed by Wyvill et al. [WMW86]. Critical to its success is finding a function f with the property that when blobs merge, the volume of the resultant blob is approximately the sum of the individual volumes.

If we consider an implicit function F as not defining a *surface* where $F = 0$, but rather a *volume* (the points P where $F(P) \geq 0$), then there are further operations we can consider. For instance, if F and G both define shapes, then $\max(F, G)$ defines the union of the shapes (the max is positive only if one of the two functions is positive), while $\min(F, G)$ defines the intersection. Unfortunately, the function $\max(F, G)$ is not necessarily smooth, even if F and G are. Since smoothness is often important in guaranteeing the quality of results for implicit surfaces, these functions are sometimes replaced by smooth approximations; with these smooth approximations, we get approximations of the union and intersection of shapes. By starting from simple shapes, defined by individual functions, and combining them with operations like translation, rotation, smooth-max, smooth-min, etc., we can create implicit representations of quite complex shapes (Figure 24.8 shows an example). Wyvill [BEG98] describes this **blob tree** approach in detail.

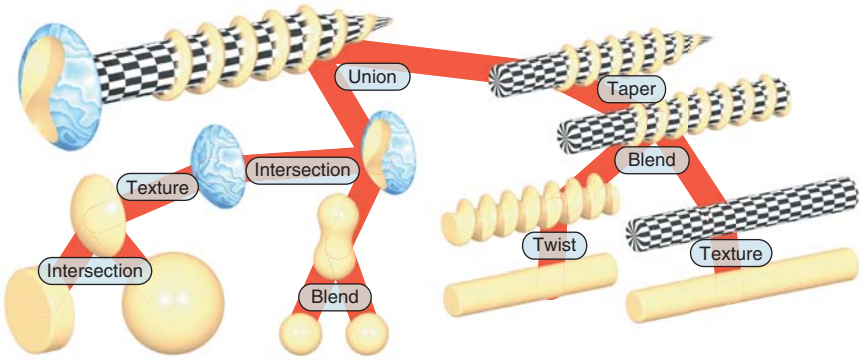


Figure 24.8: A complex shape created from simple implicit surfaces combined in a "blob tree," which defines a complex implicit function in terms of unary and binary operations on simpler implicits (Courtesy of Erwin de Groot and Brian Wyvill.)

Another approach to describing implicit functions, based on so-called "radial basis functions," is described in the web materials for this chapter.

24.6 Conversion to Polyhedral Meshes

An implicit function represented by samples on a grid can be converted to a polyhedral mesh; we'll discuss **marching cubes,** the most widely known method of doing so. Other implicit-function representations can be converted indirectly, first by sampling on a grid and then by applying marching cubes, but there are cases where it's possible to quickly find a point on each component of an implicit surface, and from this seed point construct the surface component directly [WMW86]. A rough estimate suggests that in an $n \times n \times n$ grid, one expects $O(n^2)$ polygons in an implicit surface mesh, but since marching cubes examine every cube of the grid, it takes $\Omega(n^3)$ time; thus, in cases where the structure of the implicit function gives *a priori* information, it can be very useful in reducing the isosurface-extraction time.

We'll first examine the iso-set extraction problem in two dimensions; most of the complexity of the problem is present there, but the pictures are easier to understand than those in three dimensions.

Our starting point is a grid of values; the desired output is a set of polylines representing the zero-set of the function associated to the values. We'll refer to this set of polylines as the output "mesh," in preparation for the three-dimensional example, even though it consists of only vertices and edges. Constructing the mesh can be divided into two tasks: determining the topology of the mesh (how many vertices and edges, and which are connected to which) and the geometry of the mesh (determined by the actual locations of the vertices). Figure 24.9 shows this process.

To simplify matters, we'll assume that no vertex has value 0; we'll return to this simplification after developing the remainder of the algorithm.

We'll also assume that if the topology of the isocurve within some grid square is indeterminate, then any answer consistent with the data is satisfactory. (We'll also return to this simplification later.)

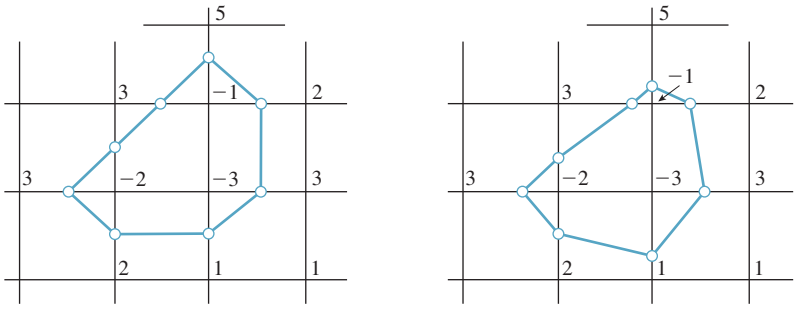


Figure 24.9: Starting from a grid of values, we first determine the topology of the isocurve for an associated function. Vertices are indicated by small circles at midpoints of edges. We then adjust the locations of the vertices to better match the input values (i.e., the small circles move to the place where a linear function on the edge would have a zero crossing). Thus, the process of isocurve extraction is divided into topological and geometric tasks.

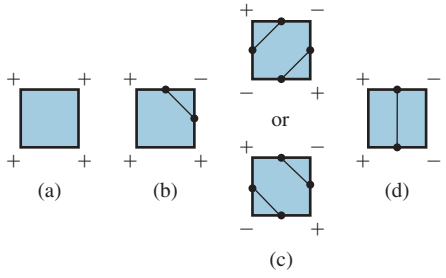


Figure 24.10: Patterns of signs on a grid square. (a) All plusses (or minuses). (b) One plus (or one minus). (c) Two diagonally opposite plusses. (d) Two adjacent plusses. All other cases are rotations or reflections of these. In each case, we've marked a dot at the center of each edge through which the isosurface passes, and shown possible patterns of edges by which these can be connected.

Finally, we'll assume that the function defined by values at the four vertices has no maxima or minima in the interior of each square, and interpolates the values linearly on each edge of the grid.

With these assumptions, we can classify each grid point as a "+" or "−" point, depending on whether the value there is positive or negative. If the ends of an edge have opposite signs, then the function must pass through zero somewhere on the edge, so we will place a vertex on that edge. Up to symmetries, there are only a few possibilities, shown in Figure 24.10. For each possibility, we've shown a way to draw in the isocurve within the grid square in a way that's consistent with the edge crossings on the boundary. In case (c), there are multiple ways to connect the edge crossings. We've shown two that result in isocurves with no self-intersections.

Choosing one way to fill in each possible configuration of edge crossings, we produce a topologically valid isocurve configuration.

Having done so, we can move each isocurve vertex from an edge midpoint to the correct location on the edge (i.e., where a linear function on the edge would have a zero crossing).

This isosurface construction approach has some rather nice properties.

- We can give each isocurve vertex a name consisting of the x- and y-coordinates of the endpoints of the segment it lies on, with the leftmost or

lower vertex coming first, so a vertex on the segment from $(1, 2)$ to $(1, 3)$ would have a name $(1, 2, 1, 3)$.

- We can process the grid of squares one at a time. For each square, we do the following.
  - Find the isocurve vertex associated to it.
  - If the vertex is new (we can use a hash table with the name as an index to check this in $O(1)$ time), we assign a new index to the vertex name, and add this index to our vertex table; if it's old, we do nothing with it.
  - For each new vertex, we use the values at the ends of the associated edge to determine its exact location, and record this in the vertex table.
  - Examine the pattern of plusses and minuses to figure out what edges must be added (we can do this with a lookup table in which the four plusses and minuses serve as a 4-bit binary index); then we add these edges to the edge table.
- The resultant set of isocurves has the property that every vertex (except those on the very boundary of the grid) is shared by exactly two edges; hence the resultant isocurves are all simple closed curves or polylines.

The square-at-a-time property will extend to 3D as well; because that 3D algorithm is called "marching cubes," this 2D algorithm can be called **marching squares.** In practice, it may make sense to process a whole row of squares at once to favor cache coherency.

Let us now return to the assumptions made at the start of our discussion of marching squares.

We assumed that no grid vertex had the value 0. If a vertex has value 0, but none of its neighbors do, we can adjust the value slightly (to, say, .001 times the next smallest vertex value adjacent to it). We then proceed with the rest of the algorithm, but at the very end, we adjust the positions of isocurve vertices that lie on edges leaving this grid vertex so that they are all this single vertex. This means that up to four different isocurve vertices may be at the same location so that the isocurve no longer necessarily consists of disjoint simple closed curves and polylines. Often, however, just two vertices get moved to sit at the grid vertex, and the edge between them ends up with zero length (see Figure 24.11), while the closed-curves-and-arcs property continues to hold. If four vertices all collapse to a single grid vertex, the property no longer holds.

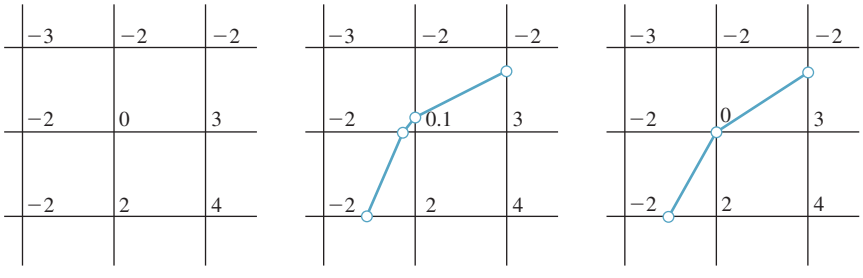


Figure 24.11: One grid vertex has value 0*; we adjust the value slightly and compute the isocurve (which passes near the grid vertex), and then, at the end, we move the two nearby vertices to the grid vertex so that the edge between them shrinks to nothing.*

If two or more adjacent vertices have value 0, more complex problems arise. For instance, if all four vertices of a grid square have value 0, then all edges of the square should be included in the isocurve, as perhaps should the whole square itself, which would make the isocurve no longer be a curve! We address these cases in the same way as the previous case: We adjust all 0 values slightly, compute the isocurve, and then adjust vertices at the end. But if a vertex lies on a grid edge where both ends have value 0, rather than moving the vertex to one end or the other, we place it at the middle of the edge. The result of this is an isocurve that's topologically correct in any grid cell with no 0 values at its corners, and is topologically consistent even in cells where there are zeroes. This is an instance of our second assumption—that in indeterminate cases, any consistent answer is acceptable—except that we do not always include the entire grid edge between two zeroes.

The difficulties of handling zeroes in the data are intrinsic to the original problem: In places where the graph of a function is nearly horizontal, level curves are *unstable,* in the sense that a small change in the input (the data values) results in a large change in the resultant level curve.

24.6.1 Marching Cubes

The **marching cubes** algorithm for finding an isosurface of a function specified at grid vertices in 3-space is exactly analogous, although there are some subtleties. Once again, it's easiest to assume that all input values are nonzero; if there's a zero in the input, perturb it by a small random amount, compute the isosurface, and then move the isosurface vertices back to the proper locations as we did in the marching squares algorithm.

Again, the output associated to a particular cube in the grid is determined by the pattern of plusses and minuses at its vertices. Since there are eight vertices, each with a plus or minus sign, we can encode the pattern of plusses and minuses with an 8-bit binary number; this can be used to index into a table of presolved examples, containing the vertex and triangle table for the mesh structure of the output; the actual locations of the vertices in the vertex table are once again determined by interpolation along edges of the grid.

Figure 24.12 shows two of these cases: The first generates a single triangle as output, and the second generates a rectangle, which would generally be represented by two triangles.

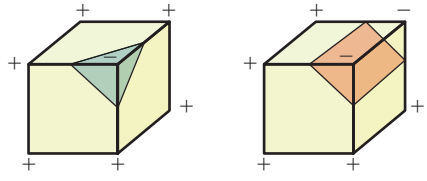


Figure 24.12: Two examples of patterns of plusses and minuses, and the associated bits of isosurface.

In the marching squares algorithm, a grid edge contained either no isocurve vertex or one isocurve vertex. In the latter case, each of the two adjacent grid squares had an edge that ended at that vertex, so each vertex met two edges, and the edges therefore fell into long chains (which either were closed curves, or terminated at the boundary of the grid). In the marching cubes algorithm, adjacent cubes meet along a face, as shown in Figure 24.13; these faces share isosurface vertices, but the way that the isosurface vertices are connected by edges within each copy of the face might not be consistent. If this happens, the resultant model of the isosurface will have edges in the interior of the grid, which is inappropriate. It's critical therefore that the 256 models used for the 256 possible cases in the marching cubes algorithm be pairwise consistent so that the resultant isosurface mesh either is closed or has boundary edges only on the boundary of the input grid.

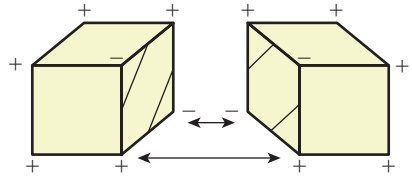


Figure 24.13: Two adjacent cubes in the marching cubes algorithm share a single face. The same four vertices appear on the four edges of this face, but the edges that join together pairs of isosurface vertices in each cube are not consistent with one another; the surface that results will have a boundary rather than being a closed surface.

As in the marching squares algorithm, the marching cubes algorithm is very well suited for a one-plane-of-data-at-a-time approach, in which the output

associated to a plane of grid cubes is computed all at once, and then the next plane of grid cubes is loaded into memory.

24.7 Conversion from Polyhedral Meshes to Implicits

Implicit representations have some important advantages over polyhedral models, as we've mentioned. It's not only possible to convert from implicit representations to polyhedral ones, but it's also possible to do the opposite: Given a nice enough polyhedral mesh, we can find a function F whose level set resembles the mesh. One class of meshes that is "nice enough" are ones with the property that the complement of the mesh—the set of all points in 3-space not on the mesh—can be divided into two sets with the property that each mesh face is on the boundary of both sets. If the mesh is a pair of cubes, for instance, one of the sets would be the interiors of both cubes, and the other would be the region exterior to both. Every face of the cube has the interior on one side and the exterior on the other. By contrast, a Möbius band is not "nice enough," because its complement consists of a single connected set.

When a mesh has this "two set" property we can declare one set to be "positive" and the other set "negative," and then define a function F on 3-space by the rule that $F(P)$ is the minimum straight-line distance from P to the mesh, multiplied by -1 if P is in the "negative" region. This is an implicit function (known as the **signed distance transform** of the mesh) whose zero-set is the mesh. Unfortunately, if we represent this function by grid samples, the level-zero isosurface will not be exactly the original mesh in general, but it will be very similar to it, provided the grid samples are closely enough spaced.

In general, however, interconverting between implicit and polyhedral models tends to be lossy, and should probably be avoided.

24.8 Texturing Implicit Models

Because implicitly defined models are generally not equipped with texture coordinates, it's common to use volume textures to texture them. Such volume textures can be procedurally defined, by a rule like

$$color = (0.3, 0.2, 0) + (0.2, 0.2, 0)\sin(x^2 + y^2), \tag{24.16}$$

which varies between dark brown and light brown cylindrical rings; an implicit object textured with such a function gets a (very simple) wood-like appearance. (Textures like this one, which can be expressed as a function of two coordinates, are sometimes called **projection textures,** because one can imagine the texture being projected from a 2D image out into space [Pea85].) More often, however, textures for implicitly defined objects are defined explicitly via a volumetric representation such as a voxel grid with colors at each voxel.

To avoid the cost of creating and storing all the voxels, while only a few are used for texturing, one can also use a hierarchical data structure like an oct tree in which most cells are empty, but ones near the implicit surface are filled in. This is also a natural structure to use in a painting interface, in which an artist directly paints texture (color, normals, displacement) onto a surface: In broad

constant areas, unrefined structure represents the texture compactly; in areas with finer detail, we can refine the oct tree so that it can hold this detail. This idea has been developed in some detail by DeBry et al. [DGPR02] and Benson and Davis [BD02].

24.8.1 Modeling Transformations and Textures

Just as we typically describe a polyhedral model in some modeling space, and then apply various transformations to it so as to put it in a particular location and orientation in world space, we typically define implicit models in some modeling space as well, and transform them into world space. For example, we define a sphere by the equation $F(x,y,z) = x^2 + y^2 + z^2 - 1 = 0$; to translate this sphere to the point $(1,3,4)$ we replace F by

$$G(x,y,z) = F(x-1, y-3, z-4). \tag{24.17}$$

Setting $G(x,y,z) = 0$ then gives a unit sphere centered at $(1,3,4)$. We can consider G as being constructed from F by the rule

$$G(P) = F(T(P)), \tag{24.18}$$

where T is the transformation "translate by $(-1,-3,-4)$," that is, exactly the *inverse* of the transformation we wanted to apply to the sphere.

Inline Exercise 24.2: The implicit formula for an ellipsoid of radii $(1/2, 1, 1)$ in x, y, and z is $x^2/4 + y^2 + z^2 - 1 = 0$.
(a) Letting $F(x,y,z) = x^2 + y^2 + z^2 - 1$, is the implicit equation of our ellipsoid $F(x/2, y, z) = 0$ or $F(2x, y, z) = 0$?
(b) What simple scaling transformation takes points of the unit sphere to points of our ellipsoid?
(c) How are parts (a) and (b) related?

In general, if S is a surface defined implicitly by the function F (i.e., if $F(s) = 0$ if and only if $s \in S$), the surface $T(S) = \{T(s) : s \in S\}$, where T is an invertible linear transformation, is implicitly defined by the function

$$G = F \circ T^{-1}. \tag{24.19}$$

In fact, the transformation T need not be linear—it need only have an inverse. This means that transformations like

$$T(x,y,z) = (x\cos z + y \sin z, -y \sin z + x \cos z, z), \tag{24.20}$$

which rotates each $z = c$ slice of 3-space by a different amount so that the strip $[-1,1] \times 0 \times \mathbf{R}$ gets twisted into a helical shape, can be used to apply a helical deformation to any implicitly defined object.

A shape that's been modeled implicitly and then transformed can be textured in world space (the texture at a point P with $F(T^{-1}(P)) = 0$ is determined by the coordinates of P itself) or in modeling space (the texture is determined by the coordinates of $T^{-1}(P)$).

24.9 Ray Tracing Implicit Surfaces

When we wanted to compute the intersection of a ray (parameterized in the usual form $t \mapsto P + t\mathbf{d}$) with an implicitly defined sphere, we ended up solving a quadratic equation in t which arose by writing out $F(P+t\mathbf{d}) = 0$, where F was the implicit function defining the sphere. But if the implicit object is more general, it's possible that the equation $F(P + t\mathbf{d}) = 0$ might have an enormously complicated form, and there may be no simple formula for finding its roots. In this case, we must fall back on numerical techniques for root finding [Pre95].

As we mentioned in Section 24.8.1, if S is an implicit surface defined by a function F, and T is a linear transformation, then the surface $T(S)$ is defined by $F \circ T^{-1}$. As hinted at in Exercises 7.17 and 11.13, the problem of intersecting a ray $t \mapsto P + t\mathbf{d}$ with $T(S)$ can be recast as the problem of intersecting a different ray with S. Since $T(S)$ is defined by $F \circ T^{-1}$, a point of our ray is on $T(S)$ exactly when

$$F(T^{-1}(P + t\mathbf{d})) = 0. \tag{24.21}$$

That's the same equation we get when asking when a point of the ray $t \mapsto T^{-1}(P) + tT^{-1}(\mathbf{d})$ is on the surface S. Finding a ray-surface intersection in the untransformed version of the implicit surface may be straightforward (as we've seen with the plane and the sphere in earlier chapters). This means that if you're willing to model a scene by applying transformations to several implicitly defined shapes like spheres or cylinders, you can ray-trace the scene by taking each ray, and for each object, apply the *inverse* of the object's modeling transformation to the ray's basepoint and direction vector. You then intersect this "back-transformed" ray with the pretransformed object to find an intersection point Q and a normal vector $\mathbf{n}$ to the object. Applying the modeling transformation to Q and its normal transformation to $\mathbf{n}$ gives the intersection point and normal in world space.

For scenes of modest complexity, this works well. For highly complex scenes, it's a better idea to use a bounding-box hierarchy to first determine which transformed implicit shapes the ray has a *chance* of intersecting, and then do the intersection test only on those that pass this test.

24.10 Implicit Shapes in Animation

 Implicit curves or surfaces can also be used in animation; within physically based animation, they play a major role under the name **level set methods** [OS88]. In such methods, there's some initial object of interest that is defined either by an implicit equality ($F_0(x, y, z) = 0$) for surfaces or by an inequality ($F_0(x, y, z) \geq 0$) for solids.[3] Various forces act on the surface or solid attempting to deform it in some way; these, in turn, are treated as attempts to deform the defining function F_0. One ends up with a differential equation of the form

$$\frac{\partial F_t(x, y, z)}{\partial t} = -\nabla F_t(x, y, z) \cdot \mathbf{v}(x, y, z), \tag{24.22}$$

3. These methods can also be applied in two dimensions to implicit curves.

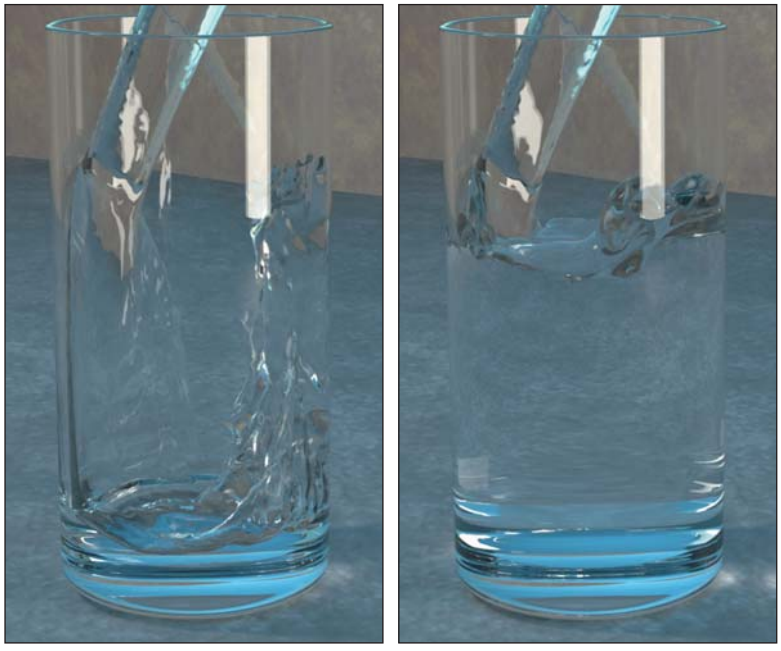


Figure 24.14: Water animated by the level-set method. Notice how droplets form and merge. (Courtesy of Stephen Marschner, ©2002 ACM, Inc. Reprinted by permission.)

where the vector field $(x, y, z) \mapsto \mathbf{v}(x, y, z)$ describes how the level set of the function F_t should move at the point (x, y, z). (The field $\mathbf{v}$ can be time-varying as well.) Solving this differential equation for F_t as a function of t gives the evolving shape of the object as the forces act on it.

Typically the forces act on the implicit surface itself, and therefore they may be known only at points where $F_t = 0$. On the other hand, the values of the function F at points far from those where $F = 0$ are unimportant, and so it's possible to keep track of F_t only near locations where $F_t = 0$. One way to do this is to extend F_t to nearby points by signed distance [LKHW03]; another way is to keep track of the values of F_t only in a narrow band around the set $F_t = 0$, and extend the field $\mathbf{v}$ to that band [AS95]. In either case, the function F_t is usually represented by voxel samples.

The advantage of the level-set approach to animation is that changing topology (like droplets of water merging into a single larger drop) is easy to generate. The method has been used to produce many of the most realistic fluid animations to date [EMF02] (see Figure 24.14).

24.11 Discussion and Further Reading

There's a duality between implicit and parametric models that we mentioned in Chapter 7, making them suited for different applications, and with the peculiar characteristic that finding the intersection between two models is easier when one is parametric and the other is implicit.

Implicit representations of shape as an *artistic* tool have fallen out of favor somewhat in recent years, but they have grown in popularity as shape

representations for simulation. This may be related to GPUs, where polygonal representations are strongly supported, or it may be related to the fact that having an artist build a function on *all* of space to get a surface that occupies just a tiny part of it is somewhat awkward. This awkwardness is partly avoidable by using the signed distance from the shape (positive outside, negative inside). To be useful in an implicit definition, this signed distance function need only be represented close to the zero-set: Its values far away cannot matter. The same is true for the volumetric texture functions described earlier in the chapter. Perhaps methods like this will rejuvenate the use of implicitly defined models.

One charm of implicit surfaces is that their geometry (i.e., tangent planes, curvature, etc.) is completely determined by the defining implicit function, and can be computed analytically. The web material for this chapter describes the computation of curve and surface curvature, for instance.

24.12 Exercises

Exercise 24.1: Explain why the formula of Equation 24.14 gives a function with the property that for any point (x, y) where x and y are both nonintegers, the value $F(x, y)$ lies between the minimum and maximum values of v at the corners of the square containing (x, y) *if* h has the property that when all the $v(i, j)$ are 1, the function F is everywhere 1.

Exercise 24.2: In the marching squares algorithm, we chose one of two possible ways to connect vertices in the case where the signs at the corner of a grid square alternated; the choice we made was independent of the values at the four corners.

(a) Explain why, if we drew a diagonal from the northeast to the southwest corner of each grid square, and treated the resultant collection of triangles as a mesh with values at vertices, the piecewise-linear interpolation of those values has a graph whose level-zero slice is consistent with our choice.

(b) Explain why, if we'd chosen the alternate diagonal for each grid square, it would be equivalent to making the other choice.

(c) Devise an algorithm in which we add a new vertex to the center of each grid square, with edges from this vertex to the four corners, and we assign a value to the new vertex that's the average of the four corner values. Use this new mesh (and these new values) to generate an isocurve for the piecewise linearly interpolated function. (Your new isocurve will have vertices both on the original grid edges *and* on the new edges you added from each center vertex to the corners.

(d) Explain how this revised approach can lead to either of the two possible ways of connecting the edge vertices in the $+ - +-$ case.

This page intentionally left blank

Chapter 25

Meshes

25.1 Introduction

Back in Chapter 8, we introduced meshes as a way to represent shapes in computer graphics. We now return to examine meshes in more detail, because they dominate present-day graphics. The vertex-and-face-tables model we introduced in Chapter 8 is widely used to represent triangle meshes, which are almost universally used in hardware rendering, because triangles are automatically convex and planar and there's only one possible way to linearly interpolate values at the three vertices of a triangle. Quad meshes, in which each face has four vertices, are also interesting in various situations. Indexing a regular planar quad mesh is very simple compared to indexing a regular planar triangle mesh, for instance. On the other hand, quads are not necessarily planar, they may be nonconvex, and if we have values at the four vertices of a quad, linear interpolation over the interior of the quad is likely to be undefined. So almost everything that's simple for triangles is more difficult for quads.

During geometric modeling, arbitrary polygon meshes, in which a face can have any number of vertices, can be a real convenience. In 2D modeling, for instance, the countries in a geography-based board game might be described by polygons with hundreds of vertices. Expressing these as triangulated polygons would introduce meaningless artifacts into the country descriptions. It would also amplify the space required to store the map. In general, such unconstrained meshes present all the problems of quad meshes, and more, but they have their place in situations where artistic intent or natural semantic structure in a model is important.

As we mentioned in Chapter 14, triangle strips and fans are sometimes used as a way to reduce mesh storage or transmission costs. In a triangle mesh, instead of having a list of vertex-index triples to represent faces, we have a sequence of triples like $(1, 4, 18, 9, 11, \ldots)$, which represents the triangles with vertices 1, 4, and 18, the one with vertices 4, 18, and 9, the one with vertices 18, 9, and 11, etc., as shown in Figure 25.1.

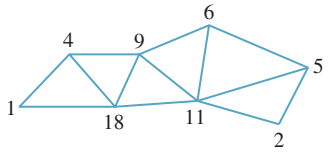


Figure 25.1: A triangle strip is represented by a stream of vertex indices; every group of three adjacent indices describes another triangle in the strip. The communication attributable to a typical triangle is therefore just a single vertex index, rather than three vertex indices.

At the abstract level—the mesh being represented—there's no important difference between the full representation and the stripped or fanned representation. The choice of whether to use strips or fans depends on whether transmission of geometry (typically from a CPU to a GPU) is the primary bottleneck, or whether something else, like **fill rate** (the speed of converting triangles into pixel values), dominates.

In *modeling,* as opposed to rendering, today's dominant technologies are splines (particularly NURBS) and quad-based subdivision surfaces; both of these are closely related to meshes. And in many application areas, including scientific visualization and medical imaging, nonmesh data structures like voxels or point-based representations dominate.

These preferred representations change depending on what kinds of data are at hand and on system properties such as memory size, memory bandwidth, bandwidth to the GPU, etc. Even now, voxel representations are being seriously considered by game developers, while work in scientific visualization has moved heavily toward meshes to make better use of GPUs. As with every other engineering choice described in this book, meshes will never be the universally right or wrong answer; they are just one more representation for your toolbox.

When we introduced meshes in Chapter 8, we did so as a means of representing geometry, like that shown in Figure 25.2. But there are other applications as well. Sometimes the vertex coordinates for a mesh are not physical locations, but instead are colors, or normal vectors, or even points in a configuration space (i.e., each vertex might describe a "pose" of a jointed figure; edges would then correspond to interpolations between poses, etc.).

The vertex-and-face-tables model of Chapter 8 appears to discount edges, which have no explicit representation. Nonetheless, faces are not all that matters. Edges are important in nonphotorealistic rendering, and in visibility-determination algorithms. Vertices are important in defining functions on meshes, as we discussed in Chapter 9. And most of all, *connectivity* is critical in many applications. The lack of explicit representation of connectivity or edges should not be taken as a mark of their unimportance.

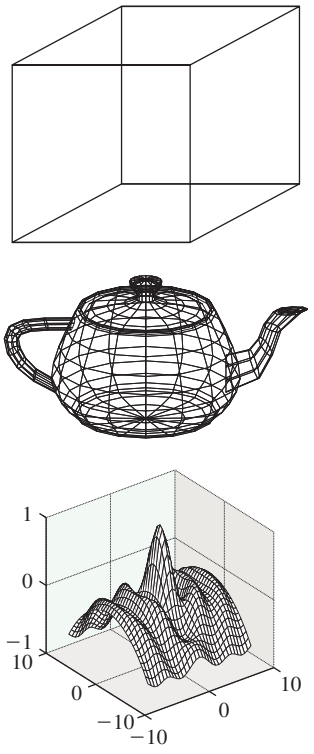


Figure 25.2: Polygonal meshes used to represent a cube, a teapot, and a smooth, wavy surface.

Inline Exercise 25.1: The vertex-and-face-table representation could be enhanced with an edge-table representation; each row of the edge table would contain indices of two vertices that were to be joined by an edge. In this exercise, we'll examine the consequences of introducing an edge table.
(a) Assuming that the edges in a mesh are exactly those that are part of the boundary of a face, describe an algorithm for deleting a face from a mesh.
(b) Suppose that edges are to be treated as **directed,** that is, the edge $(3, 11)$ is different from the edge $(11, 3)$, and that the same is true for faces. The boundary of the face $(3, 5, 8)$ consists of the three directed edges $(3, 5)$, $(5, 8)$, and $(8, 3)$. Assuming that we agree that the shapes we represent are all to be polygonal surfaces, discuss face deletion again. Make a Möbius strip from five triangles, and check whether your deletion algorithm works for the deletion of each triangle.

25.2 Mesh Topology

The **topology** of meshes—what's connected to what—is of primary importance in many mesh algorithms, which often proceed by some kind of adjacency search like depth-first or breadth-first search. In this section we'll discuss the intrinsic topology of meshes (i.e., those aspects that can be determined by looking only at the face table), and we'll briefly touch on the topic of **embedding topology** for meshes that are embedded in 3-space (i.e., for which there's a vertex table specifying positions of vertices, and for which the resultant mesh has no self-intersections, which we'll define precisely).

To specify the topology of a triangle mesh, we typically choose a collection of vertices (usually represented by vertex indices, so we speak of vertex 7, or vertex 2), a collection of edges, with each edge being a pair of vertex indices, and a collection of faces, with each face being a triple of vertex indices. We'll put additional constraints on these in a moment. Because a vertex consists of one index, an edge consists of two indices, and a face consists of three indices, it's useful to have a term that encompasses all three. We say that a k-simplex is a set of $k + 1$ vertices. Thus, a vertex is a 0-simplex, an edge is a 1-simplex, and a face is a 2-simplex. (These ideas can be generalized to describe tetrahedral meshes, in which 3-simplices are allowed, and to even higher-dimensional meshes.)

> For nontriangle meshes, a face may consist of a sequence of more than three vertices. All the algorithms presented here become substantially more complex when nontriangular faces are allowed in the mesh, so we'll generally consider only triangle meshes.

The **degree** of a vertex is the number of edges that contain the vertex. The term **valence** is also used, by analogy with atoms in a molecule. The vertex is said to be **adjacent** to the edges containing it, and vice versa. We generalize to say that the **degree of an edge** is the number of faces that contain the edge. (The edge is said to be adjacent to the faces, and vice versa.) We'll restrict our attention to meshes in which each edge has degree one, in which case it's called a **boundary edge,** or degree two, in which case it's called an **interior edge** (see Figure 25.3). An edge that's adjacent to no faces at all is called a **dangling edge** and is not allowed in our meshes.

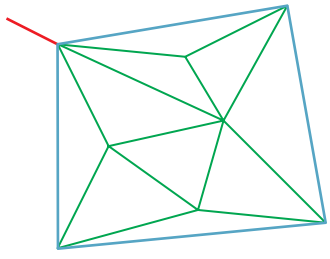


Figure 25.3: A small mesh with interior edges drawn in green and boundary edges drawn in blue. The red edge, which is not adjacent to any faces, is a "dangling" edge and is not allowed in our meshes.

25.2.1 Triangulated Surfaces and Surfaces with Boundary

To make our algorithms clean and provably correct, we will restrict the class of triangle meshes that we allow. We've already restricted our attention to meshes where each edge has one or two adjacent faces, but we will need some additional restrictions.

1. Each face may occur no more than once, that is, no two faces of the mesh can share more than two vertices.
2. The degree of each vertex is at least three.

3. If V is a vertex, then the vertices that share an edge with V can be ordered $U_1, U_2, \ldots, U_n$ so that $\{V, U_1, U_2\}$, $\{V, U_2, U_3\}$,. . ., $\{V, U_{n-1}, U_n\}$ are all triangles of the mesh, and

   (a) $\{V, U_n, U_1\}$ is a mesh triangle (in which case V is said to be an **interior vertex**), or
   (b) $\{V, U_n, U_1\}$ is not a mesh triangle (in which case V is said to be a **boundary vertex**),

   and there are no other triangles containing the vertex V (see Figure 25.4).

In the event that such a mesh has no boundary vertices, it's called a **closed surface;** if it has boundary vertices, it's called a **surface with boundary.**

Inline Exercise 25.2: Figure 25.5 shows three surfaces, each of which fails the requirements for being a surface or surface with boundary. Explain the three failures.

We have not yet said anything about orientation or orientability so that, for instance, the Möbius band is a perfectly admissible surface with boundary.

Inline Exercise 25.3: Show that the mesh with faces (1, 2, 3), (2, 3, 4), (3, 4, 5), (4, 5, 1), (5, 1, 2) and vertices 1, 2, 3, 4, and 5 is a surface with boundary (i.e., satisfies all the conditions above). Determine the set of boundary edges.

The **boundary** of a simplex is gotten by deleting each vertex of the simplex in turn, so the boundary of $\{2, 3, 5\}$ consists of the three sets $\{2, 3\}$, $\{2, 5\}$, and $\{3, 5\}$. Similarly, the boundary of the 1-simplex $\{2, 4\}$ consists of the two sets $\{2\}$ and $\{4\}$. The boundary of the 0-simplex $\{8\}$ consists of the empty set.

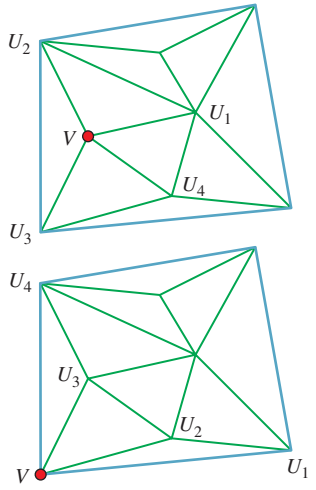


Figure 25.4: The red vertex marked with a dot in the top mesh is an interior vertex; the one in the bottom mesh is a boundary vertex.

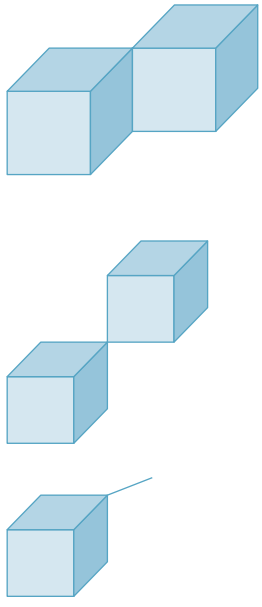


Figure 25.5: Each of these fails to be a surface mesh in some way.

25.2.2 Computing and Storing Adjacency

The basic mesh-storage structure is the face table, but adding and removing faces from an array is slow. Storing faces in a linked list rather than a table represented as a matrix allows low-overhead addition and deletion of faces, but operations like detecting adjacency can be expensive. The winged-edge data structure described in Chapter 8 makes adjacency computations very quick, but addition and deletion are relatively heavyweight operations.

Most adjacency-determining operations can be done, in a precomputation step, with hash tables. For instance, here's how to determine all boundary edges: First, for any face like $\{5, 2, 3\}$, there are three edges, each of which we'll represent with an ordered pair. But the first edge could be represented by the pair $(2, 5)$ or $(5, 2)$. We'll make the choice of representing the vertices in an edge in increasing order, so the edges are $(2, 5)$, $(3, 5)$, and $(2, 3)$. With this convention, finding the boundary edges is easy: We start with an empty hash table of edges. For each face of the mesh we compute the three edges, and for each such edge we either insert it in the table if it's not there already or delete it from the table if it is there already. When we've processed all the faces, the edges that remain in the table are those that appeared only once; in other words, they are the boundary edges.

Inline Exercise 25.4: The boundary-finding algorithm above assumed that the mesh was a closed surface or a surface with boundary, that is, it satisfied all the conditions for being a surface. Think about how you could use a similar algorithm to determine that the mesh did indeed satisfy all these conditions. Condition 3, which characterizes interior and boundary vertices, is a bit tricky, and you can skip that one if you like.

Our definition of a surface was quite general, but in practice most of the surfaces we meet in graphics are **orientable,** or even **oriented.** To define an oriented surface we need the notion of an **oriented simplex:** An oriented simplex is not just a *set,* but an *ordered set*. A 0-simplex is still just a vertex. A 1-simplex is an *ordered* pair (i,j) of distinct vertex indices. And a 2-simplex is an *ordered* triple (i,j,k) of distinct vertex indices, with the convention that (i,j,k), (j,k,i), and (k,i,j) are all considered "the same." (Alternatively, one can make the convention that the lowest-numbered vertex is always listed first; the order of the remaining two vertices determines the orientation of the face.)

The (oriented) boundary of an oriented face is built by deleting one vertex at a time, and reading the others in cyclic order, so the oriented boundary of the oriented face $(2,5,3)$ consists of the oriented edges $(2,5)$, $(5,3)$, and $(3,2)$.

We've defined the oriented boundary only for faces. Oddly enough, to define it for edges requires more machinery, as does defining it for tetrahedra and other simplices in higher-dimensional meshes.

For a mesh to be oriented requires that the faces be oriented simplices, and that if vertices i and j of edge e are contained in two distinct faces f_1 and f_2, then they must appear in the order (i,j) in one of them and (j,i) in the other.

An oriented face is often drawn symbolically with a small arrow indicating the orientation, that is, the cyclic order of the vertices. Figure 25.6 shows an example of two adjacent faces—the edge $(3,7)$ appears in the first; the edge $(7,3)$ in the second.

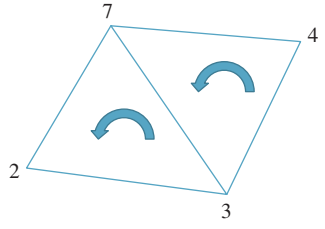


Figure 25.6: Two oriented adjacent faces, $(2,3,7)$ *and* $(7,3,4)$*. The oriented edge* $(3,7)$ *is in the boundary of the first face, while* $(7,3)$ *is in the boundary of the second.*

If we are given the face table for a *connected* unoriented mesh, we can try to create an *oriented* mesh from it: We take the first triangle, say, $\{2,6,5\}$, and assign it an order, say, $(2,5,6)$. We then seek out an adjacent face, say, $(9,2,5)$, and since it shares the vertices 2 and 5 with the first face, we have to place them in the opposite order, that is, we assign the order $(9,5,2)$. We continue in this fashion, assigning an order to each face adjacent to those already assigned, following either a depth-first or breadth-first strategy. If we encounter a face that's already been oriented, we check to verify that the orientation is consistent with the current face (i.e., it's the orientation we'd have assigned if it weren't already done!). If it's consistent, we ignore the face and continue; if it's inconsistent, we terminate the algorithm and report that the mesh is not orientable.

Inline Exercise 25.5: (a) Describe how to determine whether a mesh is connected (i.e., whether the graph consisting of all vertices and edges of the mesh is a connected graph).
(b) Apply the orientation algorithm to the five-vertex mesh of Inline Exercise 25.4 to show it's not orientable.
(c) Once we orient one face of an orientable connected mesh, all others have their orientations determined by the algorithm just described, so there are only two possible orientations of any connected orientable mesh. Suppose that some mesh M is *not* connected, and instead has $n > 1$ components. How many different orientations can it have?

The algorithm above can be slightly modified to compute the oriented mesh boundary: In the course of processing each face, if one of its edges is not part of a second face, we record that (oriented) edge as part of the oriented boundary.

Although we've described orientable meshes and an algorithm for determining an orientation (i.e., a consistent orientation for each face), in practice we usually encounter *oriented* meshes, or ones for which a particular orientation has already been chosen. When these meshes represent closed surfaces in 3-space, like a sphere or torus, the orientation of a face (i, j, k) is usually chosen so that if $\mathbf{v}_1$ is the vector from vertex i to vertex j, and $\mathbf{v}_2$ is the vector from vertex j to vertex k, then $\mathbf{v}_1 \times \mathbf{v}_2$ points "outward," that is, toward the unbounded portion of space.

Current work in collision detection primarily uses oriented meshes, and often stores triangles as lists of edges (or along with lists of their edges). The reason has to do with speeding up certain tests that occur often, namely, point-in-triangle tests. If you have a point P in the plane of a triangle ABC in space, and you want to test whether it's actually within the triangle, you can do three "Is this point on the right side of this plane?" tests to answer the question. Here's how: Let's suppose that the plane of ABC is S. Consider any plane J that contains the edge AB and is not parallel to S (see Figure 25.7). The plane J can be characterized by a point Q on the plane and a normal vector $\mathbf{n}$; the equation for J is then

$$(X - Q) \cdot \mathbf{n} = 0. \tag{25.1}$$

To determine whether P is on "the right side" of J, we can substitute C for X in Equation 25.1; the result will be nonzero. If it's negative, we'll replace $\mathbf{n}$ by $-\mathbf{n}$, so we can assume that it's positive:

$$(C - Q) \cdot \mathbf{n} > 0. \tag{25.2}$$

Now that we've arranged for $\mathbf{n}$ to point in the proper direction, the test for P being on the right side of J becomes

$$(P - Q) \cdot \mathbf{n} > 0. \tag{25.3}$$

If we compute Q and $\mathbf{n}$ once, we can associate them to the edge AB and reuse them for every point-in-triangle test.

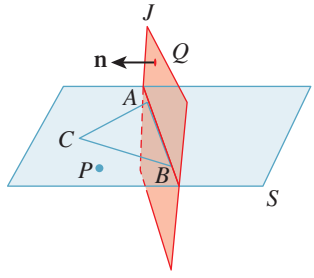


Figure 25.7: The point P is on the same side of the plane J as C, and therefore may *be in the triangle ABC. If it passes corresponding tests for planes containing the edges BC and CA, we know it's in the triangle.*

Inline Exercise 25.6: The point-versus-plane test involves a point subtraction, a dot product, and a comparison. Assuming that all points are stored as triples of coordinates, we can consider the special point $Z = (0,0,0)$ and rewrite the test as $((P-Z)-(Q-Z))\cdot\mathbf{n} > 0$, or $(P-Z)\cdot\mathbf{n} > (Q-Z)\cdot\mathbf{n}$. Assuming that this computation appears in the innermost loop of your program so that you're willing to ignore the point-versus-vector distinction, and assuming further that you're willing to do more precomputation, how few operations can you use to do this computation for each new test point P? (Answer: three multiplies, two adds, one compare. Your job is to verify this.)

One final data structure for meshes—and one that is particularly useful for meshes that will remain mostly topologically static, that is, for which addition and removal of faces is very rare—is an enhancement of the triangle mesh in which row i of the face table stores the indices of the three vertices for triangle i, but *also* stores the indices of the three triangles that are adjacent to triangle i. (If triangle i has one or two edges on the boundary of the mesh, then the corresponding index is set to `invalid`.) This structure speeds up coherent searching of the mesh data structure, especially for processes like contour finding. (If edge e is a contour edge as seen from some viewpoint, then the edges that meet e are very good candidates to also be contour edges.)

There are analogous structures for quad meshes. The difference between triangles and quads may seem small, but it's nontrivial. If you can replace a pair of adjacent triangles with a quad, you've turned five edges into four, and that may mean a 20% speedup in your program.

25.2.3 More Mesh Terminology

The **star of a vertex** consists of the vertex itself and the set of edges and faces that contain that vertex (i.e., like a "neighborhood" of the vertex). The **star of an edge** consists of the edge itself and any faces that contain the edge. Figure 25.8 shows the star of a red vertex in red tones and the star of a green edge in green tones. The **link of a vertex** is the boundary of the star. Suppose V is a vertex in a closed mesh. Each vertex in the link of V is separated from V by a single edge. These vertices are therefore called the **1-ring** of V; those separated by a distance of two edges are called the **2-ring** (see Figure 25.9). The vertices of the 1-ring of an interior vertex can always be organized into a cycle; that follows from the definition of a surface mesh. For the 2-ring, bad things can happen. For instance, in a tetrahedron, the 2-ring of any vertex is empty; in an octahedron, the 2-ring of any vertex is a single vertex. Figure 25.10 shows a mesh in which the 1-ring of a vertex is nice, but the 2-ring forms a figure-eight shape.

When we compute the boundary of a mesh, the boundary edges form chains; each such chain is called a **boundary component,** and the number of boundary components is often denoted by the letter b.

If a mesh surface M has v vertices, e edges, and f faces, the number $\chi = v-e+f$ is called the **Euler characteristic** of M, and it measures the "complexity" of the surface: A spherical topology has characteristic 2; a torus has characteristic 0; a two-holed torus has characteristic -2; and in general, an n-holed torus has characteristic $\chi = 2-2n$. If the n-holed torus also has b boundary components, then the formula becomes $\chi = 2-2n-b$.

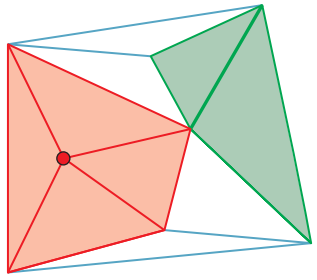


Figure 25.8: The star of a simplex consists of all simplices containing it.

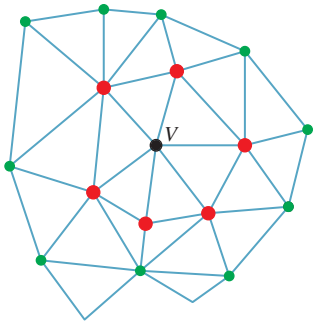


Figure 25.9: The 1-ring of V is drawn in large red dots; the 2-ring in smaller green dots.

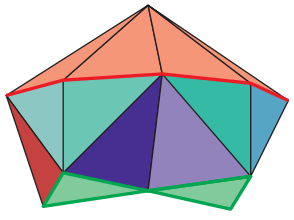


Figure 25.10: The star of the top vertex is drawn in brown; the 1-ring, which forms an octagon in the middle level, in red. The 2-ring, at the bottom drawn in bright green, is connected into a figure-eight shape.

25.2.4 Embedding and Topology

When we assign locations to the vertices of a surface mesh (and to the rest of the mesh by linear interpolation), we want the resultant shape to resemble a surface that has an interior and an exterior, and no self-intersections. To distinguish between the abstract surface mesh (a list of vertices numbered $1, \ldots, n$, and a list of "faces," or triples of vertex indices) and the geometric object associated to it, we'll use lowercase letters like i and j to indicate vertex indices, and P_i and P_j to indicate the geometric locations of vertices i and j. We'll also use $C(P_i, P_j)$ to denote all convex combinations of P_i and P_j, that is, the (geometric) edge between them, and $C(P_i, P_j, P_k)$ to denote all convex combinations of the vertex locations P_i, P_j, and P_k, that is, the (geometric) triangle with those points as vertices.

With this terminology, we'll define an **embedding** of a surface mesh as an assignment of distinct locations to the vertices, extended to edges and faces by linear interpolation and satisfying one property: The triangles $T_1 = C(P_i, P_j, P_k)$ and $T_2 = C(P_p, P_q, P_r)$ intersect in $\mathbf{R}^3$ only if the vertex sets $\{i, j, k\}$ and $\{p, q, r\}$ intersect in the abstract mesh. If the intersection is a single vertex index s, then $T_1 \cap T_2$ must be P_s; if the intersection has two vertices s and t, then $T_1 \cap T_2 = C(P_s, P_t)$; and if the intersection is all three vertex indices, then T_1 must be identical to T_2. (Note that we're assuming that i, j, and k are distinct and p, q, and r are distinct; otherwise, either T_1 or T_2 would not be a triangle.)

Figure 25.11 shows examples of nonembedded meshes. In the first example, two triangles intersect in their interiors. In the second, two triangles intersect at a point that is a vertex of one triangle, but is mid-edge on the other triangle. In the third, the intersection (shown in aqua) of two shaded triangles is only part of an edge of the left one (a situation known as a **T-junction**).

When we have both a face table and a vertex table, we can test whether the resultant geometric mesh is embedded or not, but the operation is expensive and prone to numerical errors. First, the entries of the vertex table (i.e., the vertex locations) must all be distinct. Second, for every pair of faces the geometric intersection of the faces must be empty unless the abstract faces share an edge or a vertex, in which case the geometric intersection should match the abstract intersection. Good modeling software is designed to ensure that only embedded meshes get produced so that such tests are not necessary.

If a mesh is **closed,** that is, its boundary is empty, and if it's embedded, then (1) the mesh must be orientable, and (2) the embedded mesh divides 3-space into two parts: a bounded piece called the **interior** and an unbounded piece called the **exterior.** The first statement is proved by Banchoff [Ban74]; the second, which may seem obvious, is really quite subtle, and is a consequence of the Alexander Duality Theorem [GH81], which is far beyond the scope of this book. To determine whether a point P that's not on the mesh itself is in the interior or exterior, we can create a ray r that starts at P and travels in some direction $\mathbf{d}$, missing all vertices and edges of the mesh. If the ray r intersects k faces of the mesh, then P is inside if k is odd and outside if k is even. The direction $\mathbf{d}$ can be produced by picking a direction at random; with probability one the ray r will miss all vertices and edges of the mesh.

A closed embedded mesh is, from a geometric and algorithmic point of view, an ideal object. It's suitable for use in ray tracing, in rendering with backface culling, for shadow-volume computations, etc. Furthermore, if a closed mesh is embedded and we move the vertex locations by a small enough amount, the

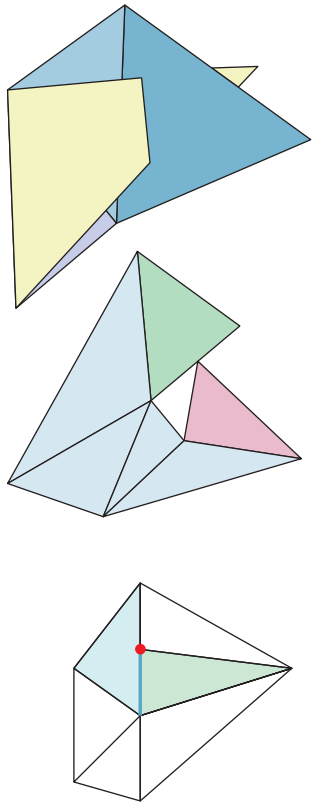


Figure 25.11: (Top) A mesh with bad self-intersections. (Middle) A mesh in which a vertex of the pink face at the right lies in the middle of an edge of the green face at the top right. (Bottom) The red vertex marked with a dot is a T-junction.

resultant mesh is *still* embedded. "Small enough" in this case means "less than $\epsilon/2$, where ϵ is the smallest distance from any vertex to any other vertex, or to the edge between any two other vertices, or to the plane containing any other three vertices, or from any edge to any nonintersecting edge of the mesh.

A closed, embedded surface mesh is sometimes called a **watertight** model, although it's possible to have a model that's intuitively "watertight" without satisfying the definitions for closed-ness, surface-ness, and embedding. As a simple example, a cube that has an additional square dividing it into two half-cubes is "watertight," in the sense that water placed in either of the two "rooms" can't escape, but it fails the tests of "surface-ness" and embedding. At a more practical level, it can be convenient to create a video-game model of, say, a robot character in which the torso is a polygonal cylinder (with endcaps) and the upper arm is an open-ended triangular prism that goes through the torso (see Figure 25.12). Clearly no water can leak from the torso into the upper arm, and this structure makes it easy to adjust the arm position over some range without worrying about things matching up perfectly.

On the other hand, while modeling and animating the character with this approach is easy, rendering it, or creating shadows from it, may be quite difficult. The same goes for models containing T-junctions.

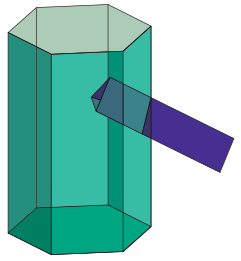


Figure 25.12: A see-through view of the torso and arm of a simple robot character.

25.3 Mesh Geometry

From now on, we'll assume we're working with embedded oriented surface meshes, although not necessarily closed meshes—there may be boundary components.

We've suggested that surface meshes are the discrete analog of smooth surfaces, but there are subtleties. On a smooth surface, for instance, at each point P there's a tangent plane that approximates the surface near P: In a small enough region around P, projection from the surface to the tangent plane is injective, that is, no two points in the neighborhood project to the same place on the tangent plane. Figure 25.13 shows this for the sphere. By contrast, there are quite simple meshes for which there is no such plane. Figure 25.14 shows an example of a vertex with the property that *no* plane passing through it will serve as a "tangent plane"; projection in any direction is never injective.

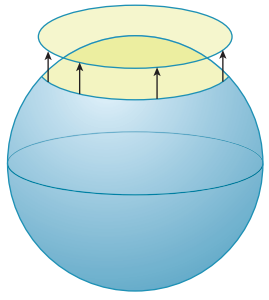


Figure 25.13: In this simple model of Earth, a small disk around the North Pole projects injectively onto a disk in the horizontal tangent plane to the North Pole.

The situation shown in Figure 25.14 may seem pathological, but it can arise quite easily during the gluing together of points obtained from a surface by scanning. A vertex like the one in the figure is said to be "not locally flat"; a **locally flat** vertex P is one for which there's a vector $\mathbf{n}$ such that projection from the star of P onto the plane through P with normal $\mathbf{n}$ is injective. Generally speaking, vertices that are not locally flat tend to break algorithms; it's a good idea to check that the vertex normal $\mathbf{n}$ assigned to each vertex P of a mesh demonstrates the local flatness property at P.

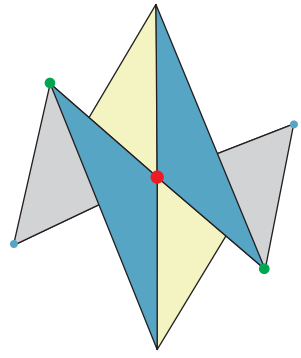


Figure 25.14: Projection from the mesh onto any plane passing through the central red vertex will not be injective. The large green vertices are closer to the eye; the small aqua ones are farther away.

The local flatness example demonstrates a general phenomenon: Things that you know about smooth surfaces don't directly apply to meshes. For instance, the slices perpendicular to some generic direction $\mathbf{d}$ of a smooth surface tend to be nice, smooth curves, except at isolated "critical" points, where there may be self-intersections that look like the letter X, as shown in Figure 25.15. By contrast, the slices of a surface mesh are polygonal curves, and at critical points they can look very complicated indeed [Ban65].

As another example, the **contours** of a smooth surface, that is, the points where the view ray lies in the tangent plane, generally form a set of smooth, closed curves on the surface, curves that don't intersect one another. For instance, the contour of a sphere is always a circle on the sphere's surface. For some surfaces, from *some* points of view, the contours may have intersections or sharp corners, but from a randomly picked direction they will be smooth. By contrast, the contours of a closed triangulated mesh consist of edges that lie between front-facing and back-facing polygons. These contours are therefore polygonal rather than smooth curves. But as Figure 25.14 shows, the polygonal curves may intersect, and may do so *generically* (i.e., there's a nonzero probability that for a randomly chosen direction such problems occur).

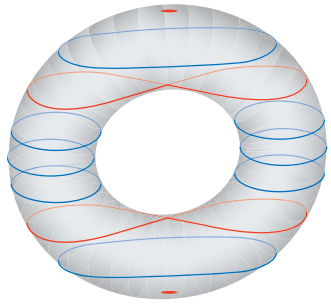


Figure 25.15: The slices of a torus are smooth (blue) except at the min, max, and the two "critical" levels where the slices are figure-eights (red).

To the degree that we treat triangulated meshes as surfaces in their own right, rather than as approximations of smooth surfaces, these oddities are hardly surprising. In a mesh, almost all points are flat: The curvature, no matter how you measure it, is zero. On the other hand, the curvature along edges or at vertices is, by the usual measures, infinite. It's as if the curvature that was spread out over a smooth surface got "condensed" into tiny packets of high curvature. All the techniques of differential geometry, which depend on taking derivatives of a parameterization of the surface, must be rethought in this context. The resultant **discrete differential geometry** is an active area of research [Ban65, BS08].

25.3.1 Mesh Meaning

In Figure 25.2, the cube has sharp edges and is *intended* to be a sharp-edged shape. By contrast, the wavy surface does have a bend at each edge, but the bend is very small, and the intent is that it should be seen as a smooth shape. When you work with a mesh in a graphics program, you don't generally know which one of these is intended. Expressed differently, you know the *data* for the mesh, but you don't know what it *means*. Just as RGB image formats that fail to say what R, G, and B mean can have many interpretations (see Chapter 17), meshes that don't include a description of their meaning are ambiguous, and any single interpretation you choose in writing a program will be wrong in some cases. This is an example of the Meaning principle: The numbers (and other data) used to represent meshes don't have enough meaning to be unambiguous, and this failure of meaning leads to failed algorithms.

For the bulk of this chapter, we'll assume that meshes are being used to approximate smooth surfaces, thus assigning a meaning to each mesh. In Chapter 6, we saw how this assumption can lead to problems. The pyramid looked odd until we remade it from several different meshes. But this remaking entailed its own problems: To make the pyramid taller now requires that we move not a single vertex, but several copies of that vertex. The important topological structure was broken in order to provide a correct rendering structure. For instance, although the original pyramid might have been a watertight mesh, the new pyramid is not. That's partly a consequence of the design of graphics interfaces. For instance, both OpenGL and DirectX require that all the properties of a vertex, such as its normal vector and texture coordinates, be tied to the vertex position in an indexed triangle strip. If you want to make a shape like a cube, where a single vertex must have multiple normal vectors, you're compelled to create multiple vertices at the same location, leading to nonwatertight models. And all current hardware APIs specify properties either at the per-mesh level or at the per-vertex level, so concepts

like face color have to be implemented by a similar vertex-duplicating strategy in which all three vertices of a face are assigned the same color, requiring multiple co-located vertices and nonwatertightness of the meshes.

As you think about applying any particular algorithm to your own meshes, be sure that your assumptions about the mesh and those made by the algorithm designer actually match, or expect bad results.

25.4 Level of Detail

A model of an office building might contain millions of polygons, both for interior detail and for things like windows, frames, exterior trim, etc. If that building is to appear in the far distance in some scene, it may suffice to remodel it as a single rectangular box. Indeed, if you do not remodel it that way, then rendering just a few city blocks will rapidly consume your entire budget for polygons, while perhaps only determining the final appearance of a small fraction of the pixels in your image. This is a clear misallocation of resources. If you're using basic z-buffering to determine visibility, and your building occupies, say, 100 pixels of the image, then (since each pixel's color is determined by the frontmost polygon drawn in it) all but 100 of your 1 million polygons will have been drawn to no avail.

It's possible to improve this situation substantially with visibility hierarchies so that, for instance, none of the polygons interior to your building get drawn. But for a building with substantial exterior detail, this is merely a palliative measure. What's really needed is a different model of the building, used when the model is in the distance. If you make an animation, the high-detail model is used when the building is nearby and the low-detail model is swapped into its place as the building recedes into the distance. Naturally, it's important that the swapping process be relatively undetectable, or the illusion of the animation will be broken. The substitution of a simpler model for the completely detailed model can be done in stages; in other words, it makes sense to build a model that includes multiple **levels of detail,** each one used where appropriate.

The inclusion of levels of detail represents a substantial architectural shift. Normally we imagine a renderer asking each object for a polygonal representation of itself, and then producing an image from these polygons. In a system using level of detail, the renderer must ask the object for a polygonal representation *and* provide some information about the level of detail. This information might be something like the distance from the camera to the object's center, or a request for the object to provide a representation with no more than 10,000 polygons, or a request to provide one of three or four standard levels of detail.

Inline Exercise 25.7: I can render a building from nearby with a wide-angle lens; the building fills most of the image. I can render it from very far away with a narrow-angle lens, and again it fills most of the image. What does this suggest to you about using the distance to the object as a level-of-detail cue? What might improve it?

A useful approach for determining level of detail is to have a coarse representation of the model, such as a bounding box; the rendering software can then quickly determine the screen area of the bounding box, and it can use this to help

the object select the appropriate level of detail. In expressive rendering (see Chapter 34), we sometimes elide detail on an object for reasons other than screen size (e.g., we might draw the details on only one or two faces in a crowd, because they are the important ones); in such a case, the level-of-detail negotiation between renderer and object may include hints other than the purely geometric ones discussed here.

While we've described level of detail here as a solution to a resource-allocation problem, it is more than that: Once we have decided to produce a final image using a single-sample z-buffer technique (or any other approach that uses a small, fixed number of samples per pixel), we've implicitly settled on a "scale" for which we can hope to produce correct results. Assuming, for the sake of argument, one sample per pixel, any geometric feature—a step, a windowsill, a doorknob—whose projected size is less than two pixels will produce aliasing instead of being accurately reconstructed. Because of this, we'd like to remove all such features, in a sense "prefiltering before sampling." Thus, using a level-of-detail approach is a matter of *correctness* as well as efficiency. We summarize these ideas in a principle:

✓ **LEVEL OF DETAIL PRINCIPLE:** Level of detail is important for both efficiency *and* correctness.

That being said, the "correctness" provided by level-of-detail simplifications is not always all that one might wish. Consider the front of a building shown in Figure 25.16. The natural "simplification" of this wall is to replace it with a single planar wall.

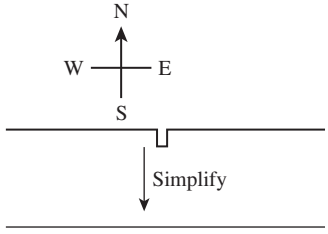


Figure 25.16: The front wall of a building, seen from above. Notice the narrow embossed portion of the wall in the center. The sides of this embossment will reflect light from the east or west, while the rest of the wall will not.

But now consider the light-reflection properties of these two versions of the building's front. If we assume that the front of the building is made of a somewhat glossy material, then in the unsimplified wall, some light from the east will reflect back toward the east, and some light from the west will reflect back to the west, while lots of light from the south will reflect back toward the south. The bidirectional reflectance distribution function (BRDF) of the wall as a whole, for three incoming light directions, is shown in Figure 25.17. The BRDF for the simplified wall is rather different, since it's everywhere zero for light from the east, for instance.

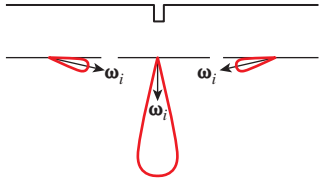


Figure 25.17: The BRDF of the wall, drawn for light coming from the east, south, and west.

We could, as we simplified the wall, still represent the geometry by encoding it in a normal map. But as we decrease the level of detail on the building further, the normal map itself will have to be simplified as well, to avoid representing too-high frequency changes. At that point, we can amalgamate the different reflection characteristics of the surface at different points into a single BRDF that resembles the "various facets in the wall look a lot like the microfacets" concept used in the Torrance-Sparrow and Torrance-Cook models, in the sense that they are geometric features that are too small (in screen coordinates) for us to detect, and hence we aggregate their effects into a BRDF.

We saw this sort of thing before, back in Chapter 18: If we're taking samples from a function in hopes of saying something about it, then our sampling rate should exceed the Nyquist rate for the signal, or we'll suffer aliasing. In this case, the "function" could be either "the x (or y or z)-coordinate of the points on the

surface," or "the BRDF of the surface" (considered as a function of position on the surface), or "the color of the surface," etc.

In fact, many of the techniques we've encountered—BRDFs, normal maps, displacement maps—provide representations of geometry at different scales. The BRDF (at least in the Torrance-Sparrow-Cook formulation) is a representation of how the microfacet slope distribution affects the reflection of light from the surface. We *could* model all those microfacets, but the space and time overhead would be prohibitive. More important, the result of sampling such a representation (e.g., ray-tracing it) would contain horrible aliases: A typical ray hits one particular microfacet and reflects specularly, rather than dispersing as we'd expect for a diffuse reflection. Displacement maps and normal maps represent surface variation even for surfaces that are, at a gross scale, represented by just a few polygons. Because of their formulation as maps (i.e., functions on the plane), they can be filtered to reduce aliasing artifacts, using MIP mapping, for example.

Fortunately, these techniques also constitute a hierarchy of sorts: As you simplify one representation, you can push information into another. A crinkled piece of aluminum foil, for instance, can be modeled as a complex mesh with a very simple (specular) BRDF, if it's seen close-up. As it recedes into the distance, we can replace the complicated geometry with a simpler planar polygon, but we can represent the "crinkliness" by a normal map and/or displacement map. As it recedes farther into the distance, and variations in the normal map happen at the subpixel scale, we can use a single normal vector, but change the BRDF to be more glossy than specular, aggregating the many individual specular reflections into a diffuse BRDF. Many of these ideas were present (at least in a nascent form) in Whitted's 1986 paper on procedural modeling [AGW86].

The correspondence with the sampling/filtering ideas is more than mere analogy: In rendering, we're trying to estimate various integrals, typically with stochastic methods that use just a few samples; from these samples, we implicitly reconstruct a function in the course of computing its integral. If the function is ill-represented by the samples, aliasing occurs. In one-sample-per-pixel ray tracing, for instance, any model variation that occurs at a level that's smaller than two pixels on the screen must either (a) be filtered out, or (b) appear as aliases.

In some sense, these observations give an abstract recipe for how you ought to do graphics: You decide which radiance samples you'll need in order to represent the image properly, and then you examine the light field itself to determine whether taking those samples will generate aliases. If so, you determine what variation needs to be removed from the light field; since the light field itself is determined by the rendering equation, you can then ask, "What simplification of the illumination or geometry of this scene would remove those problems from the light field?" and you remake the model accordingly. When you set about rendering *this* model, you get the best possible picture.

This "recipe" is an idealized one for several reasons. First, it's not obvious how to simplify geometry and illumination to remove "only the bad stuff" from the light field; indeed, this may be impossible. Second, determining the "bad stuff" in the light field may require that you solve the rendering equation with the full model as a first step, which returns you to the original question. A compromise position is that if we filter the light sources to not have any high-frequency variations, and we smooth out the geometry so that it doesn't have any sharp corners (which lead to high-frequency variations in reflected light), then the "product" of light and geometry represented by the rendering equation will end up without too

much high-frequency information in it. While this loose statement may be correct, it's not the case that the smoothed lighting of smoothed objects produces the same result as taking full-detail lighting of the full-detail model and smoothing the results. The "smoothing" operation (filtering out high frequencies) does not commute with the "product" operation in the rendering equation (another instance of the Noncommutativity principle). Then again, it's the best we've got at present, and it's an approach that forms the basis for many techniques.

Before leaving this high-level discussion, we have two more observations. The first is that, in thinking about graphics, we tend to think about the kinds of models being used to represent the world, and it's easy to confuse the "nature of the world" with the "nature of models representing it." As three points on the continuum of model classes, consider (a) tessellation-independent models, like the implicit surfaces used in making a simple ray tracer, (b) collections of triangles and/or "primitives" like cubes, spheres, cones, spline patches, etc., combined with unions, intersections, etc., to describe shapes, and (c) "object-based" graphics, in which the world is populated by objects, each one modeled in its own way and each one supporting various operations like "Where does this ray intersect you?" and "Give me a simplified representation of yourself." If objects are themselves represented as meshes (as they often are), it's very natural to ask, "What's a simplified representation of this object?" to do level-of-detail computations at a per-object level. The problem with this approach is that the result of simplification *depends on our description of the world, not the world itself.* For instance, if we have a sphere that's represented by an icosahedral mesh, it's natural to simplify it by replacing that mesh with an octahedron or tetrahedron. But if we happen to have created that same shape with 20 objects, each of them a single triangle, then there's no possible simplification: Each triangle is as simple as can be. To give another example, if we have two irregularly shaped objects partly overlapping each other (see Figure 25.18), and we simplify each of them to remove fine detail, the gap between them can remain as a small detail, and hence a source of aliasing when we sample the scene. This happens in practice when we model a city as many buildings: Even when all the buildings are simplified to cuboids, the spaces between them may be rectangular gaps that are so small, in screen space, that they produce aliases.

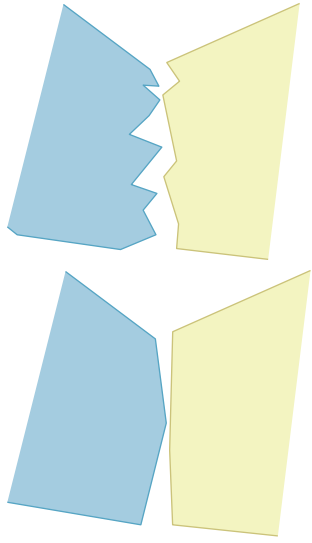


Figure 25.18: Two irregularly shaped objects overlap; when we simplify each one, getting rid of small details, the space between *the objects remains as a small detail, unrecognized by our simplification process.*

One approach to level of detail is therefore to consider the simplification of a whole scene at a time. Perbet and Cani [PC01] took this approach in modeling prairies: Grass near the camera was modeled as individual blades; slightly more distant grass was represented by flexible vertical panels on which the blades of grass were "painted." And very distant grass was represented by textures applied to large horizontal planar polygons. The intermediate representation—a textured polygon that faces the viewer—is one instance of a **billboard.** Numerous permutations of this billboard idea have been used over the years, from representing trees by an intersecting pair of billboards (which looks decent, except when viewed from above), to increasingly complex combinations such as the representation of clouds by multiple semi-transparent billboards [DDSD03], to the representation of crowd characters by billboard assemblies with time-varying textures [KDC+08].

One very natural approach to level of detail is to represent the world (or an object) as a union of spheres. Simplification is natural: One replaces several small spheres with a larger sphere that encloses (partially or completely) the small ones. Such representations are easy to translate and rotate, and spheres are simple enough that they're amenable to lots of algorithmic tricks that make the

construction of **sphere trees** [Hub95] relatively easy. This line of work goes back to Badler [BOT79].

One final class of objects deserves mention in the context of level of detail: parametric curves and surfaces. In these cases, the coordinate functions $t \mapsto (x(t), y(t), z(t))$ or $(u, v) \mapsto (x(u, v), y(u, v), z(u, v))$ are real-valued functions, typically defined on an interval or rectangle. As such, they're amenable to Fourier analysis (representation as sums of sines and cosines), and hence to filtering. In specific cases (like B-splines), there are other approaches to simplification, such as replacing a B-spline by its control-point polygon. And in other cases, bases other than the Fourier basis may be appropriate: Finkelstein et al. [CK96] represented level of detail in a B-spline wavelet basis, which allows for both simplification (by eliminating detail beyond a certain scale) and multiscale editing (see Figure 25.19); there are similar approaches for wavelet representations of surfaces [ZSS97] to allow multiscale editing and simplification.

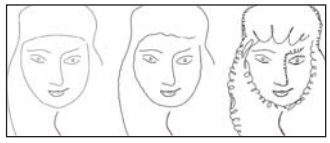


Figure 25.19: A curve, represented in a wavelet basis, consists of large- and small-scale features. The small-scale features—the "character" of the curve—can be edited without affecting the large-scale shape, and vice versa. (Courtesy of Adam Finkelstein and David H. Salesin, ©1994 ACM, Inc. Reprinted by permission.)

25.4.1 Progressive Meshes

From these generalities, we'll now move on to a specific algorithm for mesh simplification, Hoppe's **progressive meshes.** The goal of progressive meshes is to start from a mesh M^n of n nodes, and simplify it to a mesh of $n-1$ nodes by collapsing one edge (thus merging two adjacent nodes), as shown in Figure 25.20. The resultant mesh is denoted M^{n-1}. Successive collapses of edges lead to a sequence of meshes with fewer and fewer nodes, ending at M^1, which consists of a single node. This provides a "continuous" level-of-detail representation for the mesh. Furthermore, the change from M^n to M^{n-1} can be interpolated, in the sense that if we define M^n_t to be a mesh with the topology of M^n, but with geometry modified so that the position u_t of u is $u_t = (1-t)u + tw$, and similarly for v, then M^n_1 consists of exactly the same set of points as M^{n-1}; to convert one to the other requires deleting two area-zero triangles, and renaming some vertices and edges. This interpolation (see Figure 25.21) is called a **geomorph** (for "geometry morph").

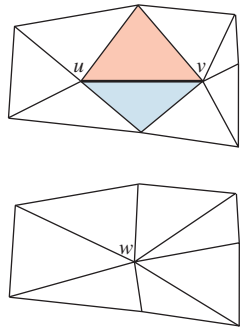


Figure 25.20: The edge from u to v has been collapsed to a single new vertex, which is labeled w. The two triangles that meet this edge have disappeared, and the four non-uv edges have collapsed into two edges. The set of triangles shown is called the neighborhood *of the edge.*

To complete the description of the algorithm, we need to know

1. How to choose a location for the new vertex w
2. How to choose, at each stage, which edge to collapse

For item 2 in the preceding list, Hoppe associates a cost (described below) to each possible edge collapse, and chooses the one with least cost (i.e., he pursues a greedy algorithm).

For item 1 in the list, Hoppe considers three possible locations for w: u, v, and $\frac{1}{2}(u+v)$. Each choice results in a different edge-collapse cost, and he chooses the one with the least cost.

25.4.1.1 Edge-Collapse Costs

To describe the cost of an edge collapse, we first need to describe how to measure how well a given mesh M fits a set of data $X = \{x_i \in \mathbf{R^3} : i = 1, 2, \ldots, N\}$. We'll use a bunch of points from the original mesh M^n as the set X, but for now, you can imagine that M^n is a nice enough mesh that we can just use its collection of vertices as X. For the rest of this discussion, we'll treat the set X as fixed, and not mention it explicitly.

The goodness-of-fit is described by an **energy,** a function that measures how efficiently the mesh M approximates M^n as a sum of several terms,

$$E(M) = E_{\text{dist}}(M) + E_{\text{spring}}(M) + E_{\text{scalar}}(M) + E_{\text{disc}}(M), \tag{25.4}$$

where, for now, we'll combine the last two terms into one,

$$E(M) = E_{\text{dist}}(M) + E_{\text{spring}}(M) + E_{\text{extra}}(M), \tag{25.5}$$

which we can ignore for the time being. The "distance" term in the energy is the sum of the squared distances from each x_i to M: For each x_i, we find the closest point of M (which is itself a minimization problem), square it, and sum up the results. The "spring" term corresponds to placing a spring along each edge of the mesh M; the rest length of the spring is zero, so the total energy is

$$E_{\text{spring}}(M) = \sum_{\substack{(\mathbf{v}_i,\mathbf{v}_j) \text{ an} \\ \text{edge of } M}} \kappa \|\mathbf{v}_i - \mathbf{v}_j\|, \tag{25.6}$$

where κ is a spring constant that we'll describe in a moment. The idea is to adjust the vertex locations for the mesh M to minimize this energy, thus finding a mesh that fits the data (X) well, while not having excessively long edges. Figure 25.22 shows why the spring-energy term is needed.

The cost of an edge collapse from a mesh M to a mesh M' is determined by computing

$$\Delta E = E(M') - E(M), \tag{25.7}$$

which will generally be positive (it's harder to fit the data with fewer vertices!). Of course, this cost depends on knowing the vertex locations for M'. Since the change from M to M' is a single edge collapse, the new knowledge amounts to just the location of the vertex corresponding to the collapsed edge, since all other vertices remain unchanged. As we said, Hoppe restricts the possible new vertex locations for the edge between $\mathbf{v}_1$ and $\mathbf{v}_2$ to three choices: $\mathbf{v}_1$, $\mathbf{v}_2$, and their average. To compute the distance plus spring cost, he uses an iterative approach, which we sketch in Listing 25.1.

Listing 25.1: Finding the optimal placement of a vertex for a collapsed edge.

```
1  Input: a mesh M and an edge v_s v_t of M to collapse
2  Output: the optimal position for v'_s, the position of
3   vertex s after the collapse
4
5  E ← ∞
6  repeat until change in energy is small:
7     Compute, for each x_i ∈ X, the closest location b_i on the
8        mesh M
9     Find the optimal location for location v'_s by solving a
10       sparse least-squares problem, using the computed locations
11       {b_i : i = 1,...K} to compute E_dist
12    Compute the energy E' of the resulting mesh
```

The only difficulty is that the location $\mathbf{b}_i$ is on the mesh M, and it must be transferred to the mesh M'; since the two meshes are mostly identical, this is generally easy. But if $\mathbf{b}_i$ lies in a triangle containing $\mathbf{v}_s$ or $\mathbf{v}_t$, we compute its

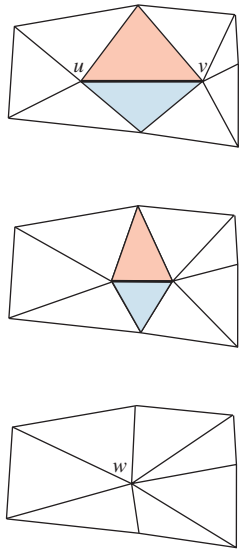


Figure 25.21: An edge-collapse can be made gradually, by interpolating from the original positions of u and v part of the way toward the final position w.

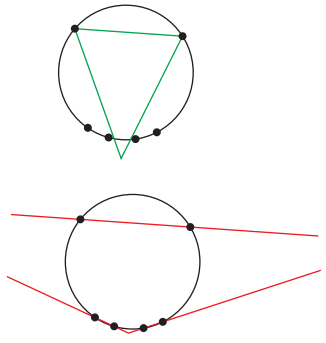


Figure 25.22: The value of "spring" energy: If we try to fit the six data points on a circle (marked with small dots) using a triangle with short edges, the green triangle (top) is a reasonably good solution. If we remove the short-edge constraint, the red triangle (bottom) is a "perfect" fit, even though it violates our expectations.

barycentric coordinates in that triangle and use these to place it in the mesh M', treating the vertex $\mathbf{v}'_s$ as the location for both $\mathbf{v}_s$ and $\mathbf{v}_t$ in M'.

We can now outline almost the entire progressive meshes algorithm (see Listing 25.2).

Listing 25.2: The core of Hoppe's progressive meshes algorithm.

```
Input: a mesh M = M^n with n vertices, and a set of points X distributed on M.
Output: A sequence of meshes M^n, M^{n-1}, ..., M^0, where M^k has k vertices, approximating the
    original mesh M.

E ← ∞
for each edge e of M:
  compute the cost of optimal collapse of edge
  insert (edge, cost) into a priority queue, Q
for k = n downto 1:
  extract the lowest-cost edge e from Q
  collapse e in M^k to get M^{k-1}
  for each edge e' that meets e:
    compute a new optimal collapse of e' and its cost
    update the priority of e' in Q to the new cost
```

One final detail remains: the spring constant, κ. Hoppe defines the ratio r of the number of points in the neighborhood of the edge to the number of faces in the neighborhood, and then sets $\kappa = 10^{-2}$ if $r < 4$, $\kappa = 10^{-4}$ if $4 \leq r < 8$, and $\kappa = 10^{-8}$ if $r \geq 8$.

The algorithm, as described so far, shows how to simplify a mesh while considering its geometric structure. But meshes often have other attributes, specified on a per-vertex level, such as color, material, etc. Some of these attributes, like color, lie in continuous spaces, where it makes sense to measure differences and to compute averages. Others, like material, can be considered to be more discrete—you're either on the metal part of the engagement ring or on the diamond; there's no halfway point between these materials. The first group consists of the **scalar** attributes and the second group consists of the **discrete attributes.**

To assign a scalar attribute to a vertex that results from an edge collapse, we try to pick a value with the property that for each point $\mathbf{x}_i \in X$ on the original surface, with corresponding point $\mathbf{b}_i$ on the simplified mesh, the scalar value $s(\mathbf{x}_i)$ and the corresponding value $s(\mathbf{b}_i)$ (which may have to be interpolated from values at nearby vertices) are as close as possible. More explicitly, we choose a scalar value at the new vertex to minimize

$$E_{\text{scalar}}(V) = \sum_i \|s(\mathbf{x}_i) - s(\mathbf{b}_i)\|^2. \tag{25.8}$$

Even scalar attributes can require special handling: Consider a cube in which each face is given a different color. At each vertex, there are three color attributes, one for each face corner at the vertex, instead of just one; a complete implementation of the algorithm must address this "distinguished-corner" situation as well.

For discrete attributes like material, Hoppe characterizes an edge as **sharp** if it's a boundary edge, its two adjacent edges have different discrete attributes, or its adjacent corners have different scalar attributes, in the sense of the preceding paragraph. The set of sharp edges on the mesh form "discontinuity" curves between regions of constant discrete attributes (or between faces that meet with distinguished corners). Hoppe then either disallows or penalizes the collapse of any

edge that would modify the topology of the discontinuity curves, depending on the application. This penalty cost, if included, is the final term E_{disc} in the energy formulation. This treatment of attributes is an instance of the Meaning principle: the difference of discrete attributes in adjacent triangles gives a meaning to the edge between them that guides the computation.

25.4.2 Other Mesh Simplification Approaches

While Hoppe's approach to mesh simplification makes sense if you are starting from a mesh, there are situations where radically different approaches make sense. For instance, if you have a spline surface that you've triangulated at some resolution, and you want a simpler triangulation, it makes sense to go back to the spline-tessellation procedure and invoke it with different parameters. Part of the evolving landscape of GPU architectures is the development of **tessellation shaders,** or small pieces of code, run on the GPU, that take some description of a shape and produce from it a tessellation—a division into polygons—of the shape, with the tessellation typically being governed by one or two parameters that determine the density of triangles produced.

25.5 Mesh Applications 1: Marching Cubes, Mesh Repair, and Mesh Improvement

We will now illustrate some typical uses of meshes. The first is the marching cubes algorithm, used to extract level-set surfaces from volumetric data. The next is an approach to mesh repair—filling in holes and cracks, etc.—based on a variant of marching cubes. The third is an algorithm for mesh "improvement," in which the interior structure of a mesh, primarily the shapes of the triangles, is improved while keeping the overall mesh geometry unchanged.

25.5.1 Marching Cubes Variants

So far we've discussed triangle meshes, reasoning that they're a dominant modeling technology. But what if you have a model presented in some other form, like an implicit model? One standard form of implicit model is the uniformly sampled grid of densities; imaging modalities like nuclear magnetic resonance often produce such data, where the number associated to each grid cell is proportional to the amount of some material within that cell. An isosurface of this data can represent the boundary between tissue and air, or between soft tissue and bone, etc. Extracting representations of such isosurfaces is one step in rendering them: We can take the resultant polygonal mesh and hand it to a polygon-rendering pipeline. The marching cubes algorithm presented in Section 24.6 does exactly this.

Marching cubes can be generalized in many ways. We'll discuss two of these in the 2D case, the 3D case being exactly analogous. If we have data values at the vertices of a square grid, linear interpolation allows us to estimate the location of zeroes along each edge. The "marching squares" algorithm fills in these estimated locations of zeroes on a cell's boundary with line segments in the interior of the cell; these line segments, taken together, constitute an estimate of the

zero-level set for the function whose values we know at the grid points, as shown in Figure 25.23.

In doing so, however, we are ignoring additional data: At each estimated zero point, we can also make an estimate of the *gradient* of the function (see Figure 25.24) (or perhaps get the gradient directly as part of gathering the original data) and use these in estimating the shape of the level set.

This can be called an "Hermite" version (see Chapter 22) of marching squares. The **extended marching cubes** algorithm [KBSS01] uses this Hermite data to determine how the interior of a square appears: If the normals for the data in a square are similar, the algorithm defaults to standard marching squares. But if the point-normal pairs for a square (say $(X_1, \mathbf{n}_1)$ and $(X_2, \mathbf{n}_2)$) are inconsistent (if $\mathbf{n}_1$ and $\mathbf{n}_2$ are not nearly parallel enough), then the square is treated differently: A new vertex, X, is placed in the square in such a way that it minimizes a **quadratic error function,**

$$X = \arg\min \sum_i ((X - X_i) \cdot \mathbf{n}_i)^2, \tag{25.9}$$

as shown in Figure 25.25; in other words, X seeks to lie on the lines determined both by $(X_1, \mathbf{n}_1)$ and $(X_2, \mathbf{n}_2)$. (In 2D, this is trivial; in 3D, where there may be more contributing point-normal pairs, the minimization can be more complicated.) The newly inserted point is connected (with a pair of edges in 2D, or with a triangle fan based at X in 3D) to the zero-set on the boundary of the grid cell.

There are two small problems.

1. If the normal vectors are sufficiently antiparallel, it's possible that the minimizer X lies outside the square, and X must be adjusted to lie within it.
2. In the 3D case, if there are "extra points" X_1 and X_2 in two adjacent cells, we perform an edge flip on the contour edge that lies in the grid face between the two cells so that this edge now goes from X_1 to X_2.

Inline Exercise 25.8: ◇ Argue that the "bad minimizer" problem above is the result of *aliasing*. What sampling is going on? Where are the too-high frequencies?

In estimating (at least in special cases) a surface point in the interior of a cell, the extended marching cubes algorithm suggests a different approach: If we had a surface point in adjacent cells, we could join these with edges (and faces, in 3D), using the points on grid edges as guides. Such an approach is called **dual contouring.** Ju et al. [JLSW02] developed a scheme for dual contouring of Hermite data, which involves two steps.

1. For each cell with differently signed vertices (i.e., not all positive or all negative), generate a mesh vertex within the cell by minimizing a quadratic error function.
2. For each edge with differently signed vertices, generate an edge (for 2D) or a quad (for 3D) connecting the minimizing vertices in the adjacent cells.

This algorithm has the pleasant characteristic that all cells are treated identically—there's no threshold on when "normal vectors are close enough"—but there are subtleties: Again, the minimizing point for a quadratic error function

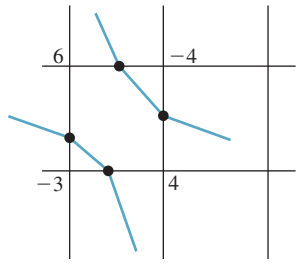


Figure 25.23: Knowing function values at grid points, we can estimate zero-crossings (black dots) by linear interpolation, and then connect the dots, in each cell, to estimate the level curve at level zero.

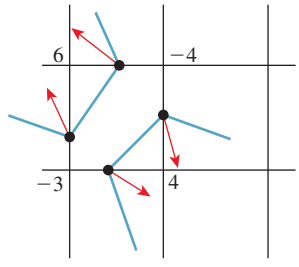


Figure 25.24: We can estimate gradients at the zero-crossings as well. Fitting a surface to the point-and-direction data gives a different estimate of the level-set.

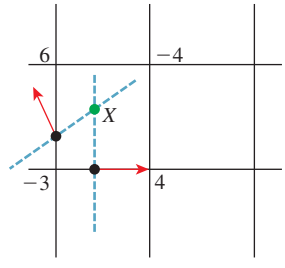


Figure 25.25: The normals at the two zero-crossings determine two lines, which intersect at a new point X.

may lie outside the cell, and furthermore, in areas where the surface is nearly planar the Hermite data for a cell may create a quadratic error function that is nearly degenerate (i.e., one whose zero-set is a whole line or plane) so that finding a minimizer is numerically unstable. Their work addresses this by careful numerical analysis, although their solution requires an ad hoc constant.

25.5.2 Mesh Repair

A mesh can be "broken" in all sorts of ways. Consider something as simple as a (hollow) triangle ABC in the plane. If we consider the edges as oriented as AB, BC, and CA, then the oriented boundary of the triangle, where the endpoint of each edge is counted as a $+1$ and the starting point as a -1, is empty. But if the edges are oriented as AB, BC, and AC, then the oriented boundary is $2C - 2A$; if we were computing the boundary as a way to check whether the triangle was watertight, we'd find that it wasn't. And if we used the oriented edges to compute normal vectors, the misoriented edge AC would cause problems with inside/outside determination. In a mesh whose representation contains an edge table, an algorithm closely related to the one for consistently orienting faces can be used to repair the edge table.

Often meshes created with rendering speed in mind have characteristics that generate problems, like T-junctions or nonwatertightness. Sometimes those created with the best intentions, like the Utah teapot, have problems. (The original teapot had no bottom!) So, while nice models are best to work with, we often encounter **polygon soup**—a collection of polygons that *nearly* form a nice surfacelike mesh—and we want to be able to make the most of what we've got. One example of this is in scanning, where a scanner may produce a great many points on a surface, and may even organize those points into triangles, but the triangles gathered from different views of the model may be inconsistent because of problems with registration of the views, or changing occlusion, etc. Such triangle soups need to be cleaned up to form consistent models for use in the rest of the graphics pipeline.

Ju [Ju04] has used his dual contouring for Hermite data to address this last problem. The ideas in the approach are simple, and the results are particularly attractive, so we sketch the method briefly here. The input is a collection of polygons; the output is a surface mesh that's consistent, in some way, with the input polygons. Figure 25.26 shows the process (in 2D) in the case where the input mesh *is already* a closed polygon; in this case, the original mesh is reconstructed almost exactly, although it has been translated by an amount smaller than the grid cell.

The steps in the process are as follows.

1. The polygon soup is embedded in a uniformly spaced grid, and grid edges that intersect any polygon are marked as intersection edges. The cells containing such edges are stored in an oct tree. The choice of a fixed grid size implicitly represents a choice about aliasing: An empty polygon soup, and soup consisting of a single tetrahedron that fits entirely within one grid cell, will produce the same (empty) output. Thus, these two "signals" end up as aliases of each other.
2. The intersection edges are used to generate signs at the grid points in such a way that each intersection edge exhibits a sign change. This may be

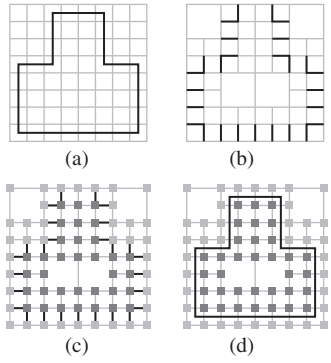


Figure 25.26: (Following [Ju04], Figure 3.) Ju's model-repair method. (a) A model, embedded in a fine grid. (b) The grid edges that intersect the model, stored in an oct tree. Each cell touches an even number of such edges. (c) Signs at grid points (indicated by light or dark shading) generated from the set of intersection edges. (d) The model reconstructed by contouring the sign data.

impossible for certain inputs, as shown in Figure 25.27. These problems arise because of "boundary" in the input data.

Ju shows a method for "filling in" boundaries like this to create a set of intersection edges that *can* be given consistent signs. The filling-in approach generates fairly smooth completions of curves (2D) or surfaces (3D).

3. From the sign data (which can be extended to the entire grid), we can use marching cubes, or extended marching cubes or dual contouring, if we have normal data, to extract a consistent closed surface.

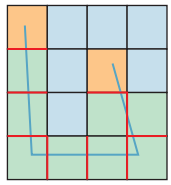


Figure 25.27: The U-shaped polygon soup generates several edges, but there's no way to consistently assign signs to the grid cells so that each intersection edge exhibits a sign change. Cells with good labelings—most of the ones meeting the polyline—are in green. The problem arises at cells (in orange at the ends of the polyline) with an odd number of intersection edges.

The method is not perfect. It can produce models in which the solids bounded by the output mesh have small holes (like those in Swiss cheese) or handles, and in areas where there are many bits of boundary in the original mesh the filling-in of the boundary may cause visually unattractive results. Nonetheless, the guaranteed topological consistency, and the high speed of the algorithm (due largely to the use of oct trees), make it an excellent starting point for any mesh-repair process.

25.5.3 Differential or Laplacian Coordinates

Just as derivatives are a central component of smooth signal processing and differences are critical in discrete signal processing, it's natural to seek something similar in the case of mesh signal processing. If we have a signal s defined on the vertices of a mesh, the differences $s(w) - s(v)$, where v and w are adjacent vertices, provide the analog to the differences $s(t+1) - s(t)$, $t \in \mathbf{Z}$, for a discrete signal.

Second derivatives also arise in an important way: In Fourier analysis of a signal on an interval, we write signals as sums of sines and cosines, which are eigenfunctions of the second-derivative operator on the space of all signals. But considering second derivatives in a more concrete way, if we have a discrete signal

$$s : \mathbf{Z} \to \mathbf{R} : t \mapsto s(t), \tag{25.10}$$

and we tell you $s(0)$ and $s'(0)$, and $s''(t)$ for every t (using the "derivative" notion for what are really differences: $s'(t)$ denotes $s(t+1) - s(t)$, and $s''(t)$ denotes $s(t+1) - 2s(t) + s(t-1)$), then you can reconstruct $s(t)$ for every t: You use $s'(0)$ to reconstruct $s(1)$, and use $s''(1)$, together with your knowledge of $s(0)$ and $s(1)$, to reconstruct $s(2)$, and then continue onward.

Inline Exercise 25.9: Carry out the computation just described. Start with $s(0) = 4$, $s'(0) = 1$, and $s''(1) = -1, s''(2) = 0, s''(3) = -1$, and figure out $s(1)$, $s(2)$, $s(3)$, and $s(4)$.

Recording the value $s(0)$ and derivative $s'(0)$, together with all the second-derivative values, thus provides an alternative representation of the signal. This representation has the advantage that if we want to add a constant to the signal, we can do so by only changing $s(0)$ and leaving the rest of the data untouched.

Inline Exercise 25.10: Carry out the same computation as in the preceding exercise, but this time start with $s(0) = 2$; confirm that all other values shift by -2.

Similarly, we can add a linear variation to the signal (a steady increase or decrease in value) by changing just $s'(0)$. Thus, the second-derivative information captures the aspects of the signal that are unaffected by translation or shearing of the signal.

The analog, for meshes, is provided by the **mesh Laplacians.** Suppose that we have a mesh, and at each vertex we have a real number, which we'll denote $s(v)$ in analogy with the one-dimensional discrete-signal case. There's no longer a notion of the previous or next signal value, but there is still a notion of adjacent values. Letting $N(v)$ denote the 1-ring consisting of vertices adjacent to v, and $|N(v)|$ denote the size of this set, we define the Laplacian of s at v to be

$$L(s)(v) = C \sum_{w \in N(v)} (s(v) - s(w)), \tag{25.11}$$

where the constant C is unimportant for now. We can bring the constant $s(v)$ outside the sum, to get

$$L(s)(v) = C|N(v)|s(v) - \sum_{w \in N(v)} s(w) = C'(s(v) - \frac{1}{|N(v)|} \sum_{w \in N(v)} s(w)), \tag{25.12}$$

where we've absorbed the number $|N(v)|$ of neighbors of v into the constant C to make C'. In this form, we see the Laplacian expresses the difference between the signal value at the vertex v and the average of the signal values at the neighbors of v.

To be clear: The Laplacian is a function from "signals on the mesh" to "signals on the mesh." Thus, if s is a signal, so is $L(s)$, and $L(s)(v)$ denotes the value of that signal at a particular vertex.

In analogy with the 1D situation, if you knew the value of s at some vertex v_0 and at all but one of its neighbors, and you knew the Laplacian of s at every vertex, then you could compute the value of s at the last neighbor. And that might give you enough information to compute the value of s at another vertex, etc.

There's a difference from the discrete-signal situation, however: The Laplacian values are not all independent.

Inline Exercise 25.11: Draw a tetrahedron, and write the numbers $1, 3, 0$, and 5 at the four vertices, thus defining a "signal" s on the tetrahedron. Compute the Laplacian $L(s)(v)$ at each of these vertices, using $C = 1$ as the constant. What do you notice about the sum of these values?

Thus, although the Laplacian of a signal once again represents the part that's independent of the addition of a constant at every vertex, and some other kind of alteration similar to shearing in the 1D discrete-signal case, it's no longer the case that an arbitrary set of values at vertices $\{h(v) : v \in V\}$ can be treated as the Laplacian of some signal and "integrated" to recover the original signal. The

usual solution is to ask for a signal s with the property that $L(s)(v) - h(v)$ is as small as possible, typically by minimizing a sum of squares

$$E(s) = \sum_{v \in V} (L(s)(v) - h(v))^2. \tag{25.13}$$

Since the Laplacian is invariant under the addition of a constant to the signal, this minimization must typically be performed with one or more additional constraints, such as the value $s(v_i)$ at some small number k of selected vertices $v_1, \ldots, v_k$.

While we tend to have many functions defined on meshes—it's common, for example, to evaluate some lighting model at each vertex of a mesh, and interpolate over edges and triangles—the function to which the Laplacian is most often applied is the one that returns, for each vertex, the coordinates of its location. For instance, we can regard the x-coordinate of each vertex as providing a signal value at that vertex. The same goes for the y- and z-coordinates. If we regard $\mathbf{x} : V \to \mathbf{R}^3$ as the vector-valued function that takes each vertex to its xyz-coordinates, then it's common to compute

$$\boldsymbol{\delta}(v) = L(\mathbf{x})(v) \text{ for } v \in V. \tag{25.14}$$

These vectors, one per vertex, are sometimes called the **Laplacian coordinates** or **differential coordinates** for the mesh.

"Laplacian coordinates" is really a misnomer; a coordinate system should have the property that no two distinct points have the same coordinates (although in some cases, like polar "coordinates," we allow a single point to be given multiple coordinates). But it's easy to see that for a regular triangulation of a plane, the Laplacian coordinates at every vertex are zero, so any two planar parts of this mesh end up with the same "coordinates." Perhaps "coordinate Laplacian" or "coordinate differentials" would be a better term, but "Laplacian coordinates" and "differential coordinates" are well established already.

Laplacian coordinates have a few obvious properties. First, they are invariant with respect to translation, that is, if M' is a translated version of the mesh M, then the Laplacian coordinates at corresponding vertices are identical. Second, they are equivariant with respect to linear transformations, that is, if M' is the mesh resulting from applying a linear transformation T to each vertex of M (e.g., rotating M 30° in the xy-plane), then the Laplacian coordinates at a vertex v' in M' can be computed from those at the corresponding vertex v by applying T. These facts, taken together, can be summarized by saying that Laplacian coordinates are equivariant with respect to affine transformations of meshes, as long as we remember that the action of an affine transformation on a vector ends up ignoring translations.

In summary: Laplacian coordinates on a mesh provide an affine-transformation-equivariant description of the geometry of the mesh. A mesh can be reconstructed from its Laplacian coordinates together with a small number of known mesh locations. And any vector-valued function of the vertices of a mesh can play the role of Laplacian coordinates if we reconstruct by solving a least-squares problem rather than seeking an exact solution.

25.5.4 An Application of Laplacian Coordinates

Nealen et al. [NISA06] describe an approach in which a mesh is "optimized" by adjusting vertex positions, but retaining the mesh connectivity. The technique is simple, and its good and bad features are self-evident.

The idea is to adjust the vertex locations to satisfy two goals: The first is that each vertex be as close to its original position as possible (with some vertices weighted more than others); the second is that the mesh Laplacian at a vertex, after adjustment, be as similar as possible to the nontangential part of the preadjustment mesh Laplacian. Since the mesh Laplacian represents the difference between a vertex and the average of its neighbors, this makes each vertex "want" to be the average of the neighbors, except for perhaps being displaced from the plane of the neighbors. (This description assumes that the neighbors actually lie more or less in a single plane, of course.)

Clearly the two goals—that vertices move toward their neighbor averages and that vertices not move at all—are in tension with each other. By adjusting the relative weights, we can arrange for greater shape preservation or greater mesh uniformity. Nealen et al. suggest some strategies for choosing weights that are widely effective.

In contrast to the usual vertex-and-face-table representation, we start with a mesh represented as a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, with vertex set $\mathbf{V}$ and edge set $\mathbf{E}$. We consider $\mathbf{V}$ as an $n \times 3$ array whose ith row contains the x-, y-, and z-coordinates of the ith vertex, which we also store in a 3×1 column vector $\mathbf{v}_i$ so that $\mathbf{V} = [\mathbf{v}_1, \ldots, \mathbf{v}_n]^{\mathbf{T}}$.

We'll compute the Laplacian coordinates, at vertex i by the rule

$$\boldsymbol{\delta}_i = \sum_{(i,j)\in E} w_{ij}(\mathbf{v}_j - \mathbf{v}_i) \tag{25.15}$$

$$= \left[\sum_{(i,j)\in E} w_{ij}\mathbf{v}_j\right] - \mathbf{v}_i \tag{25.16}$$

where for each i the weights w_{ij} sum to one, and may be chosen to be uniform, that is,

$$w_{ij}^u = 1/|\{j : (i,j) \in E\}|, \tag{25.17}$$

so that each edge arriving at $\mathbf{v}_i$ gets equal weight, or by the **cotangent rule,** in which we set

$$w_{ij} = \cot\alpha + \cot\beta \tag{25.18}$$

where α and β are the internal angles of the vertices on either side of $\mathbf{v}_j$ in the ring around $\mathbf{v}_i$ (see Figure 25.28).[1] We can then define

$$w_{ij}^c = \frac{w_{ij}}{\sum_{(i,k)\in E} w_{ik}} \tag{25.19}$$

so that the sum $\sum_j w_{ij}^c$ is 1.

If $\mathbf{v}_i$ and its neighbors all lie in a plane, then the uniform Laplacian points from $\mathbf{v}_i$ toward the average of the neighbor vertices, while the cotangent Laplacian is the zero vector. The two resultant Laplacians will be decorated with superscripts, as in $\boldsymbol{\delta}_i^u$, to indicate their nature.

Our goal is to find new vertex positions $\mathbf{v}_i'$ $(i = 1, \ldots, n)$ that are both (a) near the old positions, and (b) arranged so that their Laplacians are similar,

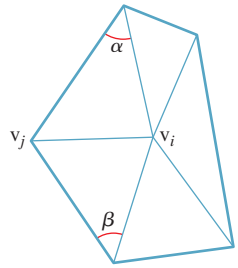


Figure 25.28: The vertex $\mathbf{v}_i$ is surrounded by a ring of vertices, one of which is $\mathbf{v}_j$; the angles on either side of $\mathbf{v}_j$ are called α and β. The cotangents of these angles are used in defining the weight of $\mathbf{v}_j$ in the cotangent Laplacian vector at $\mathbf{v}_i$.

1. The rationale for the cotangent rule is presented in the web material for this chapter.

in some way, to the old Laplacians. The first condition can be expressed by writing

$$\mathbf{W}_p\mathbf{V}' = \mathbf{W}_p\mathbf{V}, \tag{25.20}$$

where $\mathbf{W}_p$ is a diagonal matrix of weights, which gives us control over which position constraints are most important to us. Clearly this system can be solved by letting $\mathbf{V}' = \mathbf{V}$, but we'll be adding further constraints in a moment.

Inline Exercise 25.12: Since there are n vertex positions $\mathbf{v}_i$, each in 3-space, the matrix $\mathbf{V}$ is $n \times 3$. What size is each of the other matrices in Equation 25.20?

The second condition, on the Laplacian, can be written in the form

$$\mathbf{W}_L\mathbf{L}\mathbf{V}' = \mathbf{W}_L\mathbf{F}, \tag{25.21}$$

where again we've included a diagonal weighting matrix $\mathbf{W}_L$, and where $\mathbf{L}$ is the matrix of some Laplacian coordinate transform (i.e., the first row of $\mathbf{L}\mathbf{V}$ is the Laplacian at vertex $\mathbf{v}_1$, etc.). The matrix $\mathbf{F}$ is a target for the Laplacians. Taken together, in block matrix form, we'll be solving

$$\begin{bmatrix} c\mathbf{W}_L\mathbf{L} \\ \mathbf{W}_p \end{bmatrix} \mathbf{V}' = \begin{bmatrix} c\mathbf{W}_L\mathbf{F} \\ \mathbf{W}_p\mathbf{V} \end{bmatrix}. \tag{25.22}$$

Nealen et al. observe that if we set $\mathbf{L} = \mathbf{L}_{\mathrm{u}}$ and $\mathbf{F}$ contains the cotangent Laplacian coordinates of all vertices, the resultant mesh preserves the details of the original but the shapes of individual triangles are improved. If we set $\mathbf{W}$ to the identity, all vertices are allowed to move equally and the triangle shapes are slightly improved. If, on the other hand, we set the weight for a vertex to be proportional to the mean curvature at that vertex, then vertices at highly curved points will remain fixed, while others move. The problem with this is that often there are a few vertices with *very* high curvatures, and so if we map curvatures to weights linearly, only a few vertices get at all constrained. Nealen et al. suggest an alternative: They compute

$$C(\kappa), \tag{25.23}$$

the fraction of all vertices whose curvatures are no more than κ, and assign to each vertex of curvature κ a weight proportional to $C(\kappa)$. The weights are stored in a matrix $\mathbf{W}_L$. The constant of proportionality is a tuning parameter.

The result is a system that manages to improve triangle shape while maintaining the mesh features at high-curvature points.

On the other hand, if we set $\mathbf{L} = \mathbf{L}_{\mathrm{u}}$ and $\mathbf{F} = 0$, then we actually *smooth* the mesh, removing some amount of noise from the shape. Nealen et al. discuss other variants of this approach.

Inline Exercise 25.13: We have not discussed how to set the positional weights, $\mathbf{W}_p$. Can you think of any points in a typical object (e.g., a video-game character) for which large weights might be appropriate?

In both the triangle shape optimization and mesh smoothing methods, we need to solve the system of equations in Equation 25.22. This consists of a $2n \times n$ matrix

multiplied by an unknown $n \times 3$ matrix to get a $2n \times 3$ matrix. We can solve this problem for one column of unknowns at a time. That is we can solve

$$\mathbf{AX} = \mathbf{B} \tag{25.24}$$

by solving

$$\mathbf{AX}_i = \mathbf{B}_i \tag{25.25}$$

where $\mathbf{X}_i$ indicates the ith column of $\mathbf{X}$, for $i = 1, 2, 3$. To solve, we can compute an LU decomposition of $\mathbf{A}$ [Pre95]. Fortunately, this single decomposition can then be used to find each column of $\mathbf{X}$.

25.6 Mesh Applications 2: Deformation Transfer and Triangle-Order Optimization

We conclude this chapter with two more advanced applications related to meshes. The first is **deformation transfer,** in which two meshes of objects with some relationship (e.g., two quadrupeds) are matched at a few key points, after which any deformation of the first mesh can be automatically transferred to the other. The second is an approach for restructuring a mesh's face table so that during rendering the mesh will tend to use a GPU most efficiently.

25.6.1 Deformation Transfer

Suppose that using a motion-capture system, we have captured the varying position of a human actor over time; in other words, we have a mesh M (the *source* mesh) with a fixed connectivity (i.e., a fixed set of triangles, represented as vertex-index triples), and a sequence V^i, $(i = 0, 1, \ldots)$ of positions for the vertices of the mesh (i.e., V^0 is the set of vertex positions at time 0, perhaps representing the actor standing upright at rest, V^1 is the set of vertex positions at time 1, etc.). We'd like to transfer this motion sequence to a different mesh (the *target*)—a video-game character, for instance—but one that may not have a realistic human form. Or perhaps we have a recorded sequence of positions for a horse, but we want to apply them to a camel, which was unavailable for motion capture. We can consider the difference between V^0 and V^k as a **deformation** of the known model; our desire is then to transfer this deformation to the target mesh M', with a possibly different set of triangles and vertex locations W^0, to get a new set of vertex locations W^k. We'll follow the approach of Sumner and Popović [SP04] (see Figure 25.29).

Because we are considering only deformations, we need only look at $k = 1$ and apply the same technique to each subsequent value of k. So we can relabel things a bit for simplicity: We'll use V_i to indicate the initial position of vertex i in the source mesh (i.e., its position in V^0), and $\bar{V}_i$ to indicate its deformed position (i.e., its position in V^1). For notational simplicity, it's helpful if everything we consider is a vector rather than a point. We therefore pick an "origin" O_M for M, and express each vertex as an offset from this origin:

$$\mathbf{v}_i = V_i - O_M. \tag{25.26}$$

We do the same for $\bar{V}$, that is,

$$\bar{\mathbf{v}}_i = \bar{V}_i - O_M, \tag{25.27}$$

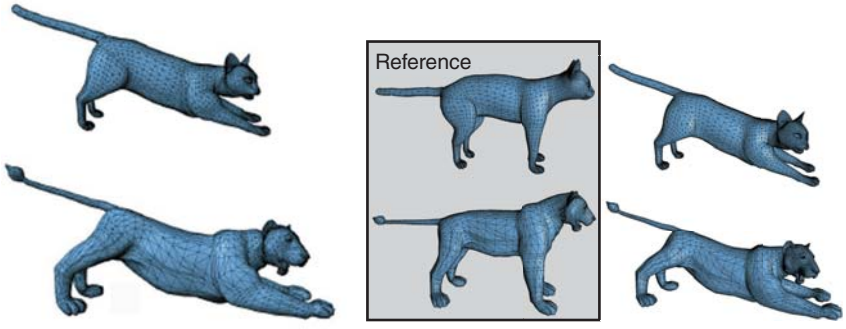


Figure 25.29: We start with a source and a target mesh, shown at left, a triangle-by-triangle correspondence between them (in this case, the correspondence is the fairly obvious one), and a deformation of the source mesh, shown at the top right. The deformation transfer algorithm provides a transformation of the target mesh that's analogous to the deformation of the source mesh. (Courtesy of Robert Sumner and Jovan Popović.)

and similarly for M', although we'll choose the origin $O_{M'}$ independently. (It's a good idea to use something like the center of mass of M as the origin for M, and similarly for M', to avoid roundoff errors in later numerical computations.)

We'll similarly use $\mathbf{w}_i$ and $\bar{\mathbf{w}}_i$ for the (vector) position of the ith target vertex before and after deformation. We'll assume that we're given $\mathbf{v}_i$, $\bar{\mathbf{v}}_i$, and $\mathbf{w}_i$ for all i, and we wish to find $\bar{\mathbf{w}}_i$ for all i.

To make the connection between the deformations of the source mesh M and the target mesh M', we need a **correspondence,** C, between them. This is provided, in the Sumner-Popović formulation, by a collection of pairs $C = \{(s_i, t_i) \in \mathbf{Z} \times \mathbf{Z} : i = 1, 2, \ldots, c\}$ between triangles. A pair (s_i, t_i) indicates that the target triangle with index t_i should deform similarly to the source triangle with index s_i. The set C is a **relation** on triangle indices: It may specify that triangle 7 in M' is to deform similarly to both triangles 2 and 96 in M (in which case the pairs $(2, 7)$ and $(96, 7)$ would both be in C), or that triangles 11 and 12 in M' should both deform like triangle 4 in M, in which case C would contain both $(4, 11)$ and $(4, 12)$. It's not required that every triangle index of M appear as the first element of some pair in C, nor that every index of M' appear as the second element of a pair. Nonetheless, it may be easiest to imagine C as being nearly a one-to-one correspondence, in which a triangle on the horse's head corresponds to a triangle on the camel's head and a triangle on the horse's left front foot corresponds to a triangle on the camel's left front foot, etc. The problem of **building** or describing such a correspondence in the first place is a separate one; it's possible to try to algorithmically guess correspondences between parts, etc., but it's probably easiest to have a user indicate correspondences between a few dozen key points, and then use some kind of breadth-first-search-followed-by-relaxation approach to "grow" the correspondence outward from these key points.

We'll now set about describing deformation transfer as an optimization problem. Before we write the optimization, however, we need one more enhancement of the meshes.

Suppose we have a triangle with vertices $\mathbf{v}_1, \mathbf{v}_2$, and $\mathbf{v}_3$, which are to be sent to another triangle with vertices $\bar{\mathbf{v}}_1, \bar{\mathbf{v}}_2$, and $\bar{\mathbf{v}}_3$. It's natural to think of the transformation from one to the other in terms of the affine transformations we use all the

time in graphics: some sort of translation, and an associated linear transformation. The problem is that the two triangles each lie in a *plane,* and there are infinitely many affine transformations taking one triangle (and its plane) to the other (and its plane): The off-plane part of the transformation is completely unconstrained.

Inline Exercise 25.14: Examine a 2D analog: Find an affine transformation $\mathbf{T}$ from $\mathbf{R^2}$ to $\mathbf{R^2}$ that takes the line segment from $(0,0)$ to $(1,0)$ on the x-axis into the line segment from $(0,1)$ to $(0,2)$ on the y-axis. Now compose your transformation with the transformation $\mathbf{S} : \mathbf{R^2} \to \mathbf{R^2} : (x,y) \mapsto (x+3y, y)$, that is, form $\mathbf{R} = \mathbf{T} \circ \mathbf{S}$, and show that $\mathbf{R}$ transforms the segments exactly the same way that $\mathbf{T}$ did.

We therefore add a new vertex $\mathbf{v}_4$ that's offset one unit along the normal to the triangle defined by $\mathbf{v}_1, \mathbf{v}_2$, and $\mathbf{v}_3$,

$$\mathbf{v}_4 = \mathbf{v}_1 + \frac{(\mathbf{v}_2 - \mathbf{v}_1)\times(\mathbf{v}_3 - \mathbf{v}_1)}{\sqrt{\|(\mathbf{v}_2 - \mathbf{v}_1)\times(\mathbf{v}_3 - \mathbf{v}_1)\|}}, \tag{25.28}$$

and a corresponding new vertex to M'. Now there's a *unique* affine transformation taking $\mathbf{v}_1, \ldots, \mathbf{v}_4$ to $\bar{\mathbf{v}}_1, \ldots, \bar{\mathbf{v}}_4$. Writing the transformation as a combination of a translation with a linear map, we see that the linear map $\mathbf{Q}$ must send the vectors $V_i - V_1$ to $\bar{\mathbf{v}}_i - \bar{\mathbf{v}}_1$ for $i = 2, 3, 4$. Writing

$$\mathbf{V} = \begin{bmatrix} \mathbf{v}_2 - \mathbf{v}_1 & \mathbf{v}_3 - \mathbf{v}_1 & \mathbf{v}_4 - \mathbf{v}_1 \end{bmatrix} \text{ and} \tag{25.29}$$

$$\bar{\mathbf{V}} = \begin{bmatrix} \bar{\mathbf{v}}_2 - \bar{\mathbf{v}}_1 & \bar{\mathbf{v}}_3 - \bar{\mathbf{v}}_1 & \bar{\mathbf{v}}_4 - \bar{\mathbf{v}}_1 \end{bmatrix}, \tag{25.30}$$

we have $\mathbf{S} = \bar{\mathbf{V}}\mathbf{V}^{-1}$; we can thus compute the translation d as

$$d = \bar{\mathbf{v}}_1 - \mathbf{S}(\mathbf{v}_1 - O), \tag{25.31}$$

where O is the origin of 3-space.

Notice that we have added one new vertex to the original and deformed meshes *for each triangle*. So our starting point becomes

- An original mesh M, with an enlarged vertex set that we'll still denote with the same symbols, $\{\mathbf{v}_i\}$, and with each "triangle" associated to *four* vertices
- A deformed mesh $\bar{M}$, with a corresponding enlarged vertex set
- A second mesh, M', with an enlarged vertex set $\{\mathbf{w}_i\}$
- A correspondence C between triangles of M and M'
- For each triangle t of M, an affine transformation $\mathbf{v} \mapsto \mathbf{S}_t\mathbf{v} + \mathbf{d}_t$ that transforms the four vertices of t to the four vertices of t in $\bar{M}$

It's convenient to think of the triangle t in M as represented by its index in M's triangle table so that $\mathbf{S}_t$ and $\mathbf{d}_t$ are indexed by integers.

Our goal is to find a collection of transformations of the *target* mesh M' that are "as much like" those of M as possible; writing the target transformation for a target triangle s in the form

$$\mathbf{w} \mapsto \mathbf{T}_s\mathbf{w} + \mathbf{d}'_s, \tag{25.32}$$

our goal is to have $\mathbf{T}_s$ and $\mathbf{S}_t$ be as similar as possible whenever $(t,s) \in C$. Notice that we've ignored the translations of the source mesh here, and concentrated on

the intrinsic deformation and translation of each triangle. Because we've ignored the source translations, our solution will not be unique: We can add any constant translation to all the $\mathbf{d}'_s$ vectors and get an equally good solution. By explicitly setting the displacement $\mathbf{d}'_s$ for one triangle s in the target, we remove this ambiguity.

There is a further problem: If the vertex $\mathbf{w}_i$ is shared by two triangles s_1 and s_2, it's possible that

$$\mathbf{T}_{s_1}\mathbf{w}_i + \mathbf{d}'_{s_1} \neq \mathbf{T}_{s_2}\mathbf{w}_i + \mathbf{d}'_{s_2}. \tag{25.33}$$

In that case, the vertex will be sent to two different places by the two different transformations, and thus will not define a transformation on the mesh, M', but rather on the set of triangles in the mesh. Letting $N(\mathbf{w}_i)$ denote the set of all triangles that contain the vertex $\mathbf{w}_i$, we therefore seek transformations satisfying

$$\mathbf{T}_{s_1}\mathbf{w}_i + \mathbf{d}'_{s_1} = \mathbf{T}_{s_2}\mathbf{w}_i + \mathbf{d}'_{s_2} \text{ for all } s_1, s_2 \in N(\mathbf{w}_i). \tag{25.34}$$

We express this goal numerically as the problem of minimizing

$$\sum_{(s,t)\in C} \|S_s - T_t\|^2, \tag{25.35}$$

subject to

$$\mathbf{T}_{s_1}\mathbf{w}_i + \mathbf{d}'_{s_1} = \mathbf{T}_{s_2}\mathbf{w}_i + \mathbf{d}'_{s_2} \text{ for all } s_1, s_2 \in N(\mathbf{w}_i), \tag{25.36}$$

where $\|\mathbf{A}\|^2$ denotes the sum of the squares of the entries in the matrix $\mathbf{A}$. (The square root of this quantity is called the **Frobenius norm** of the matrix $\mathbf{A}$.) This is a quadratic optimization problem, which can be solved by standard numerical techniques. (As an aside, we caution you against writing your own quadratic optimizer, unless you are an expert in numerical analysis. Instead, find one you like, and get to be an expert in using it.) One problem with this formulation is the number of constraints: There's a constraint for every pair of triangles that meet at a vertex. Even if every vertex had degree three, this would still be a number of constraints that's equal to the number of vertices, which is very large in general. The problem begs for reformulation.

Sumner and Popović perform a natural transformation: Instead of treating the transformations $\mathbf{T}_i$ as unknowns, with constraints on where they send the mesh vertices, they treat the eventual vertex locations $\bar{\mathbf{w}}_i$ as unknowns, and write the transformations $\mathbf{T}_i$ in terms of these.

Recall that each source deformation transformation $\mathbf{S}$ was given by an expression of the form $\mathbf{S} = \bar{\mathbf{V}}\mathbf{V}^{-1}$. If we knew the final positions $\bar{\mathbf{w}}_i$ of the target vertices, then the target deformations would similarly be given by expressions of the form $\mathbf{T} = \bar{\mathbf{W}}\mathbf{W}^{-1}$. The entries of the matrix $\mathbf{T}$ are evidently *linear* functions of the unknown positions $\bar{\mathbf{w}}_i$. The minimization problem becomes

$$\min_{\bar{\mathbf{w}}_1,\ldots,\bar{\mathbf{w}}_n} \sum_{j=1}^{|M|} \|\mathbf{S}_{s_j} - \mathbf{T}_{t_j}\|^2, \tag{25.37}$$

where the $\mathbf{S}_{s_j}$ are all known, and in the expression for $\mathbf{T}$,

$$\mathbf{T} = \bar{\mathbf{W}}\mathbf{W}^{-1}, \tag{25.38}$$

the factor $\mathbf{W}^{-1}$ is known as well. Only $\bar{\mathbf{W}}$ is unknown. Thus, the summation above is a huge quadratic in the unknown positions. If we place all of these unknown positions into a $3n \times 1$ vector $\mathbf{x}$, then the minimization can be rewritten in the form

$$\min_{\bar{\mathbf{w}}_1,\ldots,\bar{\mathbf{w}}_n} \|\mathbf{c} - \mathbf{A}\mathbf{x}\|^2, \tag{25.39}$$

where $\mathbf{A}$ is a large, sparse matrix. In fact, if we place the x-coordinates of all the unknowns in the first n entries of $\mathbf{x}$, and then the y-components in the next n, and then the z-components in the third n entries, the matrix A ends up block-diagonal, consisting of three $n \times n$ blocks.

To minimize such a quadratic expression, we set the gradient to zero and solve; the result is the system of linear equations

$$\mathbf{A}^\mathbf{T}\mathbf{A}\mathbf{x} = \mathbf{A}^\mathbf{T}\mathbf{c}. \tag{25.40}$$

The matrix $\mathbf{A}$ depends only on the known data, so it needs to be computed only once, as does $\mathbf{Q} = \mathbf{A}^\mathbf{T}\mathbf{A}$. And solving a problem of the form

$$\mathbf{Q}\mathbf{x} = \mathbf{A}^\mathbf{T}\mathbf{c} \tag{25.41}$$

can be done with the LU decomposition of $\mathbf{Q}$ (which we need only compute once) and back-substitution. The LU decomposition can be found blockwise for the three blocks, further simplifying the computation.

The trickiest part of implementing this algorithm concerns bookkeeping: Transforming from an optimization in the form of Equation 25.35 to an optimization in the form of Equation 25.37 involves careful index manipulation. If you actually want to implement this idea, we strongly suggest that you first do so in two dimensions, and that you work with an example mesh consisting of no more than about five vertices and seven edges (which play the role of triangles in the two-dimensional formulation). It will also help to formulate the computations in a language like Matlab or Octave, in which matrix formulations are built into the language. Once you have made that work, transferring to some other language is much easier, especially since you can use the matrix-formulated implementation as a reference during debugging.

25.6.2 Triangle Reordering for Hardware Efficiency

As you know, the graphics pipeline, as implemented in the GPU, has several stages. Any one of these can be a bottleneck. For some models, transforming vertices dominates, and modern GPUs tend to cache such transformed vertices because of this. When it comes to generating triangles to be drawn, we get better cache use if those that share vertices are processed at nearly the same time. Triangle strips are one way to produce this kind of mesh locality. For other models, there may be enormous complexity that is mostly hidden (e.g., a model of an office building may contain millions of triangles, of which only a few hundred are visible from any particular office space). In these cases, if we can draw the visible triangles first, then a z-test will tell us that the fragments generated by other triangles are not visible, and thus that the lighting and shading computations for those fragments need not be performed. Of course, backface culling also will help reduce the rendering load by about 50% on average. By clever clustering of

triangles, it's possible to address the cache-miss problem quite effectively. The obvious solution to the "overdraw" problem, however, is view-dependent: If we can sort the triangles front to back, we can minimize overdraw. But such a sort order will likely break up the clusters that addressed the vertex-cache problem. Nonetheless, this front-to-back approach provides the core of the algorithm. We can find a front-to-back order for the triangles by creating a graph whose nodes are triangles and where there's a directed edge from t_1 to t_2 if t_1 obscures (partly or completely) t_2. Performing a topological sort on this graph gives a drawing order, assuming that the graph is acyclic. We'll return to the problem of cycles presently.

Nehab et al. [NBS06] have developed a solution that incorporates the best of both approaches, in the presence of backface culling: They create clusters that are large enough that only a small amount of cache-missing is introduced (compared to the optimal), but which also substantially reduce overdraw when the clusters are drawn in a particular order. Their approach relies on three key ideas.

1. If two polygons' normal vectors have a dot product of -1, then their sort order will have no impact on overdraw, because whenever one is front-facing, the other will be back-facing, and hence culled. For dot products greater than -1, the chance of overdraw increases with increasing dot product.
2. If their normal vectors have a positive dot product, it's possible that one obscures the other from many viewpoints, but the other never obscures the first. In this case, any sort order where the obscuring polygon comes before the obscured reduces overdraw.
3. The preceding observations are still true even for planar clusters of polygons, and even if the clusters are nearly planar rather than planar.

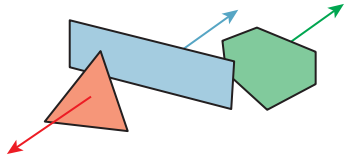


Figure 25.30: The sort order of the blue and red polygons is immaterial because of backface culling; the blue polygon obscures the green from some viewpoints, but the green never obscures the blue.

Figure 25.30 shows these situations. Notice that if the mesh is convex, then *any* sort order will minimize overdraw, because for a convex mesh, there's never any overdraw at all. Even for the nonconvex wave-shaped rooftop mesh of Figure 25.31, it's still possible to draw the polygons in an order that creates no overdraw. With these examples in mind, the algorithm has two broad steps: First, we create nearly planar connected clusters of triangles using a k-means-like clustering algorithm [HA79]; then we determine a sort order for the clusters by creating a graph whose nodes are the clusters and in which there's a directed edge from c_1 to c_2 if c_1 obscures c_2 more than c_2 obscures c_1 (averaged over all possible points of view). The edges are given weights depending on how much more c_2 obscures c_1 than c_1 obscures c_2. We then attempt a topological sort on this graph, using edge weights to break any cycles that arise.

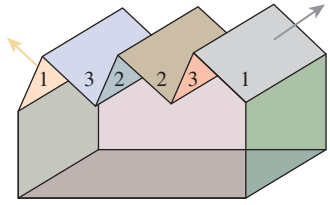


Figure 25.31: Drawing the rooftop regions of the building in increasing numerical order, and using backface culling, will prevent overdraw no matter what the viewpoint.

25.6.2.1 Clustering

The user must provide a number, k, of clusters to compute; k provides a tradeoff between vertex-cache efficiency and overdraw efficiency. (The authors report that between 10 and 100 clusters suffice for models on the order of 100,000 triangles.) We then select k random triangles and grow clusters from them. In general, a k-means algorithm has two steps that are alternated: adding items to a cluster based on a "distance" to some representative for the cluster, and updating the representative. For clustering points in a plane, the representative is typically the centroid of the cluster, the distance is Euclidean, and at each iteration, each item is added to the cluster whose center, from the last iteration, is nearest.

In the version of k-means used in this algorithm, the clusters begin with k randomly chosen "seed" triangles. Each cluster is represented by a centroid and a cluster normal; initially these are the centroid and face normal for the starting triangle. The clusters are "grown" by a breadth-first search (based on triangle adjacency) from the seed triangles, which are assigned a distance of zero from the cluster center. A triangle f adjacent to a triangle g of cluster C is assigned a distance equal to that of g plus $1 - \mathbf{n}_f \cdot \mathbf{n}_C + \epsilon$, where $\mathbf{n}_f$ and $\mathbf{n}_C$ are the face normal and average cluster normal, respectively, and ϵ is a small constant that ensures that triangles topologically close to the seed are preferred over those that are far away. Once distances from each face to all cluster centroids have been thus computed, each face is assigned to the cluster of the nearest centroid. In the second step, the cluster centroid and the average cluster normal $\mathbf{n}_C$ are recomputed, and the iteration is restarted. Because of the dot-product term in the incremental distance function, triangles closely aligned with the cluster normal tend to fall into the cluster, while those tilted away do not; this results in cluster boundaries tending to be aligned with sharp creases in the mesh. Note that a triangle may be adjacent to a cluster C along two of its edges (or all three); in this case, we add the incremental distance to the *lowest* of the already-computed distances of the neighbors.

25.6.2.2 Sorting

Once we have computed the clusters, we compute a value $I(c_1, c_2)$ that says how much cluster c_1 obscures cluster c_2, from enough viewpoints that our computation is a good estimate of the average occlusion, and we use this to build a graph whose nodes are labeled by the clusters. The value I is computed in *pixels:* It's meant to represent the number of possible overdraws that result from drawing c_2 before c_1, on average. If $I(c_1, c_2) > I(c_2, c_1)$, we should draw c_1 before c_2, so we place a directed edge from c_1 to c_2 in the graph, with edge weight $I(c_1, c_2) - I(c_2, c_1)$, which is always positive.

If the resultant graph admits a topological sort (which ignores the weights), we'll use it. If not, we want a sort order that minimizes the weights of all "violations" (i.e., cases where c_2 comes before c_1 in the sort order, even though there's an edge from c_1 to c_2). Unfortunately, this problem is NP-complete [Kar72], but a simple greedy heuristic works well. It's based on an algorithm for topological sort in which we choose as a starting node any vertex with only outgoing edges, and as an ending node any vertex with only incoming edges; we then remove these vertices and their associated edges from the graph and find a topological sort for the remainder.

In an unsortable graph, the approach above fails when every node is part of some cycle (i.e., has both incoming and outgoing edges). In this situation, we remove the node at which the weight sum for incoming edges is most different from the weight sum for outgoing edges, placing it on the "winning" side of the ordering (i.e., making it the next cluster to be drawn), and then continue with the normal topological sort.

Finally, within each cluster, we sort the triangles in a way to optimize mesh locality (i.e., to avoid vertex-cache misses), using some triangle-stripping algorithm like that of Hoppe [Hop99].

The results are impressive, generating up to 40% savings on overdraw for a model with 150,000 triangles. Of course, it's possible to construct exotic meshes for which almost no planar patch clusters exist; the algorithm will perform badly

on these. But for the kinds of meshes encountered in everyday games, for instance, the algorithm works well.

Sander et al. [SNB07] have improved on this algorithm, and we anticipate further research in this area, in which geometry and efficient computations are combined.

25.7 Discussion and Further Reading

The geometric study of meshes is a growing field, known as **discrete differential geometry.** Because of its close relationship to smooth differential geometry, you should start by becoming familiar with that material. A particularly gentle introduction is O'Neill's book [O'N06]; Millman and Parker's book [MP77] is a good follow-up. For discrete differential geometry, there are tutorials available [MDSB03], and at least one textbook [BS08].

The situation in what we've called "mesh flattening" (and what has come to be known as "parameterization of meshes") is not as hopeless as our remarks might suggest. There does not seem to be any single ideal approach to parameterization yet, but there's been substantial progress beyond the simplest approaches [SPR06, CPS11, SSP08].

The structure of a mesh and the structure of the underlying graph are closely related. The graph Laplacian has been used to address problems like graph partitioning and clustering; analogs have been used in mesh partitioning.

Ray-mesh intersection testing, since it's in the critical path for ray tracing, has been much studied. And because of its relevance to animation, so has collision detection for meshes. Many ideas can be shared between the two topics. Haines and Moller [AMHH08] give a complete overview, with details on many algorithms.

Mesh optimization has been widely studied, along with the relationship of mesh "smoothing" operations to digital filter design [Tau95] and to mean curvature flow [DMSB99, HPP05]. Methods that allow small connectivity alterations were popularized in graphics by Witkin and Welch [WW94], and can frequently be useful in adjusting meshes where some vertex degrees are so large or small that it's impossible to have all adjacent triangles approximately equilateral.

Despite all the research on meshes, they may, in fact, turn out to not be the ultimate shape representation model for graphics. For rendering, the discontinuity of reflection (you can adjust an incoming ray an arbitrarily small amount and get an arbitrarily large change in the reflected ray) is a serious problem, especially when one is trying to prove claims about convergence. The way that surfaces "condense" geometric information (like curvature) to low-dimensional subsets (vertices and edges) is reminiscent of the abstraction of the point light source, which condenses the light emitted from a small area into light emitted from a single point. Such an abstraction is convenient for some simple forms of rendering, but in fact makes others considerably more difficult. It's possible that mesh representations of surfaces will someday be regarded only as limiting cases for some other preferred kind of representation in which geometry generically has at least C^2 continuity.

Nonetheless, meshes are an active area of research. Pick up any SIGGRAPH proceedings from 2000 to 2012 and you'll find at least a dozen papers that concentrate on meshes in some form, and we anticipate that this trend will continue for some time. Read such papers once with an open mind, to get ideas, and again with

a menagerie of wild meshes, like the non-locally-flat example of Figure 25.14, and the highly wrinkled example of Figure 34.11, and a mesh built from an array of pieces shaped like Figure 25.32, where no two adjacent normal vectors are similar, and see whether the claims hold up. Another good test case is two large spheres that have been joined by removing a small triangle from each and either gluing the edges together directly, or splicing in a small triangular-prism-shaped "corridor" between them. (The boundary of the 2-ring of a vertex of that prism may well not be connected, for instance!)

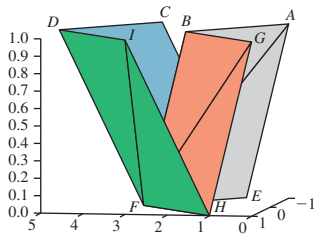


Figure 25.32: An eight-triangle building block.

25.8 Exercises

Exercise 25.1: Our algorithm for computing the boundary of a triangle mesh involved iterating through all the faces, and it runs in $O(f)$ time. Describe a connected triangle mesh with $O(f)$ boundary edges, thus showing that the $O(f)$ runtime is as good as possible.

Exercise 25.2: If we have a one-dimensional **path mesh** (i.e., each vertex has one or two edges meeting it), we can place it in the plane by assigning random locations to the vertices. In general, the result will not be an embedding: Edges will tend to cross one another. In 3-space, however, there's plenty of room, and with high probability the use of randomly assigned locations will result in an embedded path mesh. In this exercise, you'll show that for any surface mesh, there's an embedding in *some* Euclidean space. The idea is simple: For an n-vertex mesh, we'll place the mesh in $\mathbf{R}^n$ by placing vertex 1 at location $(1,0,\ldots)$, vertex 2 at location $(0,1,\ldots)$, etc. We then place edges and faces in the obvious way, using linear interpolation.

(a) Explain why the embeddings of triangles (i,j,k) and (i',j',k') do not intersect unless the sets $S_1 = \{i,j,k\}$ and $S_2 = \{i',j',k'\}$ have nonempty intersection.

(b) Show that if $S_1 \cap S_2$ contains a single index p, then the associated embedded triangles intersect only at vertex p.

(c) Show that if $S_1 \cap S_2 = \{p,q\}$, then the associated embedded triangles intersect in the embedded edge associated to $\{p,q\}$.

Exercise 25.3: Consider the mesh shown in Figure 25.32. By replicating it to the left and right, and then replicating the resultant strip in the front and back directions, we get a mesh in which no more than two adjacent triangles have remotely similar normal vectors. Show that such a mesh is likely to be a worst-case challenge to the polygon ordering algorithm of Nehab et al., as described in Section 25.6.2.

Exercise 25.4: (a) We sketched an algorithm for computing the boundary of a surface mesh by hashing edges. Adapt this to detect all *contours* of a mesh, where a contour edge is one where the normals to the two adjacent faces have dot products, with a view vector $\mathbf{v}$, of opposite signs, and the contour is the set of all contour edges.

(b) Contour edges can be generally formed into loops, although as we said, two loops may share one or more vertices. Design an $O(E)$ algorithm for assembling the E contour edges into loops. Hint: Use hashing again.

Exercise 25.5: The paper by Nealen et al. [NISA06] suggests using the mean curvature to compute weights. Make an argument for using the absolute value of the mean curvature instead. Can you think of any argument against this, or any rationale for why it might be an unimportant improvement, even if it works?

Chapter 26

Light

26.1 Introduction

We now turn to a more formal discussion of light, expanding considerably on the simple ideas from Section 1.13.1. We start with the physical properties of light, one of which is that it has many characteristics of waves, including a frequency. Our eyes perceive lights of different frequencies as having different colors, but since color is a perceptual phenomenon rather than a physical one, we treat it separately in Chapter 28.

The second part of this chapter is about the *measurement* of light and the various physical units we use to describe light. Since almost all of these can be described as integrals of one basic quantity (radiance), we also briefly discuss a few special integrals that arise often in rendering. Finally, although it is not strictly a property of *light,* we introduce the measurement of the *reflection* of light by surfaces, and compute the light reflected from a surface in two simple situations.

26.2 The Physics of Light

We live in a world in which electromagnetic radiation is everywhere. We're constantly bathed in both heat and light arriving from the sun, radio and television signals are present almost everywhere on Earth, etc. **Light** refers to a particular kind of electromagnetic radiation (of a frequency that can be detected by the human eye, or nearly so). Because of this relationship to the human eye, light has, over the years, been described not only in physical terms (like energy) but also in perceptual terms, things having to do with the way that the human visual system processes and perceives light. The most obvious of these is *color,* which we discuss in the next chapter in detail. We begin with the characteristics of the radiation at microscopic and macroscopic scales, and then move on to a discussion of how light is *measured.* The study of the measurement of radiation in general is called **radiometry,** and radiometric ideas are relatively easy to grasp, as is radiometry applied to light (i.e., electromagnetic radiation of the kind that the human eye can

detect). There's a second way of measuring light, called **photometry,** which is closely related to the human visual system; photometric measures of light tend to be **summary measures,** in that they measure things that can be computed from radiometric quantities by computing weighted sums. We'll touch on these measurement topics later in this chapter and the next.

At a macroscopic level, light can be regarded as a kind of energy that flows uninterrupted through empty space along straight lines, but is absorbed into and/or reflected from surfaces that it meets. At a microscopic level, light turns out to be **quantized**—it comes in individual and indivisible packets called **photons.** At the same time, light is **wavelike**—it is a kind of electromagnetic radiation and is characterized in part by a frequency, f. The energy E of a photon and the frequency f are related by

$$E = hf = \frac{hc}{\lambda},$$

where λ is the **wavelength** of the light in meters, $c \approx 2.996 \times 10^8\ \mathrm{m\,s^{-1}}$ is the speed of light, which is constant in a vacuum, and $h \approx 6.626 \times 10^{-34}\ \mathrm{kg\ m^2\ s^{-1}}$ is **Planck's constant.**

In graphics, we generally are interested in the macroscopic phenomena, and therefore we ignore the indivisibility of photons. But there are phenomena where the microscopic characteristics of light are important, particularly the wavelike characteristics. Since light is an *electromagnetic* phenomenon, it's actually described by both electric and magnetic characteristics; the magnetic characteristics are determined by the electrical ones, so we'll mostly ignore them. The electrical wavelike characteristics produce the phenomenon called **polarization.** Effects of the wavelike characteristics, including refraction and polarization, are actually important in some phenomena that we see in day-to-day life, such as the colors reflected by gemstones, the appearance of rainbows, the rainbow patterns seen on diffraction gratings, the colors seen in a thin layer of oil or gasoline on water (see Figure 26.1), the scattering of light by colloidal suspensions like milk, and the scattering of light through multilayered surfaces like human skin.

Figure 26.1: A thin layer of gas on wet pavement reflects a rainbow of colors due to diffraction.

We begin with the microscopic view because of its importance in explaining certain color phenomena, for instance, and because we feel that those working with light on a day-to-day basis should know something about its physical properties. But this material can safely be skipped by those who are only interested in high-level phenomena and are willing to take for granted certain claims about radiation that we'll make when discussing color.

We then continue with the macroscopic view, which can be easily understood by analogy with everyday phenomena.

26.3 The Microscopic View

In this section we'll give a high-level overview of the nature and production of light; those interested in further details should begin with a good understanding of electricity and magnetism (we particularly recommend Purcell's book [Pur11]).

Let's start with a simple model of an atom consisting of a central nucleus surrounded by electrons, which we depict in Figure 26.2 as circling about the nucleus in orbit. Electrons in orbits farther from the nucleus have more energy than those close to the nucleus (just as it takes more energy to launch a high-orbit satellite

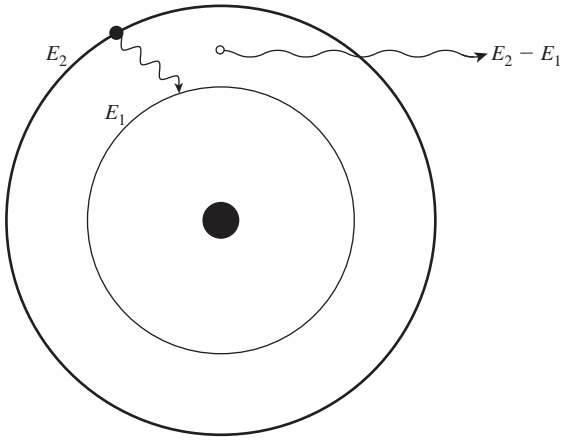


Figure 26.2: An atom has a nucleus around which electrons orbit in various orbital levels. Each orbital is associated with an energy level. An electron can "fall" from an orbital of higher energy, E_2, to an orbital of lower energy, E_1; when it does, a photon with energy $E_2 - E_1$ is emitted. The reverse can also happen: A photon with energy $E_2 - E_1$ can be absorbed by the atom, lifting the electron from the orbital of energy E_1 to that of energy E_2.

around the Earth than a similarly sized low-orbit satellite). An electron can drop from a high-energy orbit to a lower-energy one; typically when this happens, a photon is emitted; its energy is the difference between the two energy levels. An atom can also absorb a photon of energy E by having some electron move into a new orbit whose energy is exactly E higher than that of its current orbit. Sometimes there is no pair of orbits whose difference is exactly E; in this case, the photon cannot be absorbed by the atom. Electrons can also change levels through other mechanisms, one of which is vibration, in which some of the energy of an electron in a substance is converted to or from vibration of the atoms of the substance.

A typical phenomenon is that a photon is absorbed by an atom, raising the electron to a new energy level; the electron, some brief time later, then falls back down to the lower energy and a new photon is emitted. Sometimes the path to the lower energy level goes through an intermediate level: First some electron energy is converted to vibration, and then a photon-emitting energy jump takes place. The outgoing photon has a lower energy than did the incoming one; this phenomenon is called **fluorescence.** The most familiar examples are minerals which, when illuminated by ultraviolet light (sometimes called black light), emit visible light. There is a closely related phenomenon called **phosphorescence,** in which the transition from the intermediate state to the low-energy state is relatively unlikely, and therefore can take place over a long period of time. A phosphorescent material, illuminated by light, can continue to glow for some time after the illumination is removed. There's one other form of interaction between a photon and an atom: Sometimes the photon kicks an electron to a higher energy state, from which it returns to the original state almost immediately; the result is that the photon continues on its original way, slightly delayed. The likelihood of such **virtual transitions** depends on the nature of the material, but the delay they induce has an important macroscopic effect: The speed of light through materials is slower than that in a vacuum, with the slowness being determined by how often such virtual transitions occur.

The simple model of discrete energy levels really applies only to an isolated atom. When multiple atoms are in proximity (as in solids), each individual energy level available to electrons gets "spread out" into a **band** of energies. Still, electrons can generally absorb or release energies only when the amount absorbed or released represents the difference of two energies in the bands.

In some materials—like metals—certain electrons are not attached to particular nuclei, but can instead move about the material, helping to make the material conductive. These electrons have a great many possible energy states, and therefore can absorb photons of many different wavelengths and then promptly emit them again. This generally makes conductive materials like metals reflective, while most transparent materials are insulators.

In other materials—like some forms of carbon—there are also unattached electrons, but they cannot move quite as freely. Such an electron can interact with atoms of the material, causing those atoms to move and vibrate while the electron loses energy. This motion of atoms is called **heat.** Thus, materials like soot tend to absorb photons, and rather than reemitting the photons, they convert them into heat. This is why soot looks black, and why dark clothes heat up on a sunny day. Note that light of *all* frequencies is convertible to heat. In particular, infrared light (light of wavelengths slightly longer than those we can see) is a kind of electromagnetic radiation, just like the light we see; it happens to be more readily convertible to heat than is visible light, but it's still light.

In the exact reverse process, if we heat up soot, the atoms vibrate; this vibration in turn may "kick around" a loose electron, causing it to have excess energy, which it may lose by emitting a photon. Because of the many possible energy states for the loose electron, the emitted light can have many possible energies (wavelengths). Thus, materials that are good at absorbing energy and converting it to heat are also good at emitting energies of many different amounts when they are heated.

As materials are heated, they all become increasingly better at emitting electromagnetic radiation. Indeed, all bodies at all temperatures above absolute zero actually emit *some* radiation, but at the low temperatures we encounter in ordinary life, it's not very much. We mostly see things because they are *reflecting* light rather than because they are emitting it themselves. The exceptions are things like the filaments in incandescent lightbulbs, hot metal being forged by a blacksmith, or the sun.

We can measure the energy radiated by an object heated to temperature T (see Figure 26.3). For each narrow range of wavelengths, we can measure the energy radiated in that range; plotting this function $I(\lambda, T)$ against frequency λ gives a graph like that shown in Figure 26.4. At very low temperatures, such measurements are easily confounded with reflected energy. But if we imagine an ideal **black body**—one that can absorb and emit electrons as well as possible—in a room in which the only energy is in the form of heat, we get a plot like the one shown in the figure.

The dependence of I on T can be measured; the total power radiated depends on the *fourth* power of T:

$$power = \sigma T^4, \text{where } \sigma = 5.67 \times 10^{-8} \text{ W m}^{-2} \text{ K}^{-4};$$

this is known as the **Stefan-Boltzmann law.** (The K in this expression denotes "degrees Kelvin.")

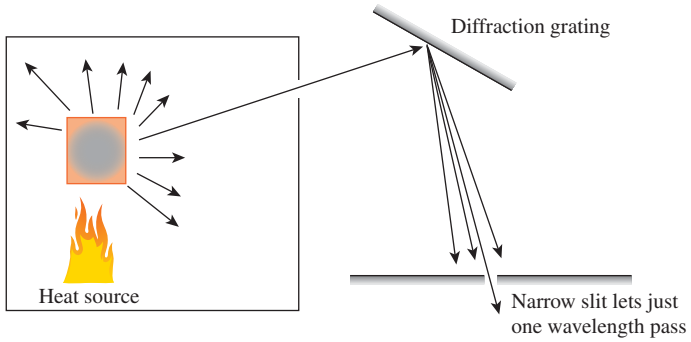


Figure 26.3: An object is heated to some temperature, and a narrow beam of the radiation it produces is focused on a diffraction grating, splitting it into energies of different wavelengths. By moving a plate with a slit in front of this diffracted energy, we can measure the energy radiated in a narrow range of wavelengths, $[\lambda, \lambda + d\lambda]$.

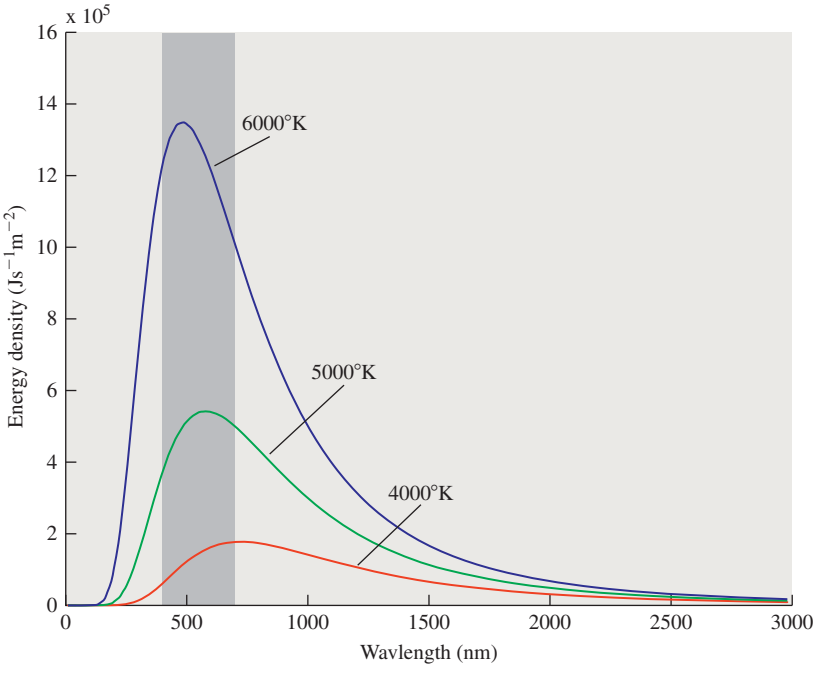


Figure 26.4: The radiation near wavelength λ *from a black body heated to some temperature T, plotted as a function of* λ*, shown for several values of T. The shaded region indicates the wavelengths of visible light.*

Inline Exercise 26.1: On a warm day the temperature is about 300°K.
(a) What does the Stefan-Boltzmann law predict as the amount of energy radiated from your body? (You should assume that your surface area is about one square meter, and that you radiate as a black body.)
(b) When sitting at home on such a day, why do you not get very cold from this loss of heat?

The other notable feature in the graph is that the location of the peak radiation intensity moves to the left as the temperature increases. At about 900°K, there is enough radiation in the visible portion of the spectrum for the eye to detect it. As a first peek at color, we mention one thing: Radiation with a wavelength between about 400 nanometers and 700 nanometers is visible to the human eye; radiation at the 700-nanometer end of the spectrum looks red, and the appearance transitions through yellow, green, and cyan as the wavelength shortens (i.e., the energy increases); and radiation near the 400-nanometer end of the visible spectrum looks blue. Since at 900°K the radiated energy at the low-frequency (i.e., long-wavelength) end of the visible spectrum is larger than that at the high-frequency end, we see such an object glowing a dull red. As we heat it further, it becomes a great deal brighter because of the exponent of 4 in the Stefan-Boltzmann law. But higher frequencies begin to mix in, and we see a combination of red and green (i.e., an orange and then a yellow color) and eventually a combination of red, green, and blue, which we perceive as white. By the time an object is glowing white, it's emitting energy at an amazing rate; at 5000°K, it's radiating at about 35 megawatts per square meter. Clearly this radiation dominates whatever light the surface might reflect from the ordinary illumination in a room, for instance.

By the way, lamps used in filmmaking and photography are often described using temperatures; that's shorthand for saying, "The spectrum of light emitted by this lamp is quite similar to that of black-body radiation of that temperature." This can be useful in adjusting a scene to appear illuminated by ordinary incandescent lamps or by sunlight.

Max Planck developed an expression for the shape of the curve in the graph above, later supported by theoretical analysis based on quantum theory; he observed that

$$I(\lambda, T) \propto \frac{1}{\lambda^5} \frac{1}{e^{\frac{hc}{\lambda kT}} - 1};$$

where h is Planck's constant and k is **Boltzmann's constant** (about $1.38 \times 10^{-23}\,\mathrm{J\,K^{-1}}$). The precise values are not important to us, but the shape of the curve is. Because $e^x = 1 + x + \ldots$, the denominator of the second factor is, for large λ, roughly proportional to $1/\lambda$, so $I(\lambda, T)$ is proportional to λ^{-4}; for small λ, the exponential dominates and the curve heads to zero. Note that $I(\lambda, T)\Delta\lambda$ is the amount of energy at wavelengths between λ and $\lambda + \Delta\lambda$, for small values of $\Delta\lambda$; to find the total energy in some range of wavelengths, you have to integrate with respect to λ over that range. The corresponding expression, in terms of frequency, which is the more common descriptor used for light in physics, is

$$R(f, T) = \frac{f^3}{e^{hf/kT} - 1},$$

in which frequency appears to the third power, while wavelength appeared to the fifth power; this is because integration with respect to f involves a change of variables from λ to f, namely, $\lambda = c/f, d\lambda = -c/f^2 df$.

26.4 The Wave Nature of Light

As mentioned earlier, light is a kind of electromagnetic radiation. (Indeed, "light" is a general term for this, with "visible light" being the radiation that the human eye can detect. We'll generally follow common usage and mean "visible light"

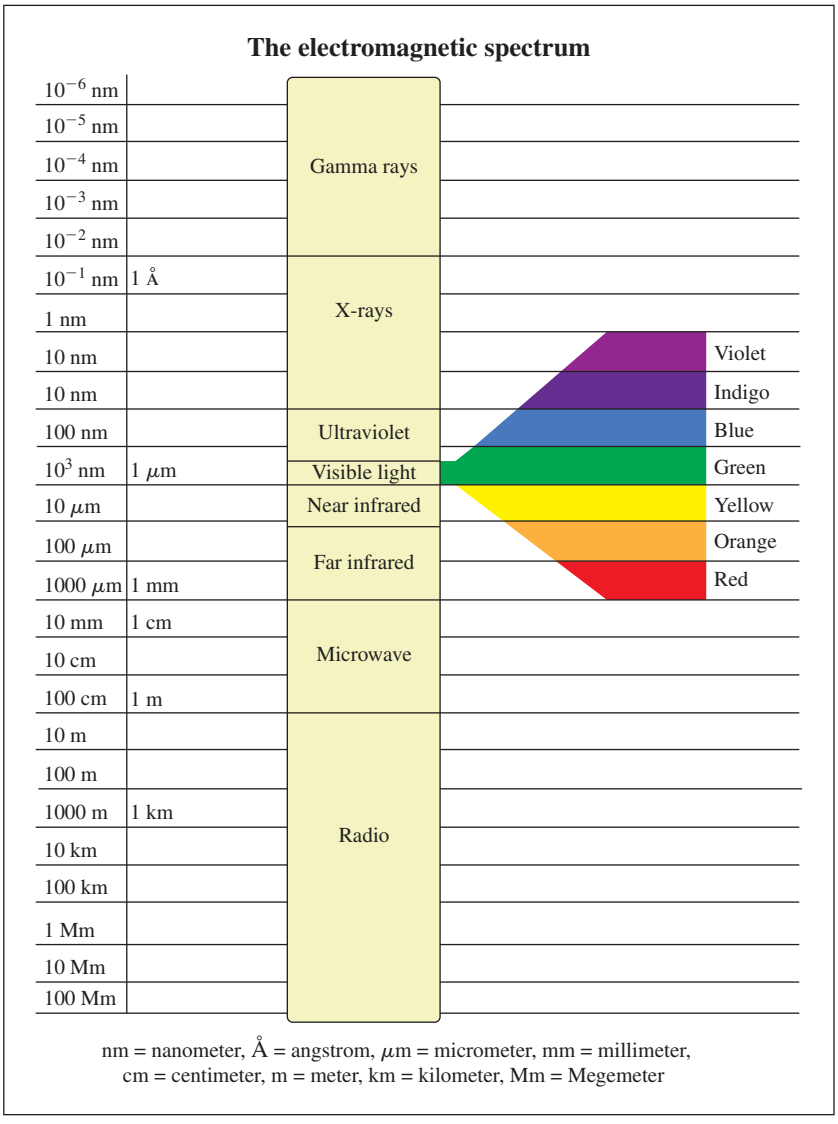


Figure 26.5: The electromagnetic spectrum includes many different phenomena; visible light occupies only a small portion of the spectrum.

when we speak of "light.") Other kinds include X-rays, microwaves, etc. (see Figure 26.5). The wave nature of light is best used when trying to understand how light propagates; in fact, a good rule of thumb is that "[e]verything propagates like a wave and exchanges energy like a particle" [TM07]. To understand the propagation of light, we must discuss kinds of waves.

Large and regular waves on the surface of the ocean are **linear waves**—each peak and trough consists of a long line that moves in a direction perpendicular to the axis of the line (see Figure 26.6). The **wavelength** is the perpendicular distance between adjacent peaks (or adjacent troughs). The **wave velocity** is the velocity with which the peak moves. This is not the velocity of any individual particle of water, which is easy to see by watching, for instance, a log floating on the

Figure 26.6: Ocean waves arriving at Panama City. The waves come in long lines, which have been slightly bent by the irregularities of the ocean floor as they approach the shore. (Courtesy of Nick Kocharhook.)

surface: As waves pass, the log rises and falls, and may also move somewhat back and forth in the direction of the wave, but long after the peak of the wave has moved on, the log remains more or less where it started.

Inline Exercise 26.2: (a) A thin human hair has a diameter of $50\,\mu\text{m} \approx \frac{1}{500}$ inch. Red light has a wavelength of about 700 nm. How many red wavelengths is one hair diameter?
(b) **Diffraction** is an effect that typically occurs when a wave phenomenon interacts with an object whose scale is about the same order of magnitude as the wavelength of the wave. Do you expect to see diffractive effects in the interaction of human hair and visible light?

A basic electromagnetic wave moving through space is a **planar wave.** Just as an ocean wave has a height at each point of the ocean surface, a light wave has an electric field at each point of space. And just as the heights of the ocean wave are the same all along a ridge line or a trough line, the electric field is the same all along a *plane* (at least within some large enough radius that this is a decent approximation). This means that we can describe the plane wave by describing its values along a single line perpendicular to that plane. For instance, if the wave is constant along planes perpendicular to the x-axis, then we can know, at each time t, its value at a point (x, y, z) by knowing its value at $(x, 0, 0)$:

$$E(x, y, z, t) = E(x, 0, 0, t).$$

The velocity with which the peaks of the wave move along the x-axis is c, the speed of light, and the wave shape is sinusoidal. This means that the expression for, say, the y-component of the wave must have the form

$$E_y(x, 0, 0, t) = A_y \sin\left(2\pi\frac{x}{\lambda} - 2\pi ft + \Delta_y\right) \tag{26.1}$$

$$= A_y \sin\left(2\pi\frac{x}{\lambda} - 2\pi\frac{c}{\lambda}t + \Delta_y\right), \tag{26.2}$$

where Δ_y is a "phase" that depends on our choice of the origin of our coordinate system. Similarly, the z-component must have the form

$$E_z(x,0,0,t) = A_z \sin\left(2\pi\frac{x}{\lambda} - 2\pi\frac{c}{\lambda}t + \Delta_z\right). \tag{26.3}$$

Inline Exercise 26.3: Suppose we change units so that the speed of light, c, is 1.0; assume that $\lambda = 1$ as well, and $\Delta_z = 0$. Plot E_z as a function of x when $t = 0$; do so again when $t = 0.25, 0.5, 0.75$, and 1.0.

Physical experiment confirms that the x-component of the electric field for a wave traveling on the x-axis is always zero. Thus, the vector $\mathbf{a} = \begin{bmatrix} A_x & A_y & A_z \end{bmatrix}^{\mathbf{T}}$ that characterizes the plane wave must always lie in the yz-plane; A_y and A_z can take on any values, but A_x is always zero.

26.4.1 Diffraction

The first important phenomenon associated with the wave nature of light is **diffraction.** Just as waves passing through a gap in a breakwater fan out into a semicircular pattern, light waves passing through a small slit also fan out. Assuming the slit is aligned with the y-axis and the plane waves are moving in the x-direction, the electric field (after the light passes through the slit) will be aligned with the y-direction, that is, A_z will be 0.

If we place an imaging plane at some distance from the slit (see Figure 26.7), a pattern of stripes indicating the wave nature of the light appears.

For the most part, this kind of diffraction effect is not evident in day-to-day life, but a closely related phenomenon, in which light of different wavelengths is reflected in different directions by some medium (things like the "eye" of a peacock feather, or a prism), is quite commonplace.

26.4.2 Polarization

In studying the electric field associated to light moving in the x-direction, we have a plane wave described by

$$E_x(x,0,0,t) = 0 \tag{26.4}$$

$$E_y(x,0,0,t) = A_y \sin\left(2\pi\frac{x}{\lambda} - 2\pi\frac{c}{\lambda}t + \Delta_y\right) \tag{26.5}$$

$$E_z(x,0,0,t) = A_z \sin\left(2\pi\frac{x}{\lambda} - 2\pi\frac{c}{\lambda}t + \Delta_z\right). \tag{26.6}$$

The phase constants Δ_y and Δ_z depend on where we choose the origin in x or t; if we replace x by $x + a$, then both Δ_y and Δ_z will change, but the *difference between them will remain the same*. This difference can be any value at all (mod 2π); in typical light emitted from an incandescent lamp, for instance, all possible differences between 0 and 2π are equally likely.

The simplest case is a plane wave where $A_y = A_z$ and $\Delta_y - \Delta_z = \pi/2$ or $3\pi/2$. Such a wave is called **circularly polarized.** If we consider the electric field of Equation 26.6 at time $t = 0$ and assume that we've adjusted the x-axis so that $\Delta_y = 0$ and $\Delta_z = \pi/2$, the field has the form

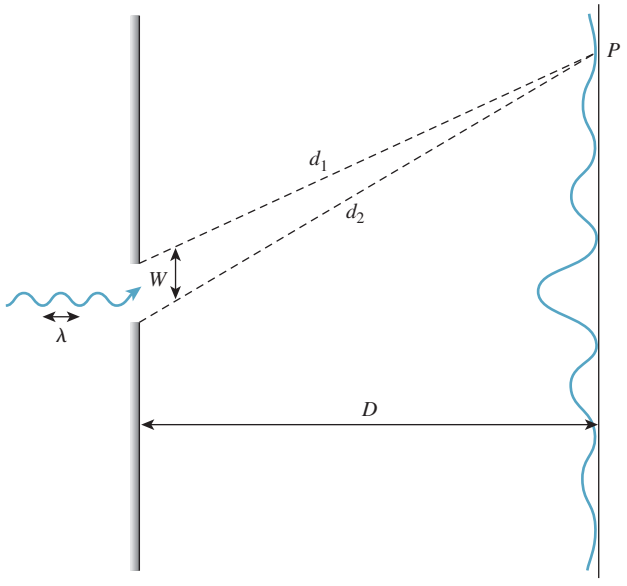


Figure 26.7: Light passing through a narrow slit spreads out to illuminate a surface behind the slit. Light from each side of the slit has different distances d_1 and d_2 to the back plane. When these distances are a half-wavelength apart, the light waves cancel; when they're a multiple of a full-wave apart, they reinforce each other. This results in a set of bands of light and dark on the imaging plane with spacing approximately $\lambda D/w$.

$$E_x(x,0,0,t) = 0 \tag{26.7}$$

$$E_y(x,0,0,t) = A_y \sin\left(2\pi\frac{x}{\lambda}\right) \tag{26.8}$$

$$E_z(x,0,0,t) = A_y \cos\left(2\pi\frac{x}{\lambda}\right). \tag{26.9}$$

Notice that for every value of x, the vector $\mathbf{E}$ is a point on the circle of radius A_y in the yz-plane. Figure 26.8 shows this. We've plotted in blue the electric field along the x-axis at a fixed time t. The projection of this field to the xy-plane, shown in red, is sinusoidal. The projection to the xz-plane, in green, is also sinusoidal, with the same amplitude, because $A_y = A_z$. The projection operation, for one vector, drawn in black, is shown by two magenta dashed lines. The projection of all these vectors to the yz-plane, shown in black, forms a *circle* in that plane.

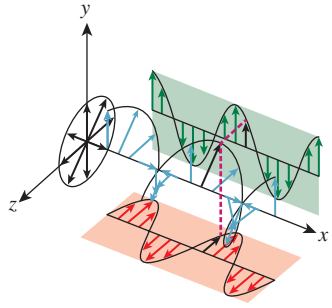


Figure 26.8: Circular polarization.

Inline Exercise 26.4: What happens to the preceding analysis when $\Delta_y = 0$ and $\Delta_z = -\frac{\pi}{2}$? These two similar, but different, situations are called clockwise and counterclockwise polarization.

At the other extreme, consider the case where $\Delta_y = \Delta_z = 0$. In this case, the electric field vector at every point of the x-axis is a scalar multiple of $\begin{bmatrix}0 & A_y & A_z\end{bmatrix}^{\mathbf{T}}$, that is, the electric field vectors all lie in one line. Figure 26.9 shows this: The projections of these vectors to the yz-plane all lie in one line, determined by the numbers A_y and A_z. Such a field is said to be **linearly polarized,** with the direction $\begin{bmatrix}0 & A_y & A_z\end{bmatrix}^{\mathbf{T}}$ being the axis of polarization.

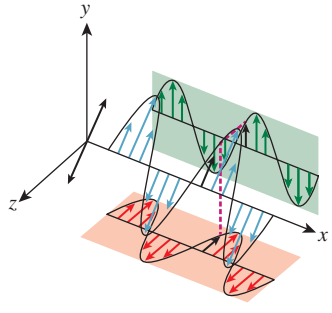


Figure 26.9: Linear polarization.

Inline Exercise 26.5: What happens when $\Delta_y = 0$ and $\Delta_z = \pi$?

Finally (see Figure 26.10), there are cases where $\Delta_y - \Delta_z$ is not a multiple of $\frac{\pi}{2}$, or where A_y and A_z differ. In this case, the projection of the field vectors to the yz-plane forms an *ellipse,* and the light is said to be **elliptically polarized.** It turns out that an elliptically polarized field can always be expressed as the sum of a circularly polarized one and a linearly polarized one, with the axis of linear polarization being along the major axis of the ellipse (see Exercise 26.11).

There are materials called **polarizers** that are transparent to waves of one polarization but opaque to those of the opposite polarization.

Light traveling in some direction **d** and reflected from a shiny surface with normal vector **n**, when reflected, ends up preferentially linearly polarized with the polarization direction being $\mathbf{d} \times \mathbf{n}$. Such reflected light, when observed through a polarizer that favors light of some other polarization, will be attenuated. This is the principle by which polarized sunglasses filter reflected sunlight. The precise nature of reflected polarized light depends on the reflecting material, as we'll see in Section 26.5.

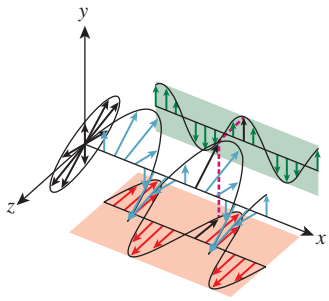


Figure 26.10: Elliptical polarization

26.4.3 Bending of Light at an Interface

In a related phenomenon, light passing from one medium to another changes speed. The speed of light in a vacuum is the highest possible speed; in other materials it may be substantially slower, due to the virtual transitions mentioned earlier. As a result, when light passes from a vacuum to some material it slows down. This does not affect the frequency of the light, that is, the number of peaks of electromagnetic radiation arriving at a fixed point in a fixed amount of time. You can convince yourself of this by observing a person who jumps into a swimming pool: The color of his or her clothing does not appear to change whether you're seeing it through water and air or just air. The speed change does affect the wave*length,* however, which is determined by

$$\lambda = s/f,$$

where s is the speed of light in whatever medium it's traveling through and f is the frequency.

The **index of refraction** or **refractive index** of a medium is the ratio of the speed of light in a vacuum to the speed of light in that medium. It's denoted by the letter n. Typical indices of refraction are 1 for a vacuum, 1.0003 for air, 1.33 for water, and 2.42 for diamond.

The difference of refractive index in different media causes a macroscopic phenomenon: Light rays bend when they go from one medium to another. The conventional name for the precise description of the bending is **Snell's law,** although the phenomenon of a consistent law of refraction was known to Ibn Sahl of Baghdad as early as 984 CE [Ras90].

The bending follows a particularly simple form (see Figure 26.11): If θ_1 and θ_2 denote the angles between the ray and the surface normal on the two sides of the interface, and n_1 and n_2 denote the two indices of refraction, then

$$\frac{n_2}{n_1} = \frac{\sin\theta_1}{\sin\theta_2}.$$

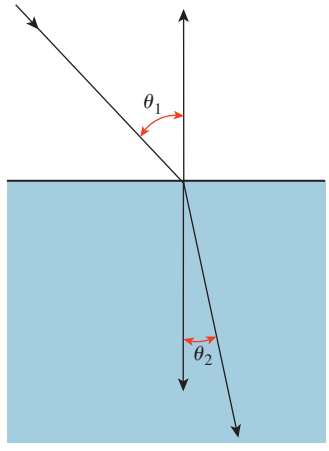


Figure 26.11: Bending of a light ray passing from one medium to another.

This tells us that if we know n_1, n_2, and θ_1, we can determine θ_2. The index of refraction, n, determines a great deal about how light interacts with a medium. For instance, when light arrives at the interface between air and glass at some angle, the refractive index determines not only the amount of bending, but also how much of the light is transmitted through the glass and how much is reflected, as we'll see in Section 26.5. Those interested in physically realistic renderings of glass and other transparent materials must take this fact into account.

Using only the assumption that the electromagnetic field has a sinusoidal form with the same frequency in each medium, and is continuous, we can prove Snell's law. Fortunately, the mathematics of this explanation apply to *any* kind of wave, not just plane waves in three dimensions, so we can illustrate the idea with linear waves in a plane. Consider the situation shown in Figure 26.12: a shallow tray whose left side is twice as deep as its right side; this makes waves on the left half travel about twice as fast as those on the right half. If we create sinusoidal waves of frequency f on the left side, moving right, when they enter the right side they "bunch up." Because of the slower speed on the right, the same number of waves per second reach the right-hand side of the tray as reached the midline.

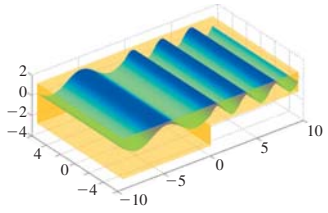


Figure 26.12: Waves traveling to the right bunch up as they slow down.

When we create waves traveling in an off-axis direction (see Figure 26.13) in the left half of the tank, they arrive at the dividing line between the two sides and continue on as waves in the shallower right-hand side of the tank. The peaks of the waves on the two sides of the tank must match up at the dividing line if the wave height is to be a continuous function; for this to happen, the directions of propagation must differ.

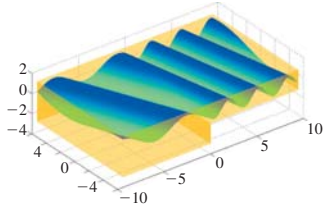


Figure 26.13: Off-axis waves change direction "at an interface."

Inline Exercise 26.6: (a) Suppose that the wavelength of the waves in the left half of the tank is λ, and the direction of propagation in the left side is at angle $\theta_L \neq 0$ to the left-right axis of the tank. Show that along the midline of the tank, the distance between peaks is $\lambda / \sin(\theta_L)$.
(b) For a corresponding statement to be true on the right side of the tank, where the wavelength is about $\lambda/2$, the distance between peaks will be $(\lambda/2)/\sin(\theta_R)$. Setting these equal, show that $\sin(\theta_L)/\sin(\theta_R) = 2$, which is exactly the ratio of the velocities in the two sides of the tank.

A similar phenomenon happens with plane waves that meet at a planar interface between media, from which Snell's law follows as a consequence.

The index of refraction of a medium is not *really* a constant: It depends on the wavelength of the light. Cauchy developed an empirical approximation for the dependence, showing it was of the form

$$n(\lambda) \approx A + \frac{B}{\lambda^2},$$

where the values of A and B are material-dependent. The exact values are not important, except that $B \neq 0$. This means that light of different wavelengths, arriving at an interface between different media, gets bent by different amounts: The different wavelengths are separated from one another. One instance of this is the rainbows cast by prisms when they are struck by sunlight. Another is that lenses, which are supposed to focus light at a single point, actually focus light of different wavelengths at different points: When red light is in focus, blue light will be blurry, etc. This **chromatic aberration** is a significant problem in lens design, and many lens coatings are designed to minimize it.

26.5 Fresnel's Law and Polarization

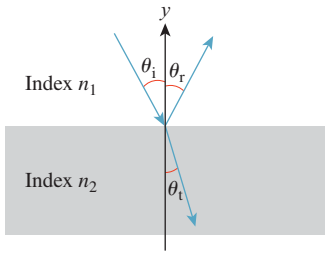


Figure 26.14: A light ray reflects and transmits through an interface between media.

Consider Figure 26.14, which shows light arriving at the interface between two media, the upper ($y > 0$) having refractive index n_1 and the lower ($y < 0$) having index n_2. For now, we'll assume that the media are insulators rather than conductors. The light's direction of propagation lies in the xy-plane, the plane of the diagram. The arriving light makes angle θ_i ("i" is for "incoming") with the y-axis; the reflected light makes angle $\theta_r = \theta_i$, and the transmitted light makes angle θ_t with the negative y-axis. Since the electric field associated to the incoming light must be perpendicular to the direction of propagation, we'll consider two special cases. In the first, the electric field, at each point of the incoming ray, points along the z-direction (i.e., *parallel* to the interface between the media, pointing either into or out of the page). A light source with this property is said to have "parallel" polarization with respect to the surface, or be **p-polarized.**

When such a wave reaches the surface the electric field interacts with the electrons near the interface, moving them back and forth in the z-direction; these motions in turn generate a new electromagnetic field that's a sum of two parallel-polarized waves, the first corresponding to the transmitted light and the second to the reflected light. The transmitted light travels in a direction described by Snell's law, and the reflected light travels according to the familiar "angle of incidence equals angle of reflection" rule: $\theta_r = \theta_i$. The *fraction* R_p of light *reflected* depends on the angle θ_i according to the rule

$$r_p = \frac{n_2 \cos\theta_i - n_1 \cos\theta_t}{n_2 \cos\theta_i + n_1 \cos\theta_t} \tag{26.10}$$

$$R_p = r_p^2. \tag{26.11}$$

the fraction transmitted T_p is just $1 - R_p$. (These fractions denote the fraction of the incoming *power* that leaves in each direction. The *amplitude* of the reflected wave is just r_p times the amplitude of the arriving wave.) These formulas can be derived, like Snell's law, by insisting on continuity at the interface [Cra68].

The *phase* of the reflected light may match that of the arriving light, lag behind it, or lead it, or be 180° out of phase with it.

The other special case is when the electric field is perpendicular to the z-axis, that is, it lies entirely in the xy-plane, perpendicular to the direction of propagation. Such a wave is said to be **s-polarized.** In this case, the reflection coefficient R_s is given by

$$r_s = \frac{n_1 \cos\theta_i - n_2 \cos\theta_t}{n_1 \cos\theta_i + n_2 \cos\theta_t} \tag{26.12}$$

$$R_s = r_s^2 \tag{26.13}$$

Once again, the transmission coefficient T_s is $1 - R_s$. These rules for the reflection and transmission coefficients for s- and p-polarized waves are called the **Fresnel equations,** after Augustin-Jean Fresnel (1788–1827).

Because every wave can be written as a sum of an s-polarized and a p-polarized wave, these two special cases in fact tell the whole story. For instance, incoming light that is linearly polarized as the sum of a wave that is equal parts s-polarized and p-polarized will reflect and again be linearly polarized. But the ratio of s- to

p-polarized components will no longer be one to one; instead it will be R_s/R_p. Because in general $R_s > R_p$, the reflected light will be more s-polarized than the incoming light. In fact, no matter what the mix of s- and p-polarized light in the arriving light, the outgoing light will be more s-polarized than was the incoming light. The same argument applies to circularly polarized light; the only difference is one of phase, which does not enter the Fresnel equations. Incoming circularly polarized light will generally reflect into elliptically polarized light, with the s-component dominating.

Inline Exercise 26.7: (a) For $n_1 = 1$ and $n_2 = 1.5$ (corresponding approximately to air and glass), plot R_p against θ_i, and observe that at about 56°, R_p is 0; what does this tell you about the polarization of the reflected light when light arrives at this angle?
(b) For any pair of materials, there's a corresponding angle; it's called **Brewster's angle.** Briefly explain why Brewster's angle depends only on the ratio of the indices of refraction of the materials.

Inline Exercise 26.8: Consider light traveling from a piece of glass to the air (so $n_1 = 1.5$ and $n_2 = 1.0$). Plot R_s and R_p against θ_i for $0 \leq \theta_i < \sin^{-1}(n_2/n_1) \approx \sin^{-1}(.66)$. At the upper end of this range, the **critical angle,** both R_s and R_p are 1; all light is reflected back into the glass and none escapes to the air. This is called **total internal reflection.**

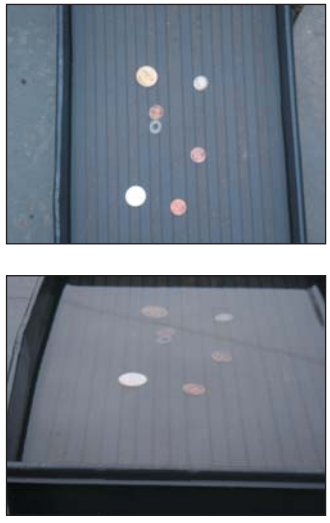


Figure 26.15: Fresnel's law in action: The coins are easily visible from overhead, but are obscured by sky reflections when seen at a diagonal.

Figure 26.15 demonstrates Fresnel's law. The first photo shows several coins and a washer in a tray as seen from above on a calm, overcast day. The second shows the same items, seen from about 45°. So much of the incident light is reflecting that it's much more difficult to see the items in the tray.

The analysis above applies to insulators. For conductors, the transmitted light is almost immediately absorbed, and the *rate* at which it's absorbed has an effect on the reflected light. One analysis revises the index of refraction to be a *complex* constant, whose real and imaginary parts correspond to the usual index of refraction and the amount of absorption in the material, known as the **coefficient of extinction** and denoted κ. An alternative approach simply treats the refractive index and coefficient of extinction as separate quantities. In this latter form, a good approximation to the Fresnel reflectance for conductors (in air) is given by

$$R_s = \frac{(n_2^2 + \kappa^2)\cos^2\theta_i - 2n_2\cos\theta_i + 1}{(n_2^2 + \kappa^2)\cos^2\theta_i + 2n_2\cos\theta_i + 1} \text{ and} \tag{26.14}$$

$$R_p = \frac{(n_2^2 + \kappa^2) - 2n_2\cos\theta_i + \cos^2\theta_i}{(n_2^2 + \kappa^2) + 2n_2\cos\theta_i + \cos^2\theta_i}, \tag{26.15}$$

where n_2 is the index of refraction of the metal and κ is its coefficient of extinction.

Snell's and Fresnel's laws are quite general, but there are materials whose behavior is more interesting than that described by these equations. Calcite, for instance, exhibits **birefringence,** in which there are *two* directions of refraction

rather than one; so does topaz. (That's because the speed of light is different in different directions through these materials!)

26.5.1 Radiance Computations and an "Unpolarized" Form of Fresnel's Equations

While Fresnel's laws describe transmitted and reflected power, in graphics we're mostly concerned with radiance, which we'll define in the next few sections. Because radiance involves an angle measure in its definition, and the angle between two light beams refracted by Snell's law is different before and after refraction, the ratio between outgoing and incoming *radiance* involves an extra factor of

$$\frac{\sin^2\theta_i}{\sin^2\theta_t} = \frac{n_2^2}{n_1^2}. \tag{26.16}$$

The derivation of this factor is given in the web materials for this chapter.

Although we've observed that light, after reflection, tends to be increasingly polarized, it's common in graphics to treat light as **unpolarized,** that is, to assume that the polarization of incident light is, on average, zero. With that assumption, the Fresnel equations can be simplified to a single factor, called the Fresnel reflectance, which is

$$R_F = \frac{1}{2}(R_s + R_p). \tag{26.17}$$

The energy reflected is R_F times the incident energy. And the energy transmitted is $(1-R_F)$ times the incident energy. This means that the reflected and transmitted radiance values can be computed as

$$L(P, \omega_r) = R_F L(P, -\omega_i) \text{ and} \tag{26.18}$$

$$L(P, \omega_t) = (1 - R_F)\frac{n_2^2}{n_1^2} L(P, -\omega_i). \tag{26.19}$$

Note that R_F here depends implicitly on θ_i, n_1, and n_2 which, together with Snell's law, lets us compute θ_t.

26.6 Modeling Light as a Continuous Flow

Imagine standing at a crossroads, looking north. You count the cars that come to the crossroads from the north, and observe that 60 cars arrive in the course of an hour. You report that the arrival rate for cars is 60 per hour, and this lets you guess that in 10 minutes, about 10 cars will arrive; in 5 minutes 5 cars will arrive, etc. Of course, at an actual crossroads, cars arrive irregularly, so your "5 cars in 5 minutes" claim is probably not exactly correct. Nonetheless, if you counted cars for each hour over the course of the day, you could make a graph like the one shown in Figure 26.16, where we've connected the dots with straight lines, but could have used a smooth curve. Later you could say something like "the arrival rate at 9:30 was about 65 cars per hour."

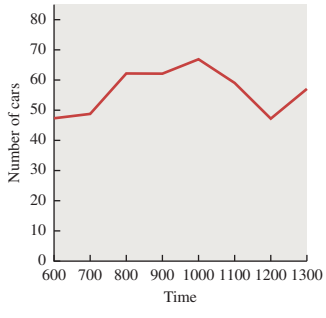


Figure 26.16: The number of cars arriving from the north at an intersection, at each hour.

In making such a statement, you are treating the arrival rate as something that makes sense *at a particular instant;* you are treating this problem as if it were

continuous rather than discrete so that the tools of calculus (e.g., instantaneous rates) actually apply to it, while in fact only finite-time rate measurements ("19 cars arrived between 9:20 and 9:43") make sense.

We'll do the same thing with light. As we observe the light arriving at a small piece of surface, we can think of ourselves as counting "arriving photons over some period of time." But instead we treat the light as if it were infinitely divisible, and talk about instantaneous rates of light arrival. In fact, rather than counting photons, we'll count the arriving energy, because photons of different wavelengths have different energies, but the idea remains the same.

This assumption that there *is* an instantaneous rate of energy arrival at a surface lets us use calculus to talk about light energy. We'll repeat this "limiting trick" twice more, once to establish a rate of arrival per area as we consider smaller and smaller areas, and again to consider the rate of energy arriving from a particular set of directions, divided by the size of that set of directions, as the size of the set goes to zero. Having described this quantity (which we'll call radiance), we'll see that all practical measurements we can make can be expressed as integrals of radiance over various areas, time periods, and sets of directions. The abstract entity, radiance, turns out to be easy to work with using calculus, and all the things we can measure are integrals of radiance.

In this discussion so far, we've moved from a discrete version of counting to one in which the light-energy arrival rate is continuous. We'll now do the same thing in two more ways, with respect to angle and area.

26.6.1 A Brief Introduction to Probability Densities

Before we do so, let's look at a related concept from probability theory, the notion of **probability density.** Consider a random number generator that randomly generates real numbers between 0 and 5. We observe 1,000 of these randomly generated real numbers, and look at how many lie between 0 and 0.5, between 0.5 and 1.0, etc. The resultant histogram (see Figure 26.17) looks fairly smooth; looking at it we might conjecture that the random number generator is *uniform,* in the sense that every number is equally likely to be generated. But if we choose smaller bins to count—say, between 0 and 0.001, between 0.0001 and 0.0002, etc.—the uniformity is no longer so obvious. Indeed, the probability of generating any *particular* random number must be zero. Thus, when we are discussing probabilities where the domain is some interval in the reals (rather than a discrete set, like the set of faces on a pair of dice), we talk not of the probabilities of generating particular numbers, but of generating numbers *within an interval* $[a, b]$. If the generator really is generating numbers uniformly, then the probability of generating a number in the interval $[a, b]$ is proportional to $b - a$. More generally, we posit the existence of a function $p : [0, 5] \to \mathbf{R}$ called the **probability density function** or **pdf** with the property that

$$\mathbf{Pr}\{\text{a random number in the interval } [a, b] \text{ is generated}\} = \int_a^b p(x)\, dx.$$

For the uniform distribution on the interval $[0, 5]$, p is the constant function with value $1/5$. For other distributions, p is not constant. But because its integral represents a probability, p must be everywhere nonnegative, and its integral over $[0, 5]$ must be 1.0.

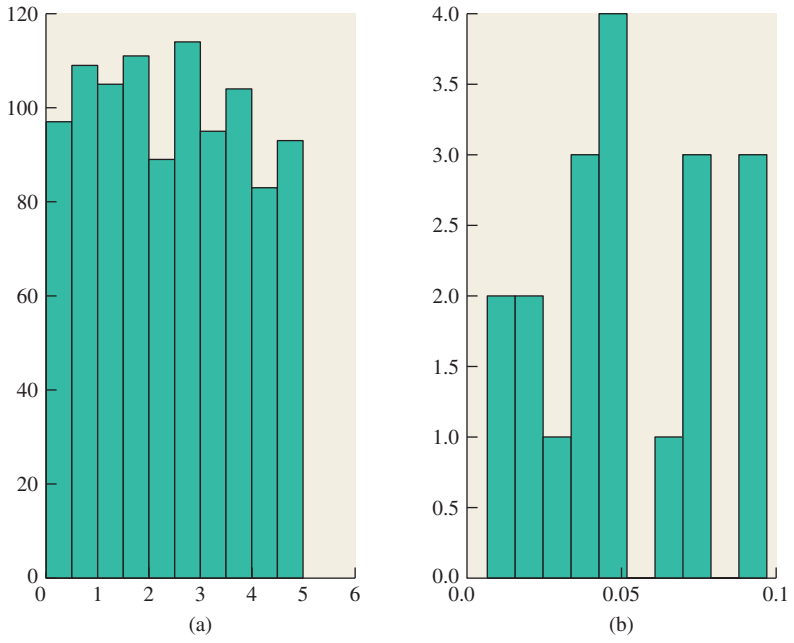


Figure 26.17: (a) A histogram of 1000 random numbers between 0 *and* 5*, in bins of width* $1/2$*; the distribution appears to be uniform. (b) A portion of a finer histogram, with bins of size* 0.01*; at this scale, it's not clear that the distribution is uniform.*

An alternative formulation for describing a distribution is the **cumulative distribution function** or **cdf,** defined by

$$F(u) = \mathbf{Pr}\{x \leq u\}. \tag{26.20}$$

If F is continuous and differentiable, then the two formulations are related by noting that $p = F'$. The big advantage of the cdf formulation is that probability *masses* can be incorporated easily, that is, single points in the real line at which a nonzero amount of probability is concentrated. At a point b where there is a probability mass, there's also a discontinuity in the cdf, with the "jump" being exactly the probability mass at b. The student who wishes to work carefully with the "impulses" that arise in the geometric optics description of mirror reflection and refraction, and which correspond exactly to the notion of probability masses, will do well to study the cdf approach to defining distributions.

Inline Exercise 26.9: Verify that the function $p(x) = \begin{cases} 2 & 0 \leq x \leq 1 \\ 0 & 1 < x \leq 2 \end{cases}$ is a probability distribution on $[0, 2]$. Notice that $p(0.5) = 2$, but this does not mean that the chance of picking 0.5 as a sample from this distribution is 2. We see from this example that while probabilities may not exceed 1.0, probability *densities* may.

26.6.2 Further Light Modeling

Now we return to the crossroads. Just as cars arrive from the north at a certain rate, other cars arrive from the south, the east, and the west. To adequately describe all the arriving cars requires you to keep multiple tallies, one for each arrival direction. If the crossroads were a more complex intersection, with five, or six, or ten roads leading into it, you'd need more and more tallies. If cars could arrive in *any* direction, then in the analogy with the probability densities we just discussed the probability of a car arriving from any *particular* direction would be zero. Instead, we'd have to talk about a density, where the probability of a car arriving from a *range* of directions was gotten by integrating the density over that range of directions.

Analogously, light energy can arrive at a point from any direction. The amount arriving from a range of directions depends in part on how large the range is: If you narrow the range of directions, you observe less incoming light energy. Indeed, if you narrow your range of directions to a *single* direction, no energy at all will arrive from that direction. We speak, therefore, of a **density,** where the amount of energy arriving in some range of directions is gotten by integrating this density over that range of directions.

Just as the energy from a single direction is zero, the energy arriving at any single *point* is also zero. To get something meaningful, we must consider the energy arriving over some small region. Once again, this is done with a density: We posit a function whose integral, over a small region,[1] gives the amount of energy arriving there.

All of this will be made more explicit in Section 26.7; for now the key idea is that our model of light moving around in a scene will be based on a density function whose arguments range over several continua: time, position, and direction.

26.6.3 Angles and Solid Angles

To define a "range of directions" for light arriving at a surface in 3-space, we need to define a notion of "solid angle" in $\mathbf{R^3}$ in analogy with the notion of angle in $\mathbf{R^2}$.

An angle in $\mathbf{R^2}$ is usually defined by a pair of rays at a point P (see Figure 26.18). If we look at a unit circle C around P, there's an arc A contained between the rays. The length of the arc A is the measure of the angle.

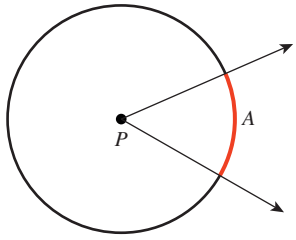


Figure 26.18: An angle at the point P.

We can revise this definition slightly, and say that the arc A *is* the angle. Clearly, if you know the arc A and the point P, you can find the two rays, and vice versa, so the distinction is a small one. But we can then generalize, and say that an angle at P is *any* subset[2] of the unit circle C at P. The measure of the angle is the total length of all the pieces of the subset. In practice, there are typically a finite number of pieces—usually just one—so this isn't a large generalization. Finally, it's often convenient to not talk about points of the circle C, but about points on the unit circle, or unit vectors. For any point X in C, we can form the unit vector $\mathbf{v} = X - P$. Given $\mathbf{v}$ and P, it's easy to recover $X = P + \mathbf{v}$. So our revised notion of an "angle at P" is this: An angle at P is either a subset of the unit circle C with center P, or a subset of the set $\mathbf{S^1}$ of all unit vectors.

1. We'll generally use the term "region" to indicate a portion of a surface, and the term "area" to indicate the size of that region (i.e., something whose units are m^2), although we'll occasionally use terms like "pixel area" to indicate a region.
2. ◇ Any *measurable* subset [Roy88]. See Chapter 30.

The notions of "clockwise" and "counterclockwise" angles, and "the angle from ray1 to ray2" (which might be much larger than π) and of angles that "wrap around multiple times" can all be defined with careful adjustments of the definition above; in our study of light, though, we'll have no need for these ideas, so we'll simply use the definition of angle and measure above.

One common use of angles is the notion of the angle **subtended** by some shape, T, at a point P (see Figure 26.19). The shape T is projected onto the unit circle C around P, and the measure of the resultant angle is called **the angle subtended by T at P.** In equations, the angle subtended by T at P is

$$\{\mathcal{S}(X - P) : X \in T\}. \tag{26.21}$$

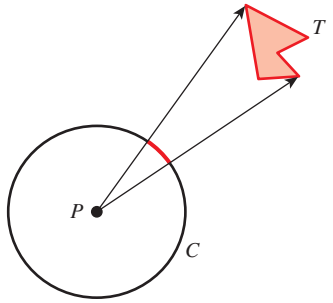


Figure 26.19: The angle subtended by T at P.

We can now describe solid angles in $\mathbf{R}^3$ by analogy. A **solid angle** at a point $P \in \mathbf{R}^3$ is a (measurable) subset Ω of the unit sphere about P, or, equivalently, a measurable subset of $\mathbf{S}^2$, the collection of all unit vectors in 3-space. The **measure of the solid angle** of Ω is the *area* of the set Ω (see Figure 26.20).

When we want to treat points in a solid angle as unit vectors, we'll use bold Greek letters, almost always using the letter ω. We'll often write "Let $\omega \in \Omega \ldots$", and thereafter treat ω as a unit vector, writing expressions like $\omega \cdot \mathbf{n}$ to compute the length of the projection of a vector $\mathbf{n}$ onto ω. In fact, this use of a solid angle as a collection of direction vectors is almost the only one we'll see.

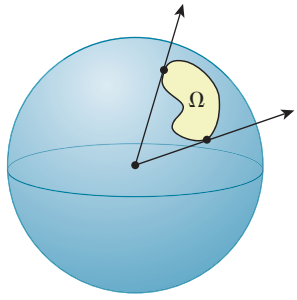


Figure 26.20: The solid angle Ω is a set on the surface of the unit sphere. The measure of the solid angle is the area of this set.

The notion of subtended angle can also be extended to three dimensions: If T is a shape in $\mathbf{R}^3$ and P a point of $\mathbf{R}^3$ with $P \notin T$, the **solid angle subtended by T at P** is the area of the radial projection of T onto the unit sphere at P, in exact analogy with the two-dimensional case. More precisely, the solid angle subtended by T at P is

$$\{\mathcal{S}(Q - P) : Q \in T\},$$

in exact analogy with the 2D case.

This definition lets us speak of "solid angles" on other spheres (e.g., like the Earth) by defining their measure to be the measure of the solid angle they subtend at the center of the sphere. It's easy to show that if U is a subset of a sphere of radius r about P, and the area of U is A, then the solid angle represented by U (i.e., the solid angle subtended by U at P) is A/r^2. When we speak of measuring a solid angle on some arbitrary sphere (like the Earth, or a spherical lightbulb), it is implicit that we mean "the solid angle subtended at the center of the sphere by this region."

Inline Exercise 26.10: Estimate the solid angle measure of your country as a solid angle on the (roughly) spherical earth. Use 13,000 km (or 8000 mi) as the diameter of the Earth.

Notation: It's conventional to use Ω to denote both a solid angle and the measure of that solid angle (just as we use θ to denote an angle and its measure in the plane). Just as we often use x as a variable of integration in calculus, it's common to use the letter Ω to denote a solid angle, and ω to denote a member of Ω, so that ω is a unit vector.

Units: Just as angles are measured in radians, solid angles are measured in **steradians,** abbreviated "sr." The entire unit sphere has a solid angle measure of 4π steradians.

26.6.4 Computations with Solid Angles

Let's now measure a few simple solid angles (see Figure 26.21). Following the standard from graphics rather than mathematics, we will treat the y-axis as pointing up, the x-axis as pointing to the right, and the z-axis as pointing toward us. Thus, the longitude is $\mathrm{atan2}(y,z)$ and the latitude is $\arcsin(y)$. (The **colatitude,** which is often denoted ϕ in spherical polar coordinates, is $\arccos(y)$. The other polar coordinate, θ, is what we've called the longitude.)

- If Ω is all of $\mathbf{S^2}$, then the measure of Ω is 4π (the area of a unit sphere).
- Any hemisphere has measure 2π.
- The "stripe" between $y = y_0$ and $y = y_1$ has area $2\pi\|y_1 - y_0\|$. This follows from the theorem below, as do the next two examples.
- The latitude-longitude rectangle between latitudes λ_0 and λ_1 and longitudes θ_0 and θ_1 has solid angle $\|\theta_1 - \theta_0\| \cdot \|\sin\lambda_1 - \sin\lambda_0\|$ (where latitude goes from $-\pi/2$ at the South Pole to $\pi/2$ at the North Pole). (When the longitudes are on opposite sides of the international dateline, this rectangle is a very long stripe wrapping around the nondateline part of the globe.)
- A "disk" consisting of all points whose spherical distance from a point P is less than r (where $r < \pi$) has solid angle measure $2\pi(1 - \cos(r))$.
- If a regular solid of n sides (cube $(n = 6)$, tetrahedron $(n = 4)$, octahedron $(n = 8)$, dodecahedron $(n = 12)$, icosahedron $(n = 20)$) is inscribed in the unit sphere, the projection of one of its faces onto the sphere (see Figure 26.21 (f)) has solid angle measure $\frac{4\pi}{n}$, because the total projected area is 4π, and by symmetry, each face has the same projected area.

All of the results above are consequences of the **sphere-to-cylinder projection theorem:** If C is a cylinder of radius 1 and height 2, circumscribed about the sphere S of radius 1, then the horizontal radial projection map,

$$p : C \to S : (x,y,z) \mapsto \left(\frac{x}{\sqrt{x^2+z^2}}, y, \frac{z}{\sqrt{x^2+z^2}}\right), \tag{26.22}$$

is area-preserving. (The proof is a simple calculus computation—see Exercise 26.1). Figure 26.22 shows this: The area of a country on the surface of the globe is the same as the area on the *plate carrée* projection shown (although many other characteristics of shape are grossly distorted, as shown for Greenland [in green]).

As an example of another use of this theorem, let's let Ω denote the northern hemisphere $y \geq 0$ of the unit sphere, and integrate the function y over this hemisphere.

That is to say, we seek to evaluate

$$B = \int_{(x,y,z)\in\Omega} y\, dA. \tag{26.23}$$

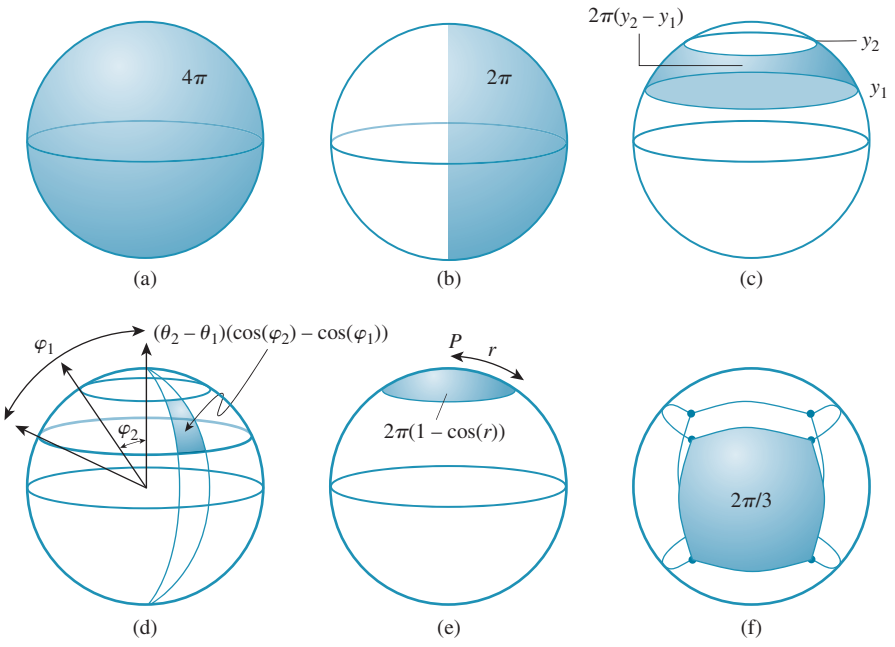


Figure 26.21: Various solid angles on the unit sphere.

Consider the upper half-cylinder $H = \{(x, y, z) : x^2 + z^2 = 1, 0 \leq y \leq 1\}$, which projects to Ω under the axial projection map p. We can perform a change of variables in the integral, and express B as

$$B = \int_{(x',y',z') \in H} y \, |Jp(x', y', z')| \, dA', \tag{26.24}$$

where $(x, y, z) = p(x', y', z')$, and dA' is area on H, and $|Jp|$ is the Jacobian for the change of variables (i.e., it represents how areas at (x', y', z') are stretched or contracted to become areas at (x, y, z)). The theorem that p is area-preserving means that $|Jp| = 1$, so the integral becomes

$$B = \int_{(x',y',z') \in H} y \, dA'. \tag{26.25}$$

Since in the formula for p, y does not change, we have $y = y'$, so this becomes

$$B = \int_{(x',y',z') \in H} y' \, dA'. \tag{26.26}$$

By circular symmetry, this is just 2π times the integral of y' from 0 to 1. That integral is $1/2$, so $B = \frac{1}{2} \cdot 2\pi = \pi$.

If instead we wanted to know the *average* of y over the upper hemisphere, we'd need to divide its integral (π) by the area of the hemisphere (2π). The average is thus $\frac{1}{2}$. This value comes up often, although it's usually in a slightly generalized form: We have a hemisphere defined by $\omega \cdot \mathbf{n} \geq 0$, and we want to know the average value of $\omega \cdot \mathbf{n}$ over this hemisphere. (Our instance is the special case where $\mathbf{n} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^{\mathbf{T}}$.) We'll state this as a principle:

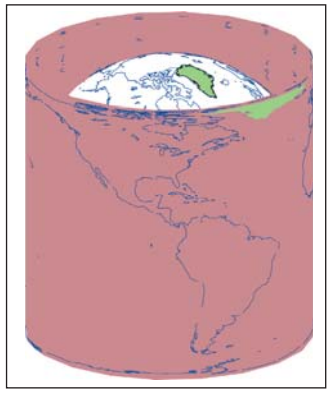


Figure 26.22: Horizontal radial projection from the sphere to the surrounding cylinder is area-preserving.

✓ **THE AVERAGE HEIGHT PRINCIPLE:** The average height of a point on the upper hemisphere of the unit sphere is $\frac{1}{2}$. Thus, for any unit vector $\mathbf{n}$, the integral

$$\int_{\{\omega \in S^2 : \omega \cdot \mathbf{n} \geq 0\}} \omega \cdot \mathbf{n} \, d\omega = \pi. \tag{26.27}$$

Inline Exercise 26.11: In various computations, one has both a solid angle Ω in the sphere S around a point P, *and* a surface M containing P, which can be locally thought of as a plane K through P (namely, the tangent plane to M at P). The **projected solid angle** Ω' is the area of the projection of Ω onto the plane K (see Figure 26.23).
(a) What is the largest possible projected solid angle for any solid angle Ω in a hemisphere bounded on one side by the plane K?
(b) For the case where P is the origin and K is the xz-plane, compute the projected solid angle of the "positive x quadrant" (the points of S with $x, y \geq 0$).
(c) Do the same for the region consisting of all points with latitude greater than 30° north (i.e., approximately the northern extra-tropical zone).
(d) Show that the solid angles of the two regions are the same.
(e) Explain why the projected solid angles are different.
(f) Compute the projected solid angle of the region $\theta_0 \leq \theta \leq \theta_1$, $\phi_0 \leq \phi \leq \phi_1$, where ϕ_0 and ϕ_1 are both between 0 and $\pi/2$, that is, the projected solid angle of a small latitude-longitude patch in the upper hemisphere. Hint: You should be able to answer every part of this question without computing any integrals; the sphere-to-cylinder projection theorem will help.

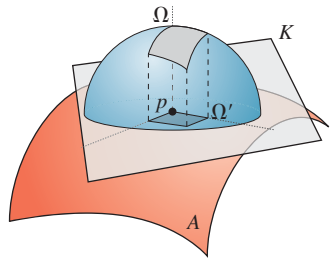


Figure 26.23: For a solid angle Ω in the unit sphere around a point p of a surface M, the projected solid angle lies in a plane K tangent to M at p. The projected solid angle Ω' will always have a smaller area than the original solid angle.

26.6.5 An Important Change of Variables

Often in the next several chapters we'll have occasion to integrate some function over the solid angle Ω subtended by some rectangle R, of width w and height h, at a point P, as shown in Figure 26.24. Usually this function involves a factor of $\omega_i \cdot \mathbf{n}$, where $\mathbf{n}$ is the surface normal at P and $\omega_i \in \Omega$ is the variable of integration, in which case the integral looks like

$$A = \int_{\omega_i \in \Omega} g(\omega_i) \, \omega_i \cdot \mathbf{n} \, d\omega_i, \tag{26.28}$$

In some cases involving transparency, the $\omega_i \cdot \mathbf{n}$ factor will be negative and will require absolute value signs.

Expressing Ω in terms of latitude and longitude, or even in terms of xyz-coordinates, may be extremely messy. It's often convenient to perform a change of variables instead, and integrate over the rectangle R. We'll carry this out for the particular case where P is the origin so that the mapping from a point (x, y, z) on R to a point on the unit sphere has a particularly nice form:

$$N(x, y, z) = \frac{1}{\sqrt{x^2 + y^2 + z^2}}(x, y, z). \tag{26.29}$$

(We've chosen the letter N here for "normalize.")

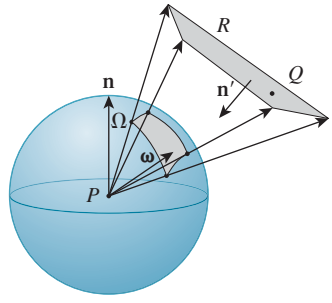


Figure 26.24: Notation for the change of variables.

The change of variables formula says that to compute

$$A = \int_{\omega_i \in \Omega} g(\omega_i)\ \omega_i \cdot \mathbf{n}\ d\omega \tag{26.30}$$

we can instead compute

$$A = \int_{Q \in R} g(N(Q))\ N(Q) \cdot \mathbf{n}\ |JN(Q)|\ dQ, \tag{26.31}$$

where JN is the Jacobian for the change of variables N.

Let's suppose that the rectangle R is specified by a corner, C, and two perpendicular unit vectors $\mathbf{u}$ and $\mathbf{v}$, chosen so their cross product $\mathbf{n}'$ points back toward P. The points of R are then of the form

$$Q = C + s\mathbf{u} + t\mathbf{v},$$

where $0 \le s \le w$ and $0 \le t \le h$. So the integral we need to compute is

$$A = \int_{s=0}^{w} \int_{t=0}^{h} g(N(C + s\mathbf{u} + t\mathbf{v}))\ N(Q) \cdot \mathbf{n}\ |JN(C + s\mathbf{u} + t\mathbf{v})|\ dt\ ds. \tag{26.32}$$

Computing the Jacobian of N at the point $Q = C + s\mathbf{u} + t\mathbf{v}$ is somewhat involved, but the end result is simple:

$$|JN(Q) = \frac{|\omega \cdot \mathbf{n}'|}{r^2}, \tag{26.33}$$

where r is the distance from P to Q and ω is the unit vector pointing from P to Q.

The intuitive explanation for this is that if the plane of the rectangle R happened to be perpendicular to ω, then a tiny rectangle on R, when projected down to the unit sphere around P, would be scaled down by a factor of r in both width and height, and that accounts for the r^2 in the denominator. If the plane of R is tilted relative to ω, then we can first project the tiny rectangle onto a plane that's *not* tilted (projecting along ω). This, by the Tilting principle, introduces a cosine factor, which is $\omega \cdot \mathbf{n}'$.

Applying this result to the point $Q(s,t) = C + s\mathbf{u} + t\mathbf{v}$, the integral A becomes

$$A = \int_{s=0}^{w} \int_{t=0}^{h} g(N(Q(s,t)))\ N(Q(s,t)) \cdot \mathbf{n}\ \frac{|(Q(s,t) - P) \cdot \mathbf{n}'|}{\|Q(s,t) - P\|^3}\ dt\ ds \tag{26.34}$$

$$= \int_{s=0}^{w} \int_{t=0}^{h} g(N(Q(s,t)))\ \frac{|(Q(s,t) - P) \cdot \mathbf{n}|}{\|Q(s,t) - P\|}\ \frac{|(Q(s,t) - P) \cdot \mathbf{n}'|}{\|Q(s,t) - P\|^3}\ dt\ ds \tag{26.35}$$

$$= \int_{s=0}^{w} \int_{t=0}^{h} g(N(Q(s,t)))\ \frac{|(Q(s,t) - P) \cdot \mathbf{n}|\ |(Q(s,t) - P) \cdot \mathbf{n}'|}{\|Q(s,t) - P\|^4}\ dt\ ds. \tag{26.36}$$

If we define $\omega(s,t) = \frac{Q(s,t) - P}{\|Q(s,t) - P\|}$, this simplifies to

$$A = \int_{s=0}^{w} \int_{t=0}^{h} g(\omega(s,t))\ \frac{|\omega(s,t) \cdot \mathbf{n}|\ |\omega(s,t) \cdot \mathbf{n}'|}{\|Q(s,t) - P\|^2}\ dt\ ds. \tag{26.37}$$

To make this concrete, Listing 26.1 shows how you might actually estimate this integral numerically, given the function g that takes a unit vector as an argument.

Listing 26.1: Integrating a cosine-weighted function over the solid angle subtended by a light source.

```
// Given rectangle information C (corner), u, v (unit edge vectors),
// w, h (width and height) and n' (unit normal), a point P on
// a plane whose normal is n, and a function g(.) of a single
// unit-vector argument, estimate the
// integral of g(ω)ω·n over the set Ω of
// directions from P to points on the rectangle.


sum = 0;
for i = 0 to N-1
   s = i/(N-1)
   Δs = 1/(N-1);
   for j = 0 to N-1
      t = j/(N-1)
      Δt = 1/(N-1)
      Q = C + s * u + t * v
      ω = S(Q − P)
      r = ||Q − P||

      sum += g(ω) |ω·n| |ω·n'| / r^2 Δs Δt

return sum
```

To summarize, when we change from an integral over solid angles to an integral over some planar surface with normal $\mathbf{n}'$, we introduce an extra factor in the integrand, of the form $\frac{|\omega \cdot \mathbf{n}'|}{r^2}$, where ω is the unit vector from P to a point Q on the surface and r is the distance from P to Q. Often the integrand will already have the form $g(\omega)|\omega \cdot \mathbf{n}|$, so the integrand for the area integral will be

$$g(\omega)\,\frac{|\omega \cdot \mathbf{n}|\,|\omega \cdot \mathbf{n}'|}{r^2}. \tag{26.38}$$

26.7 Measuring Light

With the notion of solid angle in hand, we can now precisely describe how light energy is flowing in a scene. We'll consider a function L, called the **spectral radiance.** It's a function of time, position, direction, and wavelength that captures the infinitesimal characteristics of light transport in the sense that when it's integrated over a time interval, and over some part of a surface perpendicular to the direction of transport, and over some solid angle of directions, and over some range of wavelengths, the result is the total light energy that arrives at that surface, arriving from the specified directions, within the range of wavelengths, and during the time interval. We previously discussed summing up energies for all different wavelengths, and we'll do that presently, but for now, we want to consider the per-wavelength function—yet another density!

The **integral** of spectral radiance over a small surface, and a small range of directions, and a small period of time, and a small range of wavelengths, is the sort of thing that can be measured by a physical device, while the infinitesimal

version is the thing that's easy to work with mathematically, just as it's possible to measure the distance a car travels over some small time interval, but we work with instantaneous velocity when we're studying the mathematics of motion. What are the units of L? Taking as a model "piece of surface" a rectangle in the xy-plane, and assuming that the light flow is in a set of directions Ω all of which are essentially perpendicular to the xy-plane, we know that

$$\textit{energy} \approx \int_{t_0}^{t_1} \int_{x_0}^{x_1} \int_{y_0}^{y_1} \int_{\omega\in\Omega} \int_{\lambda_0}^{\lambda_1} L(t,(x,y,0),-\omega,\lambda)\, d\lambda\, d\omega\, dy\, dx\, dt. \quad (26.39)$$

Note that in the integrand above, L has four arguments: t, the point $(x, y, 0)$, $-\omega$, and λ. The ω is negated because ω points *out* of the surface, but we want to sum up the light coming *in* to the surface.

Using MKS units for the surface and time, but nanometers for wavelength (which follows long-standing convention), we find that L must have the units of joules per second per square meter per nanometer per steradian. One joule per second is one watt, so we can also say "watts per square-meter nanometer steradian."

What happens if the direction ω along which light arrives is *not* parallel to the surface normal? Then the amount of light energy arriving at the surface, per unit area, is smaller than if it *were* parallel, by the Tilting principle.

Thus, the more general and exact formula for the energy arriving at that small region of the xy-plane from directions in the solid angle Ω, in the given time interval and wavelength interval, is

$$\textit{energy} = \int_{t_0}^{t_1} \int_{x_0}^{x_1} \int_{y_0}^{y_1} \int_{\omega\in\Omega} \int_{\lambda_0}^{\lambda_1} L(t,(x,y,0),-\omega,\lambda)\, \omega\cdot\mathbf{e}_3\, d\lambda\, d\omega\, dy\, dx\, dt. \quad (26.40)$$

For a region R of an arbitrary plane, with normal vector $\mathbf{n}$, the energy arriving at R in the interval $t_0 \le t \le t_1$, at wavelengths $\lambda_0 \le \lambda \le \lambda_1$, in directions opposite those in a solid angle Ω, is

$$\textit{energy} = \int_{t_0}^{t_1} \int_{\lambda_0}^{\lambda_1} \int_{P\in R} \int_{\omega\in\Omega} L(t,P,-\omega,\lambda)\, |\omega\cdot\mathbf{n}|\, d\omega\, dP\, d\lambda\, dt, \quad (26.41)$$

where $\int_{P\in R} \ldots dP$ is an area integral over the area R.

When we are concerned with overall light energy, rather than caring about how much is transported at each different wavelength, we can integrate L over *all* wavelengths λ, giving us a new function depending on time, position, and direction, with units of watts per square-meter steradian. This new function is called the **radiance.** This relationship between spectral radiance and radiance is quite general: For any photometric quantity, the spectral version has the wavelength λ as a parameter, while the version without the adjective "spectral" has been integrated over all possible wavelengths.

The function L, defined for all times and points and all directions (and possibly for all wavelengths) describes fully the way light flows around the scene. We'll call $L(t, P, \omega)$ or $L(t, P, \omega, \lambda)$ the "radiance" or "spectral radiance" at time t, location P, etc. But the function L, considered as a whole, is sometimes also called the **plenoptic function,** particularly in computer vision.

26.7.1 Radiometric Terms

The spectral radiance L characterizes the light energy flowing, at each instant, at each point in the world, in each possible direction. Formally its domain is

$$\mathbf{R} \times \mathbf{R^3} \times \mathbf{S^2} \times \mathbf{R}^+,$$

where $\mathbf{S^2}$ denotes the unit sphere in 3-space (the set of all possible directions in which light can flow) and $\mathbf{R}^+$ is used for the set of all possible wavelengths. In practice, $\mathbf{R}^+$ may be replaced by the range of wavelengths that are visible. The codomain of L is $\mathbf{R}$.

Starting from L, we can describe, via integration, all of the terms conventionally used in **radiometry,** the science of the measurement of radiant energy. An alternative approach is to start from energy or power, and define all the terms by differentiation. We discuss this approach briefly in Section 26.9.

26.7.2 Radiance

Spectral radiance is the quantity described by L; radiance is the quantity

$$\int_0^\infty L(t, P, \omega, \lambda)\, d\lambda, \tag{26.42}$$

which is defined for $(t, P, \omega) \in \mathbf{R} \times \mathbf{R^3} \times \mathbf{S^2}$. In engineering, the letter L is usually used for this quantity, with L_λ being reserved for spectral radiance; in graphics, however, the spectral radiance is often denoted by L. Because for us the symbol λ actually is one of the arguments to the function, it's a bad choice for a subscript. We'll therefore carry out the remainder of this discussion in the spectral case (keeping λ as an argument), and discuss the nonspectral case at the end. Until then, when we speak of radiance we'll be speaking of spectral radiance; when we speak of irradiance we'll mean spectral irradiance, etc.

The most interesting thing about radiance, from a computer graphics point of view, is that in a steady-state situation, that is, one in which L is independent of t, radiance is constant along rays in empty space (assuming, for the moment, that there are no point light sources; see Exercise 26.3). In mathematical terms, this means that the function L cannot be just any function. We also know, from physical considerations, that L can never be negative.

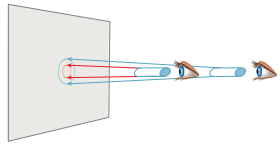


Figure 26.25: A radiance measurement tool.

Why is L constant along rays in empty space? Try an experiment (see Figure 26.25): Look through a narrow cardboard tube at a tiny region of a well-lit latex-painted wall. You'll see a small disk of light at the end of the tube, outlined in red in the figure. Now move twice as far away from the wall, and look again at the same region. Again, you'll see a small disk of light (outlined by the larger blue circle), and it will appear *equally bright* (assuming that the wall is about equally well lit over the region where you're looking). There's an easy explanation for this: When at first you were at distance r from the wall, light leaving the wall spread out to illuminate a hemisphere of radius r; when you move to distance $2r$, it's illuminating a hemisphere of radius $2r$, whose area is four times as great. But as you look through your tube, you see four times as large a region of the wall. Hence the total energy coming down the tube toward your eye is constant. In each case, the light energy passing through the eye end of the tube is approximately the integral of the radiance over the region of the tube end. Because we're assuming the wall is uniformly lit, this is just the (approximately constant) radiance times

the area of the tube end times the solid angle subtended at the eye by the far end of the tube. The fact that things look the same means that the radiance has not changed as you moved along the ray from the wall to your initial eyepoint.

To the degree that the pixel area on the sensor in a digital camera can be regarded as infinitesimal (or that the variation of radiance with position on the sensor can be assumed small) and the solid angle of rays that hit that pixel can be considered infinitesimal (or the variation with direction assumed small), the response of the sensor (assuming it responds to total arriving energy) is proportional to radiance. In fact, high-quality cameras used for computer-vision experiments produce images whose individual entries are radiance values. To be more precise, the values they produce are often integrals, with respect to wavelength, of spectral radiance multiplied by a response function that characterizes how the sensor responds to radiance of each wavelength.

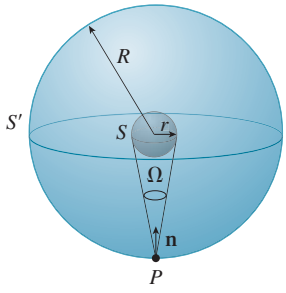


Figure 26.26: A radiating sphere inside a large receiving sphere. We'll compute arriving power density at P.

26.7.3 Two Radiance Computations

For a **Lambertian emitter,** the radiance in all outgoing directions is the same. Let's suppose that we have a Lambertian emitting sphere S of radius r, emitting total power Φ. We'll now compute the radiance along each ray leaving the sphere. The idea (see Figure 26.26) is to surround the emitter with a concentric sphere S' of radius $R >> r$. All power emitted from S must arrive at S', and the arriving power density (in $\mathrm{W\,m^{-2}}$) on S' is independent of position. If we call this density D, then

$$4\pi R^2 D = \Phi. \tag{26.43}$$

We'll compute the power density at the point P in terms of the unknown constant emitted radiance L, which will allow us to solve for L in terms of Φ. The power density at P is

$$D = \int_{\mathbf{S}^2_+(P)} L\,|\omega \cdot \mathbf{n}(P)|\,d\omega \tag{26.44}$$

$$= \int_{\Omega} L\,|\omega \cdot \mathbf{n}(P)|\,d\omega \tag{26.45}$$

$$= L \int_{\Omega} |\omega \cdot \mathbf{n}(P)|\,d\omega. \tag{26.46}$$

The transition between Equations 26.44 and 26.45 is justified by noting that for ω outside Ω, the radiance at P in direction $-\omega$ is zero, so the integral over the whole hemisphere can be reduced to an integral over just Ω.

For sufficiently large values of R, $\omega \cdot \mathbf{n}(P)$ is very close to 1, so in the limit as R approaches infinity, we get

$$D = L \int_{\Omega} 1\,d\omega = L\,m(\Omega). \tag{26.47}$$

The sphere of radius R around P (drawn in light gray in Figure 26.27) has total area $4\pi R^2$, and subtends a solid angle of 4π at P; of this $4\pi R^2$ area, an area of approximately πr^2 is occupied by the radiating sphere (i.e., as seen from P, the radiating sphere occludes a disk of area πr^2 in the entire $4\pi R^2$). The solid angle

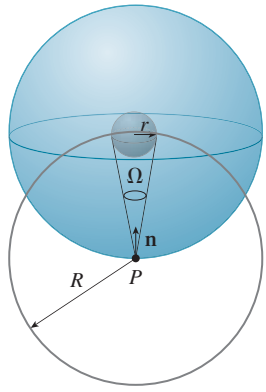


Figure 26.27: Computing the measure of the solid angle Ω.

subtended by the small sphere is therefore

$$m(\Omega) = 4\pi \frac{\pi r^2}{4\pi R^2} \tag{26.48}$$

$$= \frac{\pi r^2}{R^2}. \tag{26.49}$$

Substituting this in Equations 26.47 and 26.43, we get

$$4\pi R^2 L \frac{\pi r^2}{R^2} = \Phi, \text{ so} \tag{26.50}$$

$$L = \frac{\Phi}{4\pi(\pi r^2)}. \tag{26.51}$$

We've made two approximations in this computation: that the dot product is always 1, and that the area occluded by the emitting sphere is πr^2. If you instead evaluate the integral exactly, you'll see that the two approximations exactly cancel each other.

We'll now consider a similar example and analysis. This time we have a small disk-shaped emitter of radius r, that emits light only on one side (see Figure 26.28).

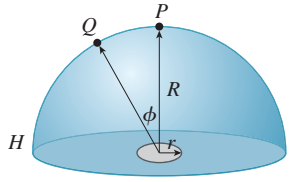


Figure 26.28: A Lambertian emitting disk that radiates on one side only.

We enclose it in a hemisphere H of radius R, and first compute the power density at the North Pole P just as before; once again the power density is

$$D_P = Lm(\Omega) = L\frac{\pi r^2}{R^2}. \tag{26.52}$$

At a point like Q that's off-axis by the angle ϕ, the Tilting principle applies, and the power density arriving at Q is only $\cos\phi$ times that arriving at P. Thus, the total power arriving at all points of the hemisphere (which must be the total emitted power Φ) is

$$\Phi = \int_H (\cos\phi) L \frac{\pi r^2}{R^2} \tag{26.53}$$

$$= L\frac{\pi r^2}{R^2} \int_H \cos\phi \tag{26.54}$$

by pulling the constant out of the integral. Further simplifying,

$$\Phi = L\frac{\pi r^2}{R^2} R^2 \int_{\mathbf{S}^2_+} \cos\phi \tag{26.55}$$

$$= L\pi r^2 \int_{\mathbf{S}^2_+} \cos\phi \tag{26.56}$$

because the area of H is R^2 times that of $\mathbf{S}^2_+$. Finally, by the Average height principle, we get

$$\Phi = L\pi r^2 \pi \tag{26.57}$$

so that

$$L = \frac{\Phi}{\pi(\pi r^2)}. \tag{26.58}$$

This generalizes to arbitrary emitter shapes; in general, a one-sided planar Lambertian emitter of power Φ and area A emits light of radiance

$$L = \frac{\Phi}{\pi A}. \tag{26.59}$$

26.7.4 Irradiance

Irradiance is the density (with respect to area, time, and wavelength) of the light energy arriving at a surface from all directions. (It's often described as the measure of the light hitting a surface "independent of direction," but if the light energy *varies* with direction, then the meaning of that phrase isn't entirely clear.) Irradiance is a useful notion when the surface is known to respond to incoming light in a way that's direction-independent, where "respond to" might mean "absorb" or "reflect." In cases where the response is direction-dependent, irradiance is generally irrelevant: Knowing only the total light energy hitting a surface will tell you nothing certain about the reflected energy.

Irradiance is usually defined only for a point P on a surface in the scene (or on a surface of some sensor like that of a virtual camera), and typically for a point where only *reflective* scattering takes place, that is, where there's no transmission through the surface, so we need only consider light arriving from one side of the surface. Equation 26.42 says that the energy arriving at a region R from directions opposite those in a solid angle Ω is

$$\textit{energy} = \int_{t_0}^{t_1} \int_{\lambda_0}^{\lambda_1} \int_{P \in R} \int_{\omega \in \Omega} L(t, P, -\omega, \lambda)\, |\omega \cdot \mathbf{n}|\, d\omega\, dP\, d\lambda\, dt. \tag{26.60}$$

The solid angle that interests us is $\mathbf{S}^2_+(P) = \{\omega : \omega \cdot \mathbf{n}(P) \geq 0\}$, the set of all outgoing directions at P. So the irradiance at a point P where the surface normal is $\mathbf{n}$ is the innermost integral, using $\mathbf{S}^2_+(P)$ as Ω. Within that integral, the dot product is always positive, so we can drop the absolute value signs,

$$E(t, P, \lambda) = \int_{\omega_\mathrm{i} \in \mathbf{S}^2_+(P)} L(t, P, -\omega_\mathrm{i}, \lambda)\, \omega_\mathrm{i} \cdot \mathbf{n}\, d\omega_\mathrm{i}, \tag{26.61}$$

where we've substituted ω_i for ω (see Figure 26.29).

This definition introduces some notational conventions we'll follow for the next several chapters. First, P typically denotes a point on some surface in the

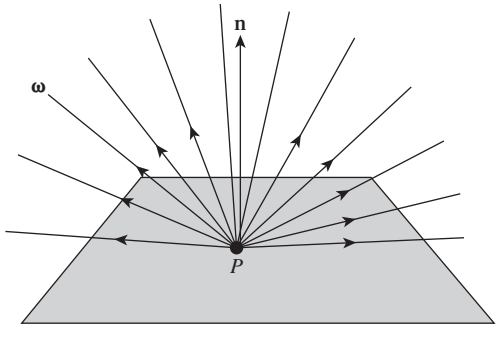


Figure 26.29: Notation for irradiance definition.

scene, while $\mathbf{n}(P)$ denotes the unit surface normal at P. The set of all "outward" vectors at P is $\mathbf{S}^2_+(P)$, that is,

$$\mathbf{S}^2_+(P) = \{\omega : \omega \cdot \mathbf{n}(P) \geq 0\}. \tag{26.62}$$

We'll sometimes generalize this and write

$$\mathbf{S}^2_+(\mathbf{n}) = \{\omega : \omega \cdot \mathbf{n} \geq 0\} \tag{26.63}$$

for the "positive hemisphere with respect to some vector $\mathbf{n}$."

Second, ω_i is a vector pointing in this outward hemisphere *toward* some source of light, so the radiance reaching P from this source is $L(t, P, -\omega_i, \lambda)$. The negative sign is important: ω_i points toward the light source, but light flows *from* the source *toward* P, hence in direction $-\omega_i$. The "i" in ω_i is a mnemonic for "incoming" rather than an index, which is why it is typeset in roman rather than italic. We'll also often use ω_o, a direction in which light *leaves* from P (often toward the observer's eye).

The units of spectral irradiance are J m^{-1} s^{-1} nm^{-1} or W m^{-2} nm^{-1}; the steradians have been integrated out.

Looking at the formula for irradiance, it becomes clear that the surface at which the light is arriving is not really important—only the surface point and normal vector are. Thus, we can regard irradiance instead as a function on *all* of $\mathbf{R^3}$, but with an additional argument to indicate the normal direction:

$$E(t, P, \mathbf{n}, \lambda) = \int_{\{\omega\,:\,\omega_i \cdot \mathbf{n} \geq 0\}} L(t, P, -\omega_i, \lambda)\, \omega_i \cdot \mathbf{n}\, d\omega_i. \tag{26.64}$$

With this revised formulation, the domain of E is $\mathbf{R} \times \mathbf{R^3} \times \mathbf{S^2} \times \mathbf{R}^+$; its interpretation is that $E(t, P, \mathbf{n}, \lambda)$ is the density of energy that *would* arrive at a surface perpendicular to $\mathbf{n}$ at the point P from all directions in the half-space determined by $\mathbf{n}$, and having wavelength λ.

It's often useful to speak of the **irradiance due to a single source.** To define this, we imagine painting everything in the scene completely black except the single source. We then use the resultant radiance field $\bar{L}$ in place of L in Equation 26.64.

In the event that we are measuring the irradiance due to a single area light source of constant radiance, L_0, that is, the radiance leaving every point of the light source in any direction is the number L_0, and the solid angle subtended by the light source lies at approximately a single latitude (if, for example, it's approximately disk-shaped and quite small), and it's completely visible from the point P, then the integral can be well approximated by assuming that the dot product $\omega_i \cdot \mathbf{n}$ is constant; since all other terms in the integral are constant as well, we can evaluate this approximation directly.

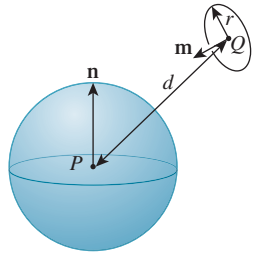


Figure 26.30: The rule of five.

Inline Exercise 26.12: Show that if, in Figure 26.30, the distance from a disk-shaped uniform light source of radius r, center Q, normal vector $\mathbf{m}$, and radiance L_0 to the point P is more than $5r$, and the source is completely visible from P, then the irradiance at P from the light is well approximated by

$$\pi r^2 L_0 \frac{(Q - P) \cdot \mathbf{n}\ (P - Q) \cdot \mathbf{m}}{\|Q - P\|^4}. \tag{26.65}$$

This is sometimes called the **rule of five.**

The notion of irradiance appears in many research papers about rendering, and it's often given the letter E. Except for brief mention in our discussion of radiosity in Chapter 31, we'll have no further use for irradiance, and we will use the letter E primarily to denote the eyepoint (or camera) in a rendering algorithm.

26.7.5 Radiant Exitance

The corresponding measure of light leaving a surface in all possible directions is called **spectral radiant exitance;** the only difference is the direction of the vector ω_o appearing in L:

$$Exitance = M(t, P, \lambda) = \int_{\omega_o \in \mathbf{S}^2_+(P)} L(t, p, \omega_o, \lambda)\omega_o \cdot \mathbf{n} \; d\omega_o. \tag{26.66}$$

Once again, we're defining this for reflective-only surfaces. And we can again extend this to be defined at any point in space: As long as we provide an additional argument indicating a surface normal, and by integrating over all wavelengths, we get the **radiant exitance.**

26.7.6 Radiant Power or Radiant Flux

The **radiant power** or **radiant flux** Φ arriving at a surface $\mathcal{M}$ (whether it's an actual surface in the scene or some virtual surface like "the surface of a sphere of radius 1 m surrounding this light source") is computed by integrating yet again. Since power is measured in joules per second, we must integrate over a region to remove the m^2 from the units:

$$Power = \Phi = \int_{P \in \mathcal{M}} \int_{\omega_i \in \mathbf{S}^2_+(P)} L(t, P, -\omega_i, \lambda)\omega_i \cdot \mathbf{n} \, d\omega \, dP. \tag{26.67}$$

The units of (spectral) power are $\mathrm{J\,s^{-1}\,nm^{-1}}$; those of power (arrived at by integrating out wavelength) are $\mathrm{J/\,sec}$, that is, W.

The meaning of "power" is only well defined when the surface $\mathcal{M}$ over which we are integrating is specified (along with the time and the wavelength).

For an imaginary surface in space, like the sphere surrounding the light source above, the power arriving at one side of the surface and the power leaving the opposite side are the same; for an actual surface in the scene, the power arriving at one side of a surface may be large, but for opaque surfaces, no power leaves the other side, although usually a lot is reflected.

To define radiant flux for a surface that both reflects and transmits, we need to extend the domain of integration to all of $\mathbf{S^2}$, and place absolute values on the dot product:

$$Power = \Phi = \int_{P \in \mathcal{M}} \int_{\omega_i \in \mathbf{S^2})} L(t, P, -\omega_i, \lambda) \; |\omega_i \cdot \mathbf{n}| \; d\omega \, dP. \tag{26.68}$$

What is the domain of the "Power" function? Certainly time and wavelength are still arguments, but what about the surface at which the power is arriving? One possible answer is that $\mathcal{M}$, the thing over which the integral is computed, can be any measurable subset of any surface in 3-space. (There's no standard name for the set of all such subsets). Most books simply ignore the question, and speak of the "radiant flux Φ," whose domain is ignored. We'll return to this briefly in Section 26.9.

26.8 Other Measurements

All radiometric quantities can be expressed as integrals of L: For the spectral radiant intensity, which we mentioned briefly above, with units of watts per steradian, you integrate out area and wavelength. For the nonspectral version, you also integrate out wavelength. **Radiosity** is a name sometimes used in graphics for a nonspectral radiant exitance; its units are watts per square meter (one integrates out wavelength and directions).

Terms like these (and like "irradiance" and "radiant exitance") are useful when we try to describe the flow of light energy in a scene with an approximation in which we aggregate light in various ways. For example, if our scene consists entirely of diffuse reflectors (e.g., items painted with a coat of latex paint), then it makes sense to do computations that ignore the direction in which light is radiated, and simply compute the total light energy radiated from a surface. In the same way, we often aggregate light into three wavelength groups, which we call "red," "green," and "blue," so that instead of computing the light transport individually for every possible wavelength λ, we simply compute it for three aggregates. This results in approximations of the correct results, but in many cases the approximations can be very good. Indeed, it's often worth writing the light energy leaving a surface as a sum of terms, each of which can be studied by a suitable algorithm; in some of these algorithms summary representations of the light may be appropriate, whereas in others the highly detailed representation provided by the radiance field L is more appropriate.

There are also other quantities that describe aggregate properties of light in terms that are relevant to human perception; these lie in the domain of **photometry** and are discussed in Section 28.4.1.

Finally, there is a term that can cause considerable difficulty: **intensity.** Intensity occurs frequently in early graphics papers, but its meaning is rarely given precisely. It's probably best to read these papers with a modern eye and regard "intensity" as a proxy for "radiance," although there may be a cosine factor or two missing in any particular discussion. When we use the word "intensity," it's strictly informal, as in the sentence "When we increase the intensity of the lamp, the scene brightens."

26.9 The Derivative Approach

An alternative approach to defining radiometric terms is to take the radiant flux Φ as a starting point, and to derive all other quantities from it through a kind of "differentiation." For instance, we can look at a point P of some surface, and a region R on the surface with $P \in R$, and consider the light arriving at R from all possible directions, and the resultant power arriving at R, which we call $\Delta\Phi$. By dividing this power $\Delta\Phi$ by the area of R, ΔA, we get a power-per-area measurement,

$$\frac{\Delta\Phi}{\Delta A}. \tag{26.69}$$

If we imagine repeating this process for various regions R, each with a smaller and smaller area, but still containing the point P, we get a sequence of power-per-area measurements. The argument is then that these measurements have a limit, as the

area of the region R goes to zero, and we call this limit

$$E(t, P, \lambda) = \frac{d\Phi}{dA}, \tag{26.70}$$

the irradiance at P.

Before proceeding further with this approach, we ask that you carefully review the definition of the derivative. Typically when we write df/dx, we require that f be a function of a variable x, and that $\frac{1}{h}(f(x+h) - f(x))$ have a limit as $h \to 0$; this limit is called the derivative. In the formulation above, there's no "variable" A, and Φ is certainly not a function of A. We can repair this problem by saying, "Let $f(r)$ be the power arriving at a disk of radius r about P in the surface; the area of that disk is πr^2, and we can define $g(r) = f(r)/(\pi r^2)$, which represents the power arriving at the disk, per area. We then define $d\Phi/dA(P)$ to be $g'(0)$." But one then must ask, "Would the result have been the same if I'd used a family of shrinking squares rather than disks? What about other shapes? And is g obviously differentiable?" And for each new concept defined by a "derivative" like this one, one has to reconsider the corresponding questions. The integral formulation we have pursued makes one single assumption—the existence of an integrable spectral radiance function L—and everything else follows from that.

Having made this critique of the "derivative" formulation, we should also mention its advantages. One of these is that when you write

$$E = \frac{d\Phi}{dA} \tag{26.71}$$

you know that if you want to compute the power, Φ, you'll need to compute an integral,

$$\Phi = \int E \, dA \tag{26.72}$$

$$= \int \frac{d\Phi}{dA} \, dA, \tag{26.73}$$

with the obvious notion that "the dAs cancel." In our experience, the presence of various cosines makes this sort of computation fraught with peril. Our students routinely draw false conclusions by being insufficiently precise about what they mean in such derivations. On the other hand, once you have some experience with this notation, and have gotten past the usual mistakes, it's a great convenience.

To continue with the standard derivative description, the radiant exitance is also defined as a derivative of power with respect to area,

$$M = \frac{d\Phi}{dA}, \tag{26.74}$$

where this time Φ means the power *leaving* the surface rather than arriving there.

The radiant intensity (a term we haven't previously defined, and will not mention again) is the derivative of flux with respect to solid angle,

$$I = \frac{d\Phi}{d\omega}, \tag{26.75}$$

and radiance is the "radiant flux per unit solid angle per unit projected area" [Jen01],

$$L = \frac{d^2\Phi}{\cos(\theta)\, dA\, d\omega}, \tag{26.76}$$

where θ is the angle between ω and the surface normal. This description of L is then inverted to write

$$\Phi = \int_A \int_\Omega L(P, \omega)\, |\omega \cdot \mathbf{n}|\, d\omega\, dP. \tag{26.77}$$

The experienced reader knows how to read this: "The total power arriving at a region A with surface normal $\mathbf{n}$, along rays in the solid angle Ω, is given by Equation 26.77."

26.10 Reflectance

How can we model reflectance? We'd like to somehow capture the idea that light striking a bit of surface from some distant point may scatter in many different directions, and that light from many different directions may therefore contribute to the light leaving in some particular direction. The great insight is to realize that the process is additive: If we can measure how light from each *single* direction is scattered, we'll know how light from a whole collection of directions is scattered as well.

A **gonioreflectometer** is a device used to measure reflectance; in the most basic design, it consists of a tiny spotlight mounted so that it can move about on a spherical shell, and a tiny sensor that can also move about on the shell (see Figure 26.31). (More modern designs, like the one shown in Figure 26.32, rely on moving the sample stage in more directions, rather than both the source and detector.)

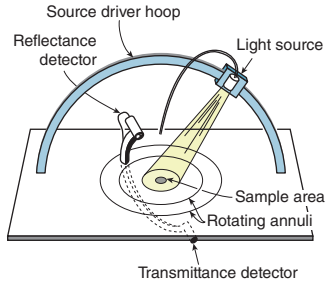


Figure 26.31: The basic idea of a gonioreflectometer (redrawn from [War92]).

The spotlight illuminates a sample placed at the center of the sphere, and the sensor detects the amount of light bouncing off the sample. (The region around the sample, and the inside of the sphere, are coated with a light-absorbing material like lampblack.) When the light source is shining on the sample at a grazing angle, the recorded reflection values are quite small (partly because so much of the illumination hits the black paint around the sample rather than the sample itself). One can imagine raising the intensity of the spotlight so that the total energy falling on the sample is constant, independent of the spotlight location; to do so, we'd have to multiply the basic intensity by $\frac{1}{\cos\phi}$, where ϕ is the colatitude of the spotlight. Because this would require some difficult engineering, we instead simply multiply the sensor reading by that value.

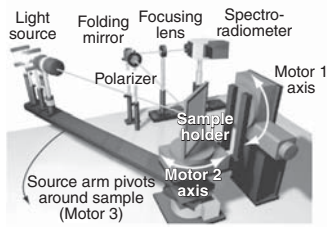


Figure 26.32: A modern gonioreflectometry system. (Courtesy of Steve Westin, "Automated three-axis gonioreflectometer for computer graphics applications" by Westin, Foo, Li, and Torrance, from Advanced Characterization Techniques for Optics, Semiconductors, and Nanotechnologies II, Proc. SPIE 5878, August 2005).

So, what does the gonioreflectometer measure? Assuming for the moment that the radius of the sphere is so large, and the spotlight, sensor, and sample are so small, that areas and solid angles can be treated as infinitesimal, it measures

$$f_r(P, \omega_i, \omega_o, \lambda) = \frac{L(t, P, \omega_o, \lambda)}{L(t, P, -\omega_i, \lambda)\cos(\phi)\, m(\Omega)}, \tag{26.78}$$

where

- Ω is the solid angle subtended by the light source at the sample, and we've written $m(\Omega)$ to indicate its measure

- ω_i and ω_o are respectively the directions from the sample *to* the source and to the sensor

You can purchase a radiance meter; the light meter used by photographers is a crude version. The computation done to get readings from the gonioreflectomer, is this: first measure the radiance, L_1, arriving from the light source. Measure the area, A, of the light source, and its distance, r, from the sample. The solid angle it subtends is then A/r^2. Now for each (ω, ω') pair, place the light source so that light arrives in direction $-\omega'$, and the radiance sensor so that it detects the radiance from the sample in direction ω. Call this radiance value L_2. Then the "reading" from the gonioreflectometer is $\frac{L_2}{L_1(\frac{A}{r^2})}\omega' \cdot \mathbf{n})$. In practice, it's best to measure L_2^0, the radiance from the sample when the illuminator is off, and L_2^1, the radiance with the illuminator on, and then compute $L_2 = L_2^1 - L_2^0$; this prevents the too-high readings for grazing angles that can arise when stray light enters the device, or when there's some offset in the calibration of the radiance meter, so that even total darkness registers as having some positive radiance.

We call f_r the (spectral) **bidirectional reflectance distribution function,** or **BRDF.** Note that f_r has units of sr^{-1}. Because the gonioreflectometer light source is constant (i.e., the radiance leaving it is constant), the value defining f_r is independent of time. One can extend the definition of f_r to directions ω_i and ω_o "on the wrong side" of the surface by defining it to be zero there. The domain of f_r is then

$$\mathcal{M} \times \mathbf{S^2} \times \mathbf{S^2} \times \mathbf{R}^+, \tag{26.79}$$

where $\mathcal{M}$ is the collection of all surfaces in the world. Note that in the definition of $f_r(P, \omega_i, \omega_o, \lambda)$, the vector ω_i is a unit vector pointing *toward* the incoming light, so the incoming light is traveling in direction $-\omega_i$.

With this definition of f_r, the correct form for relating the outgoing radiance L^r from a surface to the incoming radiance L^i is

$$L^r(t, P, \omega_o, \lambda) = \int_{\omega_i \in \mathbf{S}^2_+(P)} L^i(t, P, -\omega_i, \lambda) f_r(P, \omega_i, \omega_o, \lambda)\ \omega_i \cdot \mathbf{n}(P)\ d\omega_i \tag{26.80}$$

because the cosine of the colatitude of the incoming direction is just the negative dot product of that direction vector and the outward surface normal. This is the **reflectance equation,** the central part of a more general **rendering equation** [Kaj86, ICG86, NN85], to which we'll return in Chapter 29.

Because of the cosine factor, some books describe reflectance as the ratio of outgoing radiance to incoming irradiance. We've instead defined it directly in terms of L^i for clarity.

The BRDF generally has an important symmetry called **Helmholtz reciprocity:**

$$f_r(P, \omega_i, \omega_o, \lambda) = f_r(P, \omega_o, \omega_i, \lambda). \tag{26.81}$$

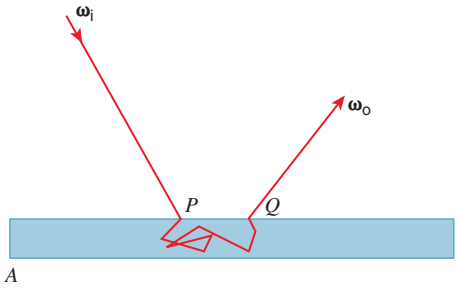


Figure 26.33: Subsurface scattering. To describe the scattering of light that arrives in direction $-\omega_i$ at a point P of a surface that's not a pure reflector, but rather allows multiple subsurface bounces before the light is emitted, requires a description of the emitted light at every nearby point Q in every outgoing direction ω_o.

This tells us, for example, that we need not place the light source and sensor at every possible pair of positions when we measure the BRDF with a gonioreflectometer; we can skip half of the positions.

But the Helmholtz reciprocity law also tells us that not just any function can be the BRDF of a material. In fact, there are further restrictions. For instance, if a certain amount of power arrives at a surface and is reflected, the amount of power leaving the surface must be no more than the amount that arrived, because of energy conservation. This places restrictions on various integrals of the BRDF.

Helmholtz reciprocity holds for a great many materials; indeed, there have been several purported proofs of it. Those proofs are thrown into doubt by the existence of materials that have been measured and shown to not satisfy the "law." Veach [Vea97] discusses the hypotheses necessary for reciprocity to hold.

26.10.1 Related Terms

A thin sheet of colored vinyl may both reflect light and transmit it; in such a case, one can build a gonioreflectometer to measure the transmitted light instead of the reflected light; the portion of the function f_r that we'd defined as zero for purely reflective surfaces becomes nonzero in this case, and the "reflectance" portion of the function is set to zero. The resultant function is called the **bidirectional transmittance distribution function,** or **BTDF.**

The sum of these two is called the **bidirectional scattering distribution function,** or **BSDF,** which we'll denote f_s.

When a ray of light meets a surface at a point P, we've been assuming that it is reflected (or transmitted) and again leaves from the point P. In many interesting materials, including human skin, hair, many forms of wood, and tree leaves, light actually enters the material, reflects multiple times under the surface, and is reemitted from some point near P (see Figure 26.33). This scattering is characterized by a **bidirectional surface scattering reflectance distribution function,** or **BSSRDF,** which has, as arguments, the point P, the direction of the arriving light ω_i, the point Q from which the light exits, and the direction ω_o in which it exits. Fortunately, for many surfaces the simpler BSDF suffices. Rendering materials using the BSSRDF, however, can produce some spectacular results [JMLH01] (see Figure 26.34).

Figure 26.34: Subsurface scattering lets light pass through the marble and skin in these images. (Courtesy of Stephen Marschner, ©2001 ACM, Inc. Reprinted by permission.)

26.10.2 Mirrors, Glass, Reciprocity, and the BRDF

Imagine trying to measure the BRDF for a perfect mirror using a gonioreflectometer. The sensor computes the incoming radiance by measuring the arriving power and dividing by the sensor area, which implicitly makes the assumption that the radiance along all rays from the sample to the sensor is approximately constant. But if the source is so tiny that its reflected rays all lie well within the sensor area, this assumption is invalid: For much of the sensor, no light is arriving at all. So a gonioreflectometer for measuring the BRDF of a mirror must have a sensor whose size is adjustable so that the whole sensor area is saturated with light. Of course, because mirror reflection is so "concentrated" (the rays of light don't spread out at all after reflection), it makes sense to try to measure the BRDF with a very tiny light source, perhaps a source with an adjustable iris. As we shut down the iris on the source, we'll have to make the sensor area smaller to compensate. The one thing we know is that the radiance of a ray from the source to the sample is the same as the radiance from the sample to the sensor, because the material is a perfect mirror. Now consider our definition for the BRDF:

$$f_{\mathrm{r}}(P, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda) = \frac{L(t, P, \omega_{\mathrm{o}}, \lambda)}{L(t, P, -\omega_{\mathrm{i}}, \lambda)\cos(\phi)\, m(\Omega)}. \tag{26.82}$$

Suppose we first perform the measurement with a source at $\phi = 0$ whose solid angle, measured from the sample, is 0.01 sr. The two radiances in the formula are the same, so they cancel, and the BRDF measurement is $100\,\mathrm{sr}^{-1}$.

Now suppose that we close down the iris on the light source so that it subtends a solid angle of 0.005 sr. We'll have to shrink the sensor as well to get a valid radiance measurement, but when we do so we'll again find that the received radiance is the same as the emitted radiance. The BRDF measurement will be $200\,\mathrm{sr}^{-1}$. As we continue closing the iris, the measurements will increase without bound. The conclusion is that for a perfect mirror, the BRDF is infinite. To be more precise: If the reflection of ω_{i} is ω_{o}, and the surface being measured is a mirror, then $f_{\mathrm{r}}(P, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda)$ is infinite.

It doesn't actually make sense to say that a function takes on the value "infinity," but there is a mathematical theory of a different class of objects, called distributions, which *can* take on infinite values. The name "bidirectional reflectance *distribution* function" reflects this. One difficulty with this notion of infinite values arises when we consider an *imperfect* mirror, one that reflects only half the

arriving light, and absorbs the rest. If we carry out the preceding analysis for such a mirror, we find that its BRDF *also* takes on an infinite value for mirror reflection, but we have the sense that it's a "different infinity" that's only half as big. We'll circumvent these problems by splitting any BRDF into two parts: a "diffuse" part and an "impulse" part, where the latter is a representation of things like mirror reflection and Snell's law refraction.

If we ignore, for a moment, the problem of the infinite values, we can still look at the formula for f_r and consider the cosine that appears there. What happens when we swap the roles of ω_i and ω_o for the perfect mirror? Since the incoming and outgoing radiances are equal, the only possible change is in the cosine. But for a mirror reflection, the incoming and outgoing angles are equal; this means that to the degree that f_r makes any sense at all, it seems to satisfy Helmholtz reciprocity.

On the other hand, when it comes to measuring f_s for a material like glass, the transmissive part of the computation involves two *different* angles, determined by Snell's law. It's evident that in this case, even if we can make sense of the infinite value of f_s, it will not satisfy the reciprocity law.

26.10.3 Writing L in Different Ways

The function L is defined on $\mathbf{R} \times \mathbf{R^3} \times \mathbf{S^2} \times \mathbf{R}^+$. Thus, it makes sense to write expressions like $L(t, P, \omega, \lambda)$, where $\omega \in \mathbf{S^2}$ and $P \in \mathbf{R^3}$; it makes equally good sense to write $L(t, x, y, z, \omega, \lambda)$, where x, y, and z are real numbers. In a computer program, of course, assuming that "point in 3-space" is a class of objects, we'd have to use an overloaded function, with arguments of type `real * point3 * spherepoint * real` or `real * real * real * real * spherepoint * real`, but the distinction between the two would be so tiny that it wouldn't matter.

But the `spherepoint` class is trickier. One *can* choose to characterize each point on the unit sphere by its (θ, ϕ) coordinates, but those two numbers are not the same thing as the point on the sphere. One could also represent the point by its (x, y, z) coordinates, although in a practical sense it's very hard to find computer-based real numbers satisfying $x^2 + y^2 + z^2 = 1$ exactly. In engineering and physics it's common to gloss over these difficulties, and, for a function U defined on $\mathbf{S^2}$, write things like

$$U(\theta, \phi) = U(x, y, z), \tag{26.83}$$

where

$$x = \cos\theta \sin\phi, \tag{26.84}$$

$$y = \cos\phi, \text{ and} \tag{26.85}$$

$$z = \sin\theta \sin\phi\text{".} \tag{26.86}$$

Unfortunately, this sort of overloading, although it can be used in computer programs sometimes, makes little sense in mathematics. The symbol U can mean only one thing.

For this reason, we'll carefully reserve the symbol L to denote the function whose domain is $\mathbf{R} \times \mathbf{R^3} \times \mathbf{S^2} \times \mathbf{R}^+$; when we need one of the closely related functions (e.g., defined in terms of polar angles in Chapter 27), we'll give it a new name to distinguish it from the original. Later when we discuss rendering

algorithms, we'll be making a steady-state assumption (that $L(t, \ldots)$ is independent of the parameter t), and the wavelength parameter λ will never enter into any formulas in any significant way, so we'll write $L(P, \omega)$ instead of $L(t, P, \omega, \lambda)$.

26.11 Discussion and Further Reading

We've described how light energy is measured, and given some details of the microscopic characteristics of light and their relationship to the characteristics of the matter with which light interacts, but proper treatment of these belongs in the realm of physics. A superb first reference is Crawford's book on waves [Cra68]. One should also understand some electromagnetic theory [Pur11]. Anyone who studies graphics will also benefit from even a partial reading of Newton's *Opticks* [New18]. It teaches not only the ideas of optics, but how a brilliant observer and experimenter works. For the application of these ideas in practical algorithms, see the book by Pharr and Humphreys [PH10].

We've discussed the *radiometric* terms used in measuring light, but there are also **photometric** terms, which attempt to capture the human perception of light. In particular, light of different wavelengths can be perceived as equally bright, and so by summing up the radiance at many different wavelengths, each multiplied by a factor $\bar{y}(\lambda)$ representing the perceived brightness of a certain amount of light energy at wavelength λ, you can get a single number (called **luminance**) that represents total brightness. The function $\bar{y}$ is called the **luminous efficiency;** it also is used in the CIE color system, which we discuss in Chapter 28. Luminance is measured in **lumens.** Such a single number for brightness of a light makes sense in contexts where most light is broad-spectrum, and most reflectors reflect light across much of the spectrum. But because light energy and reflectivity are spectral quantities, it's possible to have a light of high luminance (e.g., a bright red laser) and a surface with high reflectivity (where this reflectivity is an average over all wavelengths of the spectral reflectivity), and yet have the reflected light be low luminance, if the surface, for example, happens to absorb rather than reflect light at the wavelength of the laser. Because of this, such photometric terms see little use in graphics, but they are of considerable importance in illumination engineering.

26.12 Exercises

Exercise 26.1: We claimed in the chapter that the horizontal projection P to the unit sphere from the vertical cylinder enclosing it was an area-preserving map. Compute the derivative of this map at a point $(x, y, 0)$ of the cylinder, and verify that it *is* area-preserving. Why is it sufficient to carry out this computation at a point where $z = 0$?

Exercise 26.2: We computed the solid angle subtended by a spherical cap of radius $r < \pi$ on the unit sphere as $2\pi(1 - \cos(r))$.
(a) What is the solid angle subtended by a spherical cap of radius r on a sphere of radius R?
(b) What is the actual *area* of a spherical cap of radius r on a sphere of radius R? You should be able to do this problem almost by inspection, without any integrals at all.

◈ **Exercise 26.3:** We computed the radiance emitted from each point of a spherical uniform source of radius r and total power Φ in each outward direction ω

as $\frac{\Phi}{4\pi(\pi r^2)}$. Now consider some distant point P, lying on a surface that faces the center C of the spherical source, and that the distance from C to P is $R >> r$. We can compute the irradiance at P by computing the solid angle subtended by the spherical source, and the radiance arriving at P along each direction of that solid angle, etc.
(a) Do this computation to determine the irradiance at P.
(b) Now suppose that the total power H remains constant, while the radius r of the source shrinks. What is the limit of the expression you found in part (a) as $r \to 0$?

Exercise 26.4: It would be nice to imagine a "beam" of light arriving at a point P of a surface in direction ω_i, and being reflected out in many directions; we could then look at how much light goes in each direction and talk about the "scattering" from the surface. The problem is that light arriving at a single point cannot carry any energy, and light arriving in a single direction cannot carry any energy. Instead, we can imagine light arriving in directions η with $|\eta - \omega_i| \leq \epsilon$ (i.e., directions very nearly parallel to ω_i), and arriving at points Q with $\|Q - P\| < r$, for some small r, with constant radiance ℓ. In this problem, we'll examine the irradiance due to this "beam." If the BRDF is continuous as a function of position and incoming direction, then the outgoing radiance in direction ω_o will vary smoothly as a function of r and ϵ.
(a) What's the solid angle of the incoming rays as a function of ϵ? What's a simplified expression for small ϵ?
(b) How should we adjust the radiance ℓ along incoming rays, as we reduce r and ϵ toward zero, to make certain that the incoming power is constant? When we adjust the incoming radiance in this way, and take a limit, we can speak of a beam of light in direction ω_i having a certain irradiance; the resultant radiance in direction ω_o can then be measured (theoretically). The ratio

$$\frac{L_o}{\ell(r,\epsilon)|\mathbf{n}\cdot\omega_i|\pi\epsilon^2\pi r^2}$$

has a limit as ϵ and r go to zero because of the form of ℓ, and the limit is exactly the BRDF; this is the justification for defining the BRDF as "the ratio of the outgoing radiance in direction ω_o to the incoming radiance of a beam in direction ω_i." Therefore, the integral of the BRDF, over all outgoing directions, multiplied by $\|\omega_o \cdot \mathbf{n}\|$, tells how much of the power arriving in the beam gets reflected in any direction at all, and is therefore called the **directional hemispherical reflectance.**

Exercise 26.5: Latex wall paint is designed to be **Lambertian,** that is, it's designed so that when illuminated by light from any direction, it has the same apparent brightness regardless of the direction from which it's viewed (i.e., the radiance along every outgoing ray is the same). Furthermore, the outgoing radiance should be independent of the incoming direction of the illumination, so long as the power arriving at a fixed region of the painted surface is constant. Good latex paint very nearly achieves this goal, although at grazing angles the reflectance varies from the ideal. If it *were* such an ideal reflector, what would its BRDF look like?

Exercise 26.6: A planar surface S sits in a room that's bathed in light so that the radiance along every ray arriving at S is the same constant, 10 watts per steradian per square meter. What's the irradiance at a point P of the surface?

Exercise 26.7: Two incandescent bulbs emit the same total power in the visible spectrum; one has a filament at 4000 K, the other at 6500 K. Because of the

Stefan-Boltzmann law, the second filament must be much smaller than the first. Which one emits more power in the *invisible* portion of the spectrum?

Exercise 26.8: A student is given data captured from a digital camera; the scene viewed by the camera is a spherical frosted incandescent lamp that emits 1.2 W of power in the visible range. The lamp has a radius of 0.0175 m. The camera shutter speed has been adjusted so that the sensor is neither saturated nor starved; indeed, the values in the image are near the center of the camera's pixel-value range, and can be assumed to be proportional to radiance. The value observed for pixels that lie in the lamp is 4000 (on a scale from 0 to 8191). The student wants to know the constant of proportionality between the radiance of arriving light and the sensor value. The student says, "The lamp is supposed to emit uniformly because of its frosting, so except for really tangential angles, we can figure that the radiance of all outgoing rays is constant. The area of the lamp is $4\pi r^2 \approx 0.000\,962\,\mathrm{m}^2$; the hemisphere over which the light is radiated has solid angle measure approximately 6.28 sr, so the radiance, by division, is the power divided by the angle and the area, giving $L \approx 1.2\,\mathrm{W}/(0.000\,962\,\mathrm{m}^2 * 6.28\,\mathrm{sr}) \approx 0.006\,\mathrm{W}/\mathrm{m}^2\mathrm{sr}$. So to get the radiance L from the sensor value, we multiply by $\frac{0.006}{4000} = 1.5 \times 10^{-6}$." Critique the student's approach, and give the correct answer.

Exercise 26.9: We said that "A 'disk' consisting of all points on the unit sphere whose spherical distance from a point P is less than r (where $r < \pi$) has solid angle measure $2\pi(1 - \cos(r))$." But you'd expect, for small r, that the solid angle would contain an r^2 factor, because the formula for the area of a disk in the plane contains an r^2 factor. Reconcile the two by recalling the Taylor series for $\cos(r)$ at $r = 0$.

Exercise 26.10: Consider a disk of radius s in the plane, centered at the origin, and the point $P = (0, 0, -h)$ that's distance h below the disk. Assume that the disk is a Lambertian emitter, emitting radiance L in every direction from every point, and is the only surface in the scene.

(a) Write out an integral for the irradiance at P.

(b) Evaluate the integral. Switching to polar coordinates on the plane will help.

(c) Show that if $h < s/10$, then the irradiance at P is essentially the same as it would be if the disk covered the entire plane (i.e., if s were very large). Thus, for small values of h, the irradiance is nearly constant.

(d) Show that for $h > 4s$, the irradiance is within 1% of $\pi L(s/h)^2$.

Exercise 26.11: Show that any plane wave E traveling along the x-axis as in Equation 26.6 can be expressed as the sum of two plane waves, $E_|$ and $E_\circ$, the first being linearly polarized and the second being circularly polarized. Hints: The axis of the linearly polarized wave will be $\begin{bmatrix}0 & A_y & A_z\end{bmatrix}^\mathbf{T}$; the magnitude of the circularly polarized wave will be $\sqrt{A_y^2 + A_z^2}$.

Exercise 26.12: Suppose that in Figure 26.11, we draw a circle of radius r about the refraction point. On the line tangent to the top of this circle, we place equispaced points, and from each point draw a ray toward the refraction point. These rays refract with different angles into the lower material. Each refracted ray meets a horizontal line tangent to the *bottom* of the circle.

(a) Draw a figure depicting this situation.

(b) Show that the points of intersection on the bottom line are also equispaced.

(c) What is the ratio of the bottom-line spacing to the top-line spacing?

Exercise 26.13: Use Planck's formula for blackbody radiation $R(f, T)$ in terms of frequency to approximate the location of the power peak in $R(f, T)$ as a

function of T (i.e., for fixed T, find the f that maximizes power). Show that this is roughly linear in T. (This is known as **Wien's displacement law.**) To do so, fix T, and define a variable $u = f/T$, and write $R(f, T)$ as a function $S(u) = \frac{T^3 u^3}{D(u)}$, where $D(u) = e^{(h/k)u} - 1$, which depends only on the single variable u. Under what condition on u is $S(u)$ a maximum? (You may assume that $e^{(h/k)u} >> 1$.)

For T near 10,000°K (roughly the temperature of the surface of the sun), the peak is at about frequency 5×10^{14} Hz, which is just about in the middle of the range of visible light (roughly 4×10^{14} Hz to 7×10^{14} Hz); this means that the human visual system is most sensitive to the energy that's most common (sunlight).

Chapter 27

Materials and Scattering

27.1 Introduction

This chapter is about the way we model the interaction of light with objects. The first several sections are about the physics and mathematical representation of these interactions. At the end, we briefly discuss a software interface that's well suited to the rendering we'll do later.

The first step in our discussion is to limit consideration, for much of the chapter, to light-object interactions that occur at surfaces; late in the chapter we briefly discuss the volumetric interaction that occurs in things like fog and colored water, etc. All these interactions, when considered *locally* (e.g., "Where does the light arriving at this bit of surface end up going?"), are called **scattering.** Mirror reflection, for instance, is one very special kind of scattering; Lambertian reflection is another.

Scattering is a messy process. For many materials of interest, the physical features of the material are at a scale of just a few wavelengths of light, so diffraction effects matter. The interaction of light with materials is dependent on the chemistry of the materials—the degree to which the material is a conductor or insulator, and the distribution of electron-energy levels in the material, as we saw in Chapter 26—which is highly variable. And even for the simplest of rough materials, light interacts with the rough material in varied and complex ways. The result is that scattering code is messy. If you peek inside almost any renderer, the representation of scattering is either oversimplified or very messy.

27.2 Object-Level Scattering

Operational definitions, like Le Corbusier's "A house is a machine for living," have an intrinsic appeal: They get to the heart of a subject from the speaker's point of view. From the point of view of a renderer, an **object** is a machine for converting an incoming light field into an outgoing one, through some kind of interaction that's of no particular relevance to the renderer. The machine has a few useful properties, determined by the laws of physics: If we sum two incoming

light fields, the outgoing light fields also sum. This linearity places very serious restrictions on the kinds of transformations that can take place. It also means that we can study the "response" of the object to incoming radiance along single rays, and then integrate over a field of such rays to get the outgoing radiance field for an arbitrary incoming field.

While intrinsically appealing, such a representation isn't really practical in general: Writing down the response to all possible incoming light fields (even single-ray responses!) requires too much memory. But it's worth holding on to the idea that any representation we make must somehow encompass the "transformer-of-light-fields" ideal just presented.

Some objects, such as fog, have no explicit geometry. But in the case where an object *does* have some geometry, it's useful to "factor" the way it interacts with light into *geometry* and **material** where by "material" we mean to suggest the characteristics of the object that are independent of position. Thus, "aluminum" is a material, and an aluminum sphere scatters light in the same way from its north pole as from its south pole. This splitting into geometry and material represents an enormous simplification and compression: We need to know how one tiny bit of material scatters light, and then we reuse this knowledge at other points. Of course, for this to work well, the object must be made of a homogeneous material. If the material varies from point to point (e.g., as with a sedimentary rock), a compromise solution is often workable: We describe a parameterized *class* of materials, and associate (through texture mapping) some parameters to each point of the surface so that at one surface point the material is red sandstone and at another it's ochre sandstone, for instance.

Such factoring can be taken further. We sometimes factor the bidirectional scattering distribution function (BSDF) into two parts: a "surface color" at each point, and an underlying BSDF. To compute scattered light, we use the underlying BSDF to compute how much light is scattered, and then compute the spectral distribution of the outgoing light as a product of the spectral distribution of the incoming light times this basic reflectance times the "surface color," which is really a per-wavelength reflectivity, typically represented by just three values (usually called "red," "green," and "blue"). You already saw an example of a BRDF-like reflection model in Chapter 6—where we described a "lighting model" for surfaces that involved diffuse and specular RGB colors, and how they got multiplied by incoming light to compute the color with which a surface should be rendered—and in a more physically correct form in Chapter 14.

27.3 Surface Scattering

As we mentioned in the previous chapters, the single-point description of scattering is typically represented by a BRDF. For materials where the light-object interaction involves transmission, or takes place throughout the material rather than at its surface (e.g., many cheeses), richer descriptions like the bidirectional scattering distribution function (BSDF) or bidirectional surface scattering reflectance distribution function (BSSRDF) are needed. For volumetric materials, like fog, even more complex descriptions are needed. We'll concentrate on the surface-material examples, but we will touch on the others as well. The questions to consider, as we do so, are the following.

- What BRDF (or BSDF, or BSSRDF, etc.) is the one to use to model some material?

- What mathematical or computational representation should we use for this model?

Henceforth, we'll generally refer to the model of scattering as the BSDF, except (a) when we're talking about reflection-only scattering, where terms like "the Blinn-Phong BRDF" are common, and (b) briefly in our discussion of reciprocity, and when we talk about subsurface scattering. In most equations, wherever it makes sense we'll use f_s rather than f_r.

27.3.1 Impulses

One of the most challenging problems in the numerical representation of the scattering of light by surfaces is the difference between **specular** reflections—the mirrorlike reflections that we see in extremely shiny surfaces—and **diffuse** scattering, in which an incoming beam of light spreads out into light going in almost every direction, as happens when it meets a surface made of flat latex paint. Most of a light beam hitting a mirror scatters in one primary direction, but some small amount scatters in other directions as well. By just about any measure of light quantity, the scattering in the primary direction is a *huge* multiple of the scattering in other directions. (A multiplier of 10^{10} is well within reason.) And the falloff in light quantity, as one moves away from the primary direction, is exceptionally rapid as well. It therefore makes some sense to separate out the specular reflection and treat it as a *pointwise* phenomenon, and regard the rest of the scattering as a smooth function of outgoing direction. The same applies to the kind of intermaterial transmission described by Snell's law: Incoming light in one direction essentially exits in some other direction. We'll call both of these **impulses** in the scattering, and treat them separately from the diffuse effects.

27.3.2 Types of Scattering Models

We'll discuss the following types of scattering models.

- **Empirical/phenomenological models:** These are models crafted to simulate some observed scattering phenomenon. The Phong model of Section 6.5.3 is an example. With little physical motivation, it's designed to allow a user to vary between nearly Lambertian and highly glossy appearances for a surface.
- **Measured/captured models:** These are models in which the BSDF is carefully measured and stored; when we need the BSDF for a particular pair of directions (ω_i, ω_o), we look for them (or directions near them) in a large table of stored data, perhaps interpolating from nearby samples when necessary.
- **Physically based models:** These are models based on some degree of understanding of the physical interaction of light with materials. They occupy the bulk of this chapter.

27.3.3 Physical Constraints on Scattering

We cannot, for a passive material, have more light scattered from the surface than arrives there. (An "active" object, like a photosensor that triggers a strobe light whenever it detects light, can obviously emit more light than it receives.)

The condition that no more energy leave the surface than arrives there is called **energy conservation** (with the assumption that unscattered energy is "conserved" by appearing as heat). Not every scattering model is energy-conservative. Phong's original model had no physical units attached, so it was impossible to tell whether it was conservative or not! In general, conservation can be expressed as a constraint on the integral of the BSDF; we'll see this worked out in detail for Lambertian scattering.

The other commonly used constraint on scattering is **reciprocity:** If $(\omega_i, \omega_o) \mapsto f_r(\omega_i, \omega_o)$ is the BRDF for some material, then $f_r(\omega_i, \omega_o) = f_r(\omega_o, \omega_i)$. Veach [Vea97] generalizes this to include transmission: For light arriving from direction ω_i in a medium of refractive index n_i, and scattering in direction ω_o in a medium of index n_o,

$$\frac{f_s(\omega_i, \omega_o)}{n_o^2} = \frac{f_s(\omega_o, \omega_i)}{n_i^2}. \tag{27.1}$$

Note that when this is applied to reflection, the two refractive indices are identical, and the equation simplifies to the usual symmetry law.

It's well known in graphics that the BRDF is symmetric, that is, $f_s(\omega_i, \omega_o) = f_s(\omega_o, \omega_i)$, and this equality is usually attributed to Helmholtz. Veach describes a proof of symmetry, and explains why Helmholtz's remarks, which involve mirror reflectors and lenses, are insufficient to imply reciprocity and why several other purported proofs have flaws in them. Despite this, the reciprocity property is still widely known as "Helmholtz reciprocity."

Despite Veach's proof of reciprocity, there are materials such as pearlescent paints for which reciprocity apparently does not hold [CPMV+09]. Such materials do not contradict the proof, which assumes that the materials involved in the scattering are homogeneous.

So is the BRDF symmetric or not? For a very wide range of measured materials, the answer appears to be "It is, for almost all practical purposes." Snyder [SWL98] explains this in some detail.

27.4 Kinds of Scattering

Looking at the differences among solid materials, one of the first things that attracts our attention is the range of shininess, from the matte appearance of chalk to the very shiny appearance of a polished metal surface. Another is that some surfaces are transparent while others are reflective. As we already saw in Chapter 26, these differences are in part due to fundamental physical processes and structures: Conductive materials, with lots of free electrons, tend to be reflective; those whose electron orbital energy levels lack "gaps" corresponding to the energies of quanta of visible light tend to be transparent, etc. But at a higher level, it's useful to have a language for describing kinds of scattering: reflective, transmissive, mirror, impulse, glossy, diffuse, Lambertian, retroreflective, refractive. To give these meaning, we'll consider a flat surface (see Figure 27.1) with outward normal vector $\mathbf{n}$ (i.e., $\mathbf{n}$ points from the material toward empty space), and the hemispheres $\mathbf{S}^2_+(\mathbf{n})$ and $\mathbf{S}^2_-(\mathbf{n})$, where $\mathbf{S}^2_+(\mathbf{n})$ is the set of all unit vectors ω with $\omega \cdot \mathbf{n} \geq 0$, and $\mathbf{S}^2_-(\mathbf{n})$ is the set of all those for which the dot product is nonpositive. Since we'll mostly be considering a single surface with a fixed normal

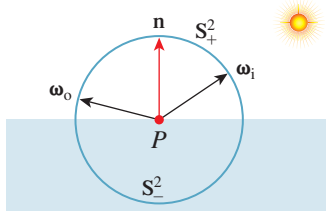


Figure 27.1: We'll consider a surface (shown shaded) with normal vector $\mathbf{n}$*;* $\mathbf{S}^2_+$ *consists of all vectors pointing* away *from the surface, while* $\mathbf{S}^2_-$ *consists of vectors pointing* into *the surface. Light arrives from a source that's in direction* $\omega_i \in \mathbf{S}^2_+$*, and scatters in directions* ω_o*, which may be in either* $\mathbf{S}^2_-$ *or* $\mathbf{S}^2_+$*.*

$\mathbf{n} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^{\mathbf{T}}$, we'll simply write $\mathbf{S}^2_+$ or $\mathbf{S}^2_-$. Indeed, our diagrams will always show a flat surface facing up, and we'll sometimes refer to the "upper" and "lower" hemispheres. We'll consider light arriving from some direction $\omega_i \in \mathbf{S}^2_+$ (i.e., traveling in direction $-\omega_i$), and consider the scattered light from the surface, using ω_o to denote a generic direction of scattered light. We'll use $f_s(\omega_i, \omega_o)$ to denote the BSDF for light arriving in direction ω_i and leaving in direction ω_o. (In the case of transmission, we'll denote the transmitted light direction ω_t.)

While the formulation given here—an object that lies in the $y \leq 0$ half-space, scattering light upward—is nice and simple, it's an approximation of a larger truth, which we'll return to in Section 29.4, namely, that we have a surface that's a boundary between *two* materials. For a glass marble, there's an outward-pointing normal vector that points into the surrounding air, but for the mass of surrounding air, the outward normal points into the glass! We tend to think of air as "nothing" in graphics, but in a situation like a glass filled with wine, the boundary between the glass and the wine serves to delimit both the glass and the wine. Surface scattering is really a property of a *pair* of media rather than a single medium.

Figure 27.2: A teapot with generic reflective scattering (image sequence by Kefei Lei).

Before we describe the kinds of scattering, we caution that some terms are used very informally. For example, "diffuse" can be used to mean "anything except mirror reflection" or "very similar to Lambertian."

Here is a collection of terms used for scattering, with figures showing some of them.

- **Reflective** (Figure 27.2): The scattered light is all in the upper hemisphere, that is, $\omega_o \cdot \mathbf{n} \geq 0$. More precisely, $f_s(\omega_i, \omega_o) = 0$ for $\omega_o \notin \mathbf{S}^2_+$.
- **Transmissive** (Figure 27.3): The scattered light is all in the lower hemisphere, that is, $\omega_o \cdot \mathbf{n} \leq 0$. More precisely, $f_s(\omega_i, \omega_o) = 0$ for $\omega_o \notin \mathbf{S}^2_-$.
- **Mirror** (Figure 27.4; "specular" is a synonym): The scattered light is all in a single direction, the mirror-reflection direction $\omega_r = 2(\omega_i \cdot \mathbf{n})\mathbf{n} - \omega_i$. The "function" f_s has an infinity: $f_s(\omega_i, \omega_o) = \infty$. More precisely, trying to measure the reflectance in the usual way fails, because the outgoing radiance is independent of the solid angle subtended by the light source; this means that our usual BSDF approach is inappropriate for handling mirror-reflected light.[1] Practically speaking, this means that in the programs you write, you need to handle mirror reflection as a special case.
- **Impulse:** The scattered light is all in a single direction, but this direction is not necessarily the direction of mirror reflection. For example, we may want to model a camera lens as purely transmissive, with all

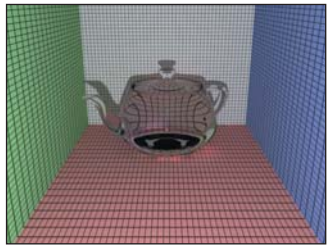


Figure 27.3: The teapot with primarily transmissive scattering.

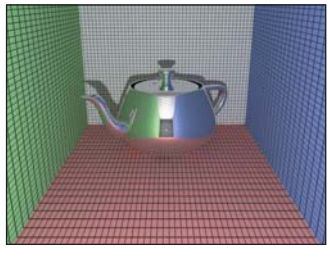


Figure 27.4: The teapot with mirror scattering.

1. ◇ In truth, this is a case where the notion of "distribution" applies: What we tend to write as integrals involving the BSDF typically have the form $\int f_s L g$, where L is some representation of radiance and g represents other terms like change of variables Jacobians, etc.; such integrals transform the radiance field L into either a number or another function, and they do so linearly. Thus, "integrating against the BSDF" is just a way to write a linear function from one function space to another. A few such linear maps *cannot* be written this way, just like the "delta functions" of Chapter 18, but in the quest for consistent notation, researchers in graphics pretend that they can and claim the BSDF "is infinite" at certain points of its domain.

light transmitted according to Snell's law. Once again, such impulse cases require special handling in our programs.

- **Glossy** (Figure 27.5): The scattered light is concentrated around some particular direction $\omega_s \in \mathbf{S}^2_+$, which is typically at or near the mirror-reflection direction. As noted earlier, the word "specular" is sometimes used to mean "glossy," typically when the concentration of scattered light is very tight. The reflection from an enameled coffee mug has a substantial glossy component: You can see things reflected in the mug's surface, although they appear *slightly* blurred. A waxed linoleum floor has a somewhat glossy appearance: You may be able to see objects reflected in it, but usually you can only see the gross outlines, and all detail in the reflection has been blurred away. The concentration of the scattered light is much less than that scattered from the enameled mug. The Phong model of Chapter 6 is a glossy-scattering model.
- **Diffuse:** The scattered light is spread out in all possible directions, that is, $f(\omega_i, \omega_o)$ is nonzero for all $\omega_o \in \mathbf{S}^2_+$, or, more weakly, $f(\omega_i, \omega_o) > 0$ for a large fraction of all directions. Many of the materials we encounter in everyday life exhibit diffuse scattering: paper, wood, brick, most cloth, etc.
- **Lambertian** (Figure 27.6): This is a special case of diffuse reflection in which the outgoing radiance in direction ω_o is independent of ω_o. No matter what position you view it from, a diffuse surface looks equally bright. This means that the BRDF, as a function of its second argument, is constant: $f_r(\omega_i, \omega_o) = f_r(\omega_i, \omega'_o)$ for any two vectors $\omega_o, \omega'_o \in \mathbf{S}^2_+$.
- **Retroreflective:** A surface is retroreflective if $f_s(\omega_i, \omega_o)$ is relatively large for some vector ω_o that's close to ω_i. Often the surface is specular, and the specular peak direction is very near ω_i. Retroreflective paint is used on road signs to make them more visible to drivers (whose headlights illuminate the signs), and retroreflective fabrics are often sewn into clothing for athletes to make them particularly visible to cars at night.
- **Refractive:** This is a special case of transmission analogous to mirror reflection: The transmitted light all lies in the direction $\omega_t \in \mathbf{S}^2_-$ determined by Snell's law.

Figure 27.5: Glossy scattering.

Figure 27.6: Lambertian scattering, the most diffuse possible.

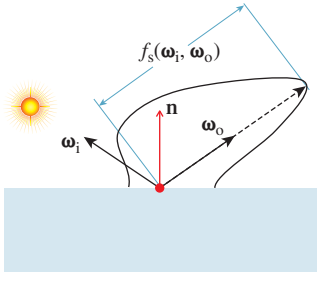


Figure 27.7: Plotting the BSDF. The arriving light comes from the left along direction $-\omega_i$; a typical outgoing direction ω_o is shown. The polar plot intersects the radial direction of ω_o at a distance $f_s(\omega_i, \omega_o)$.

For each of the classes of scattering described above, we can translate the description into a characterization of the BSDF, with some of the characterizations more precise than others. Because the BSDF is a function of two arguments, the direction ω_i of the arriving light and the direction ω_o of the scattered light, it's difficult to give a complete depiction. Instead, we pick a representative direction ω_i, and we plot the BSDF as a function of ω_o, that is, we draw a representation of the function $\omega_o \mapsto f_s(\omega_i, \omega_o)$. We further simplify by limiting ourselves to the case where $\omega_i, \mathbf{n}$, and ω_o are all in the same plane; with this restriction we can draw a radial graph in the plane as shown in Figure 27.7.

The dependence of the BSDF-curve shape on ω_i tends to be relatively simple in many cases, so this single plot can give a good sense of the overall function.

One important thing to understand is that the BSDF curve is *not* the pattern of emitted radiation. You might imagine that if you sent a stream of photons toward the material traveling in direction $-\omega_i$, the resultant outgoing photons (e.g., in the $\omega_i - \mathbf{n}$ plane) would have an angular distribution given by the BSDF curve in that plane (i.e., where the BSDF curve is twice as large, the probability density of a photon emerging in that direction is twice as large). That's not correct, however, as

we can see by considering a perfect Lambertian reflector, for which $f_s(\omega_i, \omega_o) = 1/\pi$ for all $\omega_i, \omega_o \in \mathbf{S}^2_+$. (We'll soon see why $1/\pi$ is the right constant.)

Suppose for a moment that for a photon arriving from a source in direction ω_i, the probability density of scattering in direction ω_o were the same in all directions ω_o. To estimate the radiance at a point Q_1 on the unit hemisphere around some surface point P (see Figure 27.8), we'd take a solid angle Ω_1 at Q_1 and measure the density of the light energy arriving in that solid angle. For the radiance to be constant, as we know it is for a Lambertian surface, we'd have to get the same value when we estimated it at Q_2 with a solid angle Ω_2 of the same measure. But the amount of reflecting surface subtended by the two solid angles varies like the inverse of the "outgoing cosine," leading to an extra $1/(\omega_o \cdot \mathbf{n})$ factor in the outgoing radiance. The probability density of an incoming photon being scattered in direction ω_o from a Lambertian surface must therefore be proportional to $f_s(\omega_i, \omega_o)(\omega_o \cdot \mathbf{n})$.

Figure 27.9 shows several overlapping classes of scattering that we've discussed, with the BSDF drawn in black and the scattering probability density drawn in blue.

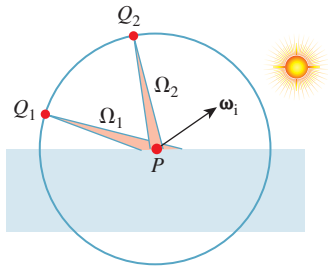


Figure 27.8: Computing the radiance at points near a hypothetical surface from which photons scatter equally in all directions.

27.5 Empirical and Phenomenological Models for Scattering

We now introduce a few basic scattering models that will serve several functions. The mirror and Lambertian models are the basis for several microfacet-based models that we'll discuss when we examine physically based models. And the Blinn-Phong model, although not strictly physically based, is very widely used in practice.

27.5.1 Mirror "Scattering"

An ideal mirror-reflecting plane (for which all light is reflected rather than absorbed), shown in Figure 27.10, reflects light from a source in such a way that the emitted light distribution is exactly the same as would be produced by an identical source at some position behind the mirror's location (assuming the mirror was removed). Because the outgoing radiance along each ray in these two situations is the same, the mirror-reflection process evidently results in no change in the net light energy in the scene.

Perfect mirrors are rare. More commonplace is that a mirrored surface in fact absorbs some amount of light, and reflects the remainder. The outgoing radiance along the mirror direction $\omega_r = \omega_i - 2(\mathbf{n} \cdot \omega_i)\mathbf{n}$ is therefore a constant multiple of the incoming radiance:

$$L(P, \omega_o) = \rho L(P, -\omega_i), \tag{27.2}$$

where the *reflectivity* ρ is a number between 0 and 1. All that's needed to specify the mirror reflectance from a surface is the normal vector $\mathbf{n}$ and the reflectance constant ρ, which is unitless. This constant may, however, have some spectral dependence (i.e., light at different wavelengths may be reflected differently).

The spectral dependence of reflectance depends on the underlying material. In broad terms, for insulators like plastic, the mirror-reflected light has the same

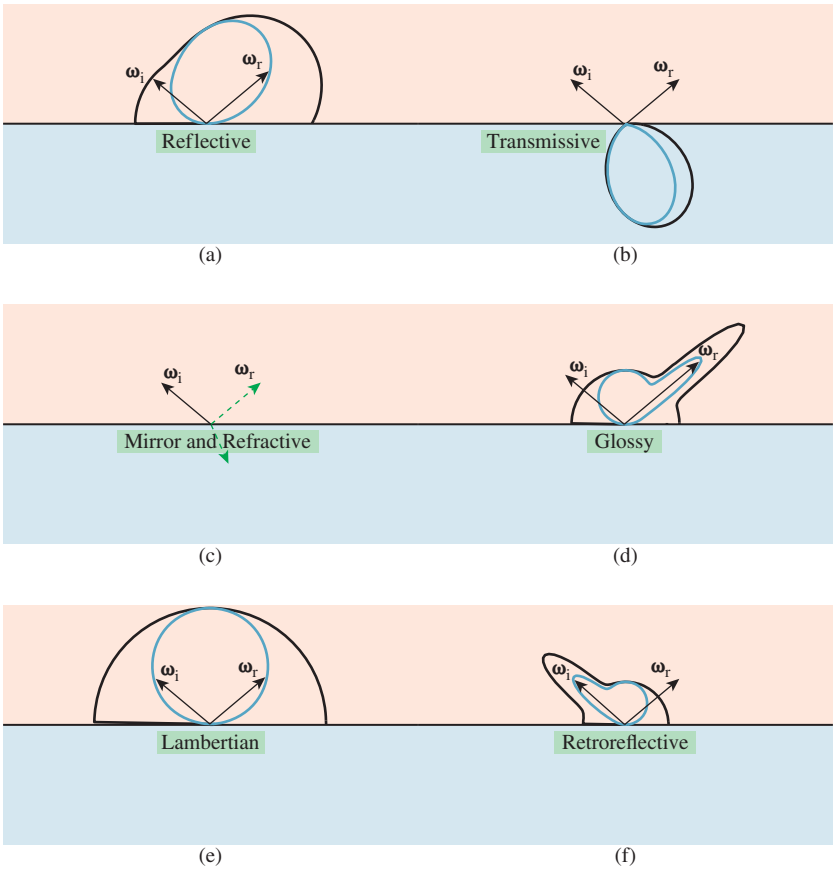


Figure 27.9: BSDF (black outer curves) and probability density plots (blue inner curves) for several classes of reflections; for impulse scattering, like mirror reflection, we've indicated the impulse direction with a green arrow as in the two right-pointing arrows in (c). For the others, the peaks in probability density and BSDF are offset from one another because of the cosine weighting. (a) A generic reflective material. (b) A purely transmissive material, which is physically unrealistic. (c) A material with mirror and refractive impulses; this is the kind of scattering we expect at an air-to-water interface. (d) Glossy scattering. (e) Lambertian scattering. (f) Retroreflective scattering.

spectral distribution as the incoming light, while for conductors, certain frequencies of light are preferentially reflected. This is why a polished piece of plastic has white highlights, while a polished piece of gold has gold-colored highlights. We'll return to this in Section 27.8.3.

The simplest summary representation of the spectral dependence of the reflected light is to just give RGB values, representing the overall reflectance of the material in response to long-, middle-, and short-wave incoming radiation. Thus, the summary computational model becomes

$$L(P, \omega_o, \lambda) = \begin{cases} \rho(\lambda) L(P, -\omega_i, \lambda) & \text{if } \omega_o = \omega_i - 2(\omega_i \cdot \mathbf{n})\mathbf{n} \text{ and } \omega_i \cdot \mathbf{n} > 0, \\ 0 & \text{otherwise,} \end{cases} \tag{27.3}$$

where λ represents the wavelength of the light as usual.

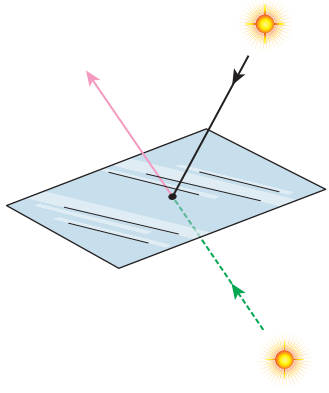


Figure 27.10: A mirrored plane, reflecting a light source above the plane (black solid line), produces the same outgoing radiance field (pink) as a "virtual" source below the plane placed at the proper location behind the mirror would produce (dashed green) in the absence of the mirror.

This simple model of mirror reflection ignores the Fresnel equations that arise from the polarization of light. There is no physical surface that actually reflects like a perfect mirror (or one with constant reflectivity $0 < \rho < 1$) for all angles of incoming and outgoing light. We'll return to a more sophisticated version of mirror scattering when we discuss physically based models.

27.5.2 Lambertian Reflectors

A Lambertian surface has the property that when it's illuminated, the outgoing radiance in every (reflected) direction is the same (there's no transmission). Furthermore, this outgoing radiance varies linearly with irradiance: Whether we reduce the incoming illumination or make it arrive at a grazing angle, the outgoing radiance varies in the same way. Thus, the BRDF is constant; we usually write $L(P, \omega_o) = \rho/\pi$, where ρ is a constant indicating what fraction of the arriving light energy is scattered.

Let's now check for which values of ρ this reflector will be energy-conserving. Imagine (as shown in Figure 27.11) that light arrives at a small, rectangular region, R, of material with area A, from a source like the sun: All incoming rays are in some small, solid angle Ω, and the radiance along each ray in a direction from Ω is the same constant ℓ. Then the total rate of energy arrival at the surface region R is the integral over R and Ω of the arriving radiance, multiplied by the dot product of the incoming direction and surface normal:

$$\begin{aligned}
\text{Power} &= \text{Energy arrival rate} && (27.4)\\
&= \int_{P\in R}\int_{\omega_i\in\Omega} L(P,-\omega_i)(\omega_i\cdot\mathbf{n})\,d\omega_i\,dP && (27.5)\\
&= \int_{P\in R}\int_{\omega_i\in\Omega} \ell(\omega_i\cdot\mathbf{n})\,d\omega_i\,dP && (27.6)\\
&= \ell A\int_{\omega_i\in\Omega} (\omega_i\cdot\mathbf{n})\,d\omega_i && (27.7)\\
&\approx \ell A\int_{\omega_i\in\Omega} \cos(\theta)\,d\omega_i && (27.8)\\
&= \ell A m(\Omega)\cos(\theta), && (27.9)
\end{aligned}$$

where $m(\Omega)$ denotes the measure of the solid angle Ω, and θ is the angle between a typical vector $\omega \in \Omega$ and the (constant) surface normal $\mathbf{n}$. As Ω gets small, the approximation of the dot product by a single central dot product gets increasingly accurate.

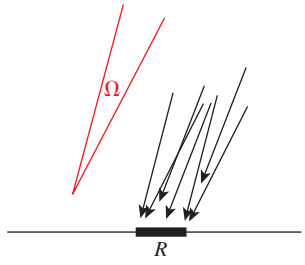


Figure 27.11: Light arrives at a small, rectangular sample R from directions $\omega \in \Omega$, a small solid angle; the radiance of the incoming light is independent of position and angle.

Inline Exercise 27.1: Use the reflectance equation to show that the radiance of a ray emitted from the region R is well approximated by $\ell(\rho/\pi)\cos(\theta)m(\Omega)$.

To compute the rate at which light energy is emitted, let us surround the sample by a very large, black, completely absorptive hemisphere, and determine the rate of energy arrival at that sphere. Just as in Chapter 26, we'll make the sphere so large that the ray from the point Q to any point on the emitter R always has essentially the same direction, independent of the emitter point.

The density $d(Q)$ of light energy (per second) arriving at a point Q of the hemisphere is the integral, over all directions, of the radiance arriving at Q. Since

the only source of incoming light is the region R, this simplifies to an integral over directions ω that point from Q to some location in R; we'll denote this set of directions Ω_Q. The density is then

$$d(Q) = \int_{\omega \in \Omega_Q} L(Q, -\omega)(\omega \cdot \mathbf{n}(Q))\, d\omega. \tag{27.10}$$

We know the radiance arriving at Q from the inline exercise above. Furthermore, since the disk R appears very small as viewed from Q, the vectors ω all point in approximately the same direction (namely, $\mathcal{S}(P - Q)$, where P is the center of the region R, and hence the center of the sphere as well). This means that we can approximate $d(Q)$ well by

$$d(Q) = m(\Omega_Q)\ell(\rho/\pi)\cos(\theta)m(\Omega)(\omega \cdot \mathbf{n}(Q)). \tag{27.11}$$

The last dot product is approximately 1, because ω points almost exactly toward the center of the large sphere on which Q lies. We can rewrite the measure of Ω_Q as $(A/r^2)\cos(\theta')$, where θ' is the angle between $\mathbf{n}$ and $Q - P$, that is, the outgoing angle, to get

$$d(Q) = \frac{A}{r^2}\ell(\rho/\pi)\cos(\theta)m(\Omega)\cos(\theta'). \tag{27.12}$$

Everything in this expression is constant (as a function of Q) except the final cosine. When we integrate this power density over the entire hemisphere, to get the total power arriving at the hemisphere, the result is

$$\text{Arriving power} = \frac{A}{r^2}\ell(\rho/\pi)\cos(\theta)m(\Omega)\int_{\mathbf{S}^2_+(r)} \cos(\theta') \tag{27.13}$$

$$= \frac{A}{r^2}\ell(\rho/\pi)\cos(\theta)m(\Omega)\pi r^2 \tag{27.14}$$

$$= A\ell\rho\cos(\theta)m(\Omega), \tag{27.15}$$

which is exactly ρ times the power that arrived at our Lambertian surface. So the scattering conserves power only if $\rho \leq 1$.

The computational model for a Lambertian surface consists of a normal vector and a per-wavelength (or per-primary) reflectance value.

The number ρ is called the Lambertian reflectance value; it also happens to be the cosine-weighted integral of the Lambertian BRDF over the upper hemisphere, and it represents the fraction of arriving power that's reflected by the surface. While such a notion might be useful for other kinds of scattering as well, for a general BRDF f_r the ratio of leaving power to arriving power depends on the distribution of arriving power, so reflectance becomes a function of both the BRDF and the light field. We'll have no use for this more general notion.

By the way, one explanation of Lambertian reflectance is that it arises in part from lots of subsurface scattering; in fact, a standard material used as a calibration tool for optics (it has 99% reflectivity over the visible spectrum, with a very nearly exactly Lambertian BRDF, when measured at a scale of a millimeter or so) is **Spectralon**. These reflectance properties are due to its physical structure: It's a porous thermoplastic that generates many subsurface reflections in the first few tenths of a millimeter. Thus, what appears macroscopically to be a Lambertian "surface" reflector is really a complex *sub*surface reflector.

As a final observation about Lambertian surfaces, we note that the argument above describing the difference between the BRDF and the probability of photon scattering tells us that if a photon arrives at an ideal, perfectly reflecting ($\rho = 1$) Lambertian surface traveling in direction $-\omega_i$, it leaves in some other direction ω_o, which can be thought of as being drawn from some probability distribution with probability density function p. The distribution p is given by

$$p(\omega_o) = \frac{1}{\pi}(\omega_o \cdot \mathbf{n}). \tag{27.16}$$

Here are a few common statements about Lambertian reflection, with commentary.

Lambertian reflection scatters light equally in all directions.	Too vague to be meaningful.
The Lambertian BRDF is constant.	True. $f_r(\omega_i, \omega_o)$ is a constant function of both ω_o and ω_i.
A photon arriving at a Lambertian surface from anywhere is equally likely to scatter in any direction.	False. The probability of scattering in direction ω_o is proportional to $\omega_o \cdot \mathbf{n}$.
A photon leaving a Lambertian surface in direction ω_o is equally likely to have come from a source in any direction ω_i.	Half true. If the surface is bathed in a uniform light field with equal radiance in every direction arriving at the surface, then this statement is true. If the surface is illuminated only by a narrow beam from a laser, then the incoming light *has* to have come from that small range of directions.

27.5.3 The Phong and Blinn-Phong Models

You've already seen two forms of the Phong model: the first in Chapter 6, where light was measured in some ill-defined units of "intensity," with values ranging from 0 to 1, and the second in Section 14.9.3, where actual physical units were used, and the constants had been adjusted so that the model was energy-conservative provided that the sum of the specular and diffuse constants was no greater than 1. In addition, the latter form eliminated the so-called "ambient" term, which was an ad hoc construct that was included to simulate the effects of multi-bounce light transport in a scene.

The general form of the simplified Blinn-Phong BRDF from Chapter 14 is given by

$$f_s(\omega_i, \omega_o) = \frac{k_L}{\pi} + k_G \frac{8+s}{8\pi} z^s, \text{ where} \tag{27.17}$$

$$z = max(0, \mathbf{h} \cdot \mathbf{n}) \text{ and} \tag{27.18}$$

$$\mathbf{h} = \frac{\omega_i + \omega_o}{2}. \tag{27.19}$$

In Equation 27.17, $\mathbf{h}$ is called the **half-vector** and k_L and k_G are the Lambertian and glossy reflectances, respectively, and may range from 0 to 1. The model is energy-conservative if $k_L + k_G \leq 1$.

Blinn's actual model was derived from physical considerations motivated by the microfacet models we'll discuss shortly, and also included a factor for the Fresnel equations, but we'll omit those details for now.

27.5.3.1 Historical Notes

The original Phong model described the reflected light in terms of "intensity," which was not carefully defined, and included a third term, for reflection of "ambient light," that is, all the light in the scene that underwent multiple reflections until it was widely diffused. Thus, you'll sometimes encounter reflection models that use an ambient, a diffuse, and a specular reflection constant, as you saw in Chapter 6.

Phong's original model also had just one constant for the specular term (which we call the glossy term); this meant that reflectance was independent of wavelength, and a white directional light tended to produce a white highlight, no matter what the material. This wavelength independence is characteristic of many insulators, but does not hold for conductors in general (e.g., look at the reflection of a white light in a gold ring). The specular reflectance was therefore given a wavelength dependence (typically by specifying red, green, and blue reflectance values). The diffuse reflectance was similarly extended to include RGB variation (e.g., a red shirt reflects lots of red light, and little blue or green light). And finally, the intensity of light was known to fall off quadratically as a function of distance from the light source (although this notion was problematic for "directional light sources" that were assumed to be "at infinity"!). Thus, for many years the "standard" lighting model looked like

$$I = k_a I_a + f(d) I \left[k_d (-\omega_\mathrm{i} \cdot \mathbf{n}) + k_s, (\mathbf{n} \cdot \mathbf{h})^{n_s} \right], \tag{27.20}$$

where the terms labeled I are all "intensities," the k factors are the constants associated with ambient, diffuse, and specular reflection (we've folded the ambient, diffuse, and specular "color" into these for simplicity), $\mathbf{h}$ is the average of the vector to the light and the vector to the eye, n_s is the specular exponent, d is the distance from the light to the point P, and

$$f(d) = \min\left(1, \frac{1}{a + bd + cd^2}\right) \tag{27.21}$$

was a "quadratic falloff" term, although in practice c was often set to zero, and there's no reasonable explanation for the "min" in the expression except that "things got dark too fast otherwise." When there were multiple light sources, the bracketed term in Equation 27.20 was repeated once per source.

From a modern viewpoint, this entire model, especially the "ambient term," was a terrible thing: Instead of solving the underlying problem (light transport), you apply a "hack" in a completely different area (scattering at a point). But from the point of view of the engineering of the day, it was a reasonable choice: Doing accurate light-transport computations was clearly beyond the capacity of the computers of the day, while evaluating the more straightforward solution provided by Phong's model was relatively simple and drastically improved the empirical results. But you should not be fooled into thinking that the model has any physical basis.

There's also a terminology challenge: The computation of the amount of light reflected from a surface was sometimes called "lighting" or "illumination,"

although the standard interpretation of these words as descriptions of the light *arriving* at the surface was also common. The "lighting model" was typically evaluated at the vertices of a triangular mesh and then interpolated in some way to give values at points in the interior of the triangle. This latter interpolation process was known as **shading,** and you'll sometimes read of **Gouraud shading** (barycentric interpolation of values at the vertices) or **Phong shading,** in which, rather than interpolating the values, the component parts were interpolated so that the normal vector was reestimated for each internal point of each triangle, and then the inner product with the incoming light vector was computed, etc. In each case, the desire was to reduce the artifacts arising from computing the scattering only at the triangle vertices. One of the problems, for instance, with Gouraud shading was that the different rates of intensity variation in two triangles that shared an edge led to an enhanced perception of that edge through Mach banding (see Sections 1.7 and 5.3.2) rather than causing the edges to disappear. Thus, although the quality of the approximation to the ideal was good when measured in physical terms ("How different is the intensity from the true one?"), in perceptual terms ("How different does this surface appear from the true surface?") it was not.

Nowadays we refer to shading and lighting differently: The description of the outgoing light in response to the incoming light is called a **reflection model** or **scattering model,** and the program fragment that computes this is called a **shader.** Because of the highly parallel nature of most graphics processing, the scattering model is usually evaluated at every pixel, often multiple times, and the "shading" process (i.e., interpolation across triangles) is no longer necessary; furthermore, so many triangles are subpixel in size that this interpolation would never be used anyhow. So the modern use of the word "shader" is unlikely to lead to confusion.

27.5.4 The Lafortune Model

The Phong model expressed the specular component of the BRDF as a cosine power (specifically, as a power of the cosine of the angle between the outgoing direction and the mirror-reflection direction). Letting $R(\omega, \mathbf{n})$ denote the mirror-reflection direction of ω at a surface with normal $\mathbf{n}$—in the case where $\mathbf{n} = \begin{bmatrix}0 & 0 & 1\end{bmatrix}^{\mathbf{T}}$, $R(\omega, \mathbf{n})$ is just $\begin{bmatrix}-\omega_x & -\omega_y & \omega_z\end{bmatrix}^{\mathbf{T}}$—the glossy part of the Phong BRDF is simply

$$f_r(P, \omega_i, \omega_o) = C(\omega_o \cdot R(\omega_i, \mathbf{n}))^e, \tag{27.22}$$

where C is a normalization constant. This BRDF is evidently reciprocal in the case where $\mathbf{n} = \begin{bmatrix}0 & 0 & 1\end{bmatrix}^{\mathbf{T}}$: Swapping ω_o and ω_i merely negates the x- and y-coordinates of each vector, thus leaving the dot product unchanged.

Inline Exercise 27.2: Why does the reciprocality in the case where $\mathbf{n} = \begin{bmatrix}0 & 0 & 1\end{bmatrix}^{\mathbf{T}}$ prove that the formulation is reciprocal in all cases?

Lafortune et al. noticed that measured BRDFs tended to have lobes that were not necessarily centered about the mirror direction, and that sometimes they appeared to have multiple lobes. By taking a sum of Phong-like terms, but with varying substitutes for the mirror direction, they generalized to produce a much richer model, based on a collection of lobes centered at k different vectors

$\{\omega_k : k = 1, \ldots, n\}$,

$$f_s(P, \omega_i, \omega_o) = \frac{\rho_d}{\pi} + \sum_{k=1}^{n} (\omega_o \cdot \omega_k)^{e_k}, \tag{27.23}$$

where ρ_d is the diffuse reflectivity. For this to be reciprocal requires that the vectors ω_k be expressed as term-by-term multiples of ω_i so that ω_1, for instance, is

$$\omega_1 = (\omega_{i,x} a_{1,x}, \omega_{i,y} a_{1,y}, \omega_{i,z} a_{1,z}). \tag{27.24}$$

Alternatively, one can express this by building a diagonal matrix A_1, and then saying that $\omega_k = A_1 \omega_i$. Then the Lafortune model ends up being

$$f_s(P, \omega_i, \omega_o) = = \frac{\rho_d}{\pi} + \sum_{k=1}^{n} (\omega_o^{\mathbf{T}} \mathbf{A}_k \omega_i)^{e_k}, \tag{27.25}$$

where the fact that the matrices $\mathbf{A}_k$ are all diagonal guarantees that the BRDF is reciprocal.

Inline Exercise 27.3: Quickly verify the preceding claim. Now suppose that $\mathbf{A}_1$, instead of being diagonal, represents rotation through 90° about the z-axis. Show that the resultant BRDF is *not* reciprocal. Conclude that the Lafortune BRDF is reciprocal if and only if all the matrices $\mathbf{A}_k$ are symmetric.

In practice, since the Lafortune model only uses diagonal matrices, it makes much more sense to just store the three diagonal entries than the whole matrix, and to treat the matrix-vector multiplication as a term-by-term multiplication.

The Lafortune model is very general. In fact, it's possible to approximate almost any conservative, reciprocal function on $\mathbf{S^2} \times \mathbf{S^2}$ by using a large enough value of n. But to get a good fit may require a very large n indeed.

To add spectral dependence to the BRDF representation, we need to let the diagonal matrices $\mathbf{A}_k$ (or their three diagonal entries) be functions of wavelength; this is typically done with RGB values.

The Lafortune model is a hybrid. It's based on a phenomenological model (Phong), but it is motivated by a rather different kind of phenomenon: the appearance of measured BRDFs! In some sense, the Lafortune model can be seen not as a model of light scattering, but as a model of a class of functions, with the property that observed BRDFs tend to be representable with relatively few coefficients, and hence be amenable to rapid evaluation.

27.5.5 Sampling

We've described the BRDF for the mirror and Lambertian and Blinn-Phong models in a form where, given ω_i and ω_o, you could easily compute $f_s(\omega_i, \omega_o)$. But in ray tracing/path tracing and photon mapping, which we'll describe in more detail in Chapters 31 and 32, there are two other computations we need to perform.

For photon mapping, we're given ω_i and we need to randomly select a vector ω_o with probability density proportional to $f_s(\omega_i, \omega_o)(\omega_o \cdot \mathbf{n})$. (Probability densities are described in detail in Chapter 30; for now it's sufficient to think, "We want to pick a vector ω_o more often if $f_s(\omega_i, \omega_o)(\omega_o \cdot \mathbf{n})$ is large, and less often if it's small.")

For mirror reflection, this is easy: We always return ω_r, the mirror-reflected version of ω_i. For Lambertian scattering, we need to pick a direction on the hemisphere, favoring the North Pole, and fading off in probability as we approach the equator. Fortunately, using the Average Height principle, it's not too difficult to do this. Section 30.3.8 gives the details.

For Blinn-Phong scattering, things are not so simple. Although it's possible to sample directly from the BRDF by doing some careful computations, that's only because of the nice power-of-a-cosine form; by the time other factors, like the Fresnel term, get included such direct sampling is no longer possible. Far better is to use an approach like that of Lawrence [Law04], which approximates the BRDF with terms that are amenable to efficient sampling.

For ray tracing/path tracing, we have a similar problem, except that we're given ω_o and want to select ω_i with density proportional to $f_s(\omega_i, \omega_o)$. And for direct computation of the reflectance integral, we may want to sample in proportion to $f_s(\omega_i, \omega_o)(\omega_i \cdot \mathbf{n})$.

The same arguments apply in these cases, except that in the Lambertian case for the ray tracing/path tracing computation, instead of using the cosine-weighted BRDF, we just need to sample in proportion to the BRDF, which is constant. In other words, we just need to pick points uniformly on the hemisphere, which is easy with the cylinder-sphere projection theorem.

27.6 Measured Models

Phenomenological models tend to approximate well those things that we, as humans, recognize as "phenomena." But it's possible that other aspects of scattering, when combined with light transport, produce other "phenomena" as well, and if we suppress those aspects, these secondary phenomena will never be simulated. The only way to know is to have a ground-truth representation of the scattering, and compare results of simulations that use this ground truth to those that use either phenomenological or physically based approximations to it, and see whether the results differ significantly.

One such ground-truth representation is provided by the full BRDF measurements made by Matusik et al. [MPBM03] of about one hundred isotropic materials. For an isotropic material, the BRDF, represented in polar coordinates, depends only on the *difference* between the longitude coordinates of ω_i and ω_o, so the data can be tabulated in a three-index table (two latitudes, one longitude difference). Tabulated at approximately one sample per degree, these tables have many entries $(90 \times 90 \times 180)$, each of which is an RGB triple. (The sampling near glossy highlights is deliberately somewhat denser so that very shiny materials can be accurately represented.)

Others have measured various anisotropic materials [War92], texture characteristics of surfaces [DvGNK99], and more complex data like subsurface scattering distribution [JMLH01], and have developed image-based approaches to measuring BRDFs without the high cost of a gonioreflectomer [MLW+99].

One value of these measurements is that they can be used to compare the expressive power of various BRDF models: if optimally adjusting all the parameters of the Blinn-Phong model, for instance, only allowed you to get within 5% of the measured values, and that only for, say, 90% of all possible (ω_i, ω_o) pairs, you might conclude that the Blinn-Phong model was not sufficiently rich to

represent real materials decently. Matusik in fact took this approach in comparing two BRDF models, the first due to Ward [War92] and the second to Lafortune [LFTG97]; he found that the Lafortune model was better able to fit the data in many cases, but even so, there were cases for which the average error was as large as 20%, where the difference was measured as a difference of *logarithms,* to discount somewhat the overwhelming effect of glossy peaks in the BRDF.

One drawback to using measured BRDFs in rendering is the cost of performing effective sampling. While Matusik describes techniques for this, they require a large amount of extra precomputed data and are still slow compared to those few models for which explicit sampling techniques are known.

There are some limits to using measured BRDFs. We can only render images of scenes for which we know the BRDFs of all materials, and measuring the BRDF of a material is nontrivial; gathering observations at grazing angles is particularly challenging. And we can only gather the BRDF of a material that already exists. We can't create new BRDFs by adjusting parameters, as we can do with the various physically based models described below. Finally, there's the problem that the gathered data may well contain errors, errors that can make the observed BRDF turn out to be physically unrealistic. Matusik handles this in part by projecting every measured BRDF onto the reciprocal-BRDF subspace, by replacing $f_s(\omega_i, \omega_o)$ with the average of $f_s(\omega_i, \omega_o)$ and $f_s(\omega_o, \omega_i)$, thus ensuring reciprocity, and by discarding certain outlying measurements.

27.7 Physical Models for Specular and Diffuse Reflection

We now turn to physically based models of reflective scattering. There is a choice to be made in attempting to explain scattering phenomena: Should we use **physical optics,** based on the wave theory of light, in conjunction with a geometric model, to determine the scattering, or **geometric optics,** in which the reflection of light by surfaces is determined entirely by a billiard-ball-bounce model, in which the arriving light reflects from the surface in the mirror direction? At first glance, the geometric optics approach seems destined for failure; after all, not every surface is mirrorlike. This can, however, be addressed by examining the interaction of light with a *rough* surface, in which the reflection is mirrorlike, but the geometry is extremely complex, consisting of many tiny reflecting facets oriented in many possible directions. Since the roughness can be described probabilistically, this approach is actually feasible. In contrast, the physical optics approach presents enormous computational challenges, in the sense that to apply it, we must apply Maxwell's equations to relatively complex situations, where any hope of an easily expressed formula is lost; our best hope is for rapid numerical solutions of the equations. We'll return to this after examining the geometric optics approaches.

Geometric optics is really only appropriate when the small reflecting parts of the surface are large compared to the wavelength of the incident light. Since the incident light that interests us is in the visible range, we can say that the wavelength is about 0.5 to 1.0 microns; this means that the microfacets must be at the very least 1 micron in size. When you recall that a human hair is on the order of 15 microns in diameter, and that it's easy to feel a single hair on a flat surface, you realize that the geometric optics assumption for ordinary materials is just barely reasonable: 15 micron sandpaper feels about like newsprint; 2 micron sandpaper

is used to smooth out automotive finishes and to polish sharp knives. Thus, materials whose roughness is between that of a highly polished metal and a piece of newsprint are likely to contain scratches that are on the scale of a few wavelengths of light. Despite this, the geometric optics approach seems to make good predictions in practice at this scale. We'll describe the main ideas of the geometric optics approach in the next several sections.

We do so with a caution, however: Given the scale disagreements (facets must be large compared to the wavelength of light, but in practice are quite close to it), these models are at best weak approximations of the underlying phenomena. Recent careful measurement work has shown the weakness of the approximations [Lei10].

27.8 Physically Based Scattering Models

The underlying physics of reflection from a flat surface depend on the electrical properties of the atomic structure of the material, some of which we described in Chapter 26. In particular, metals, which tend to make the best mirror reflectors, have many free electrons that float about the surface, creating an almost perfectly planar sheet of constant electrical potential with which the electromagnetic light-wave interacts. The Fresnel equations determine the degree of reflection for light of varying polarizations; in graphics, we typically assume unpolarized light, and thus average the perpendicular and parallel terms of the Fresnel equations. We'll review these equations, and describe how they're applied in practice, since they are part of all the physically based scattering models.

As we said, the assumption, in the physical computations that support these reflectance models, is that the reflecting surface is large compared to the wavelength of the arriving light, or else diffraction will start to dominate. The mirror-plane model can also be used to compute the reflection from a mirror surface that's nonplanar, provided that its curvature is not too great; to compute the reflection at a point Q, we use the normal vector $\mathbf{n}(Q)$ to compute the mirror direction just as before. If the curvature is too large (i.e., if the normal vector changes too fast), then again the "sheet of constant potential" model fails, and diffraction starts to dominate. A radius of curvature (in any surface direction) that's near or lower than the wavelength indicates a place where the mirror model is no longer appropriate. It's worth noting that in polygonal models, the curvature at every point of every edge is infinite. This is typically ignored in ray tracers, where a ray hits either one facet or the other, and proximity of the ray to an edge is ignored. If the ray actually hits an edge precisely, it may be ignored or treated as lying on one of the two adjacent surfaces. The results look correct enough that they have not generally been a point of concern.

27.8.1 The Fresnel Equations, Revisited

In Chapter 26, we saw that at a surface between dielectric materials such as water and air, or glass and air, the amount of light reflected and transmitted depended strongly on the angle of the arriving light. Under the assumption that the arriving light was unpolarized, the fraction of light energy reflected is a function of the refractive indices of the two materials and the incident angle $\theta_i = \cos^{-1}(\omega_\mathrm{i} \cdot \mathbf{n})$, and the angle of the transmitted ray is θ_t. The Fresnel reflectance R_F is the average

of the parallel- and perpendicular-polarized terms, R_s and R_p, which are given by

$$r_p = \frac{n_2 \cos\theta_i - n_1 \cos\theta_t}{n_2 \cos\theta_i + n_1 \cos\theta_t} \tag{27.26}$$

$$R_p = r_p^2 \tag{27.27}$$

$$r_s = \frac{n_1 \cos\theta_i - n_2 \cos\theta_t}{n_1 \cos\theta_i + n_2 \cos\theta_t} \tag{27.28}$$

$$R_s = r_s^2. \tag{27.29}$$

Recall that θ_i and θ_t are related by Snell's law:

$$\sin(\theta_t) = \frac{n_1}{n_2} \sin\theta_i. \tag{27.30}$$

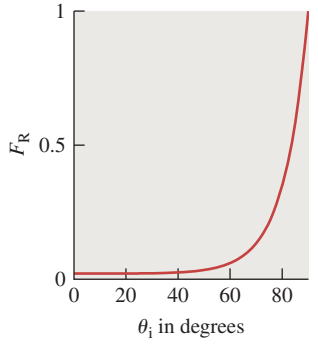


Figure 27.12: The Fresnel reflectance F_R as a function of θ_i, for an air-to-water interface

For an air-water interface, we have n_1, the refractive index of air, is approximately 1.0, while that of water is approximately 1.33. The plot of R_F as a function of θ_i is shown in Figure 27.12.

As you'll observe, the function is nearly constant until we approach grazing, at which point it rises rapidly. If you plot F_R for some other ratios of refractive indices (see Exercise 27.5), you'll see that this characterization is quite general: nearly constant for small angles, a sudden rise near grazing angles.

For metallic surfaces, the formula for R_F is somewhat more complex, but it exhibits the same general characteristics.

Schlick [SCH94] observed that for *metallic* surfaces R_F could be well approximated by a simple expression, and others have observed that the approximation works reasonably well even for nonmetallic materials. The approximation is

$$R_F(\theta_i) = R_F(0) + (1 - R_F(0))(1 - \cos\theta_i)^5, \tag{27.31}$$

where $R_F(0)$ is the reflection at normal incidence ($\theta_i = 0$) and $\theta_i = \cos^{-1}(\omega_i \cdot \mathbf{n})$ is the angle of incidence. When the cosine is 1, we get $R_F(0)$; when the cosine is 0, we get 1.0.

Inline Exercise 27.4: There's usually no reason to compute θ explicitly, since many formulas involve the cosine or sine of θ rather than θ itself. Rewrite Schlick's approximation in terms of ω_i and $\mathbf{n}$ rather than θ.

For insulators, $R_F(0)$ tends to be small, so there's large variation in R_F with angle θ_i, leading to a pronounced Fresnel effect. For conductors, $R_F(0)$ tends to be large (typically greater than 0.5), so the Fresnel effect is less pronounced.

Note that the index of refraction and the coefficient of extinction depend on wavelength (although they have not been tabulated for many materials, which is a problem); this means that the Fresnel reflectance is also a function of wavelength. For many metals, this dependence is considerable. For gold, for instance, the extinction coefficient drops substantially above about 500 nm, while the index of refraction rises steadily above about 500 nm, which together give gold its characteristic yellow appearance. For insulators, the refractive index is nearly constant with respect to frequency, causing highlights on insulators to be the color of the incoming light.

Actually *using* the Schlick approximation requires that you know $R_F(0)$. But the original Fresnel equations (for dielectrics) give you this value:

$$R_F(0) = \left(\frac{n-1}{n+1}\right)^2, \text{where} \tag{27.32}$$

$$n = \frac{n_2}{n_1}. \tag{27.33}$$

Inline Exercise 27.5: Verify this formula.

In graphics, most objects sit in air, so $n_1 = 1$, and the formula is slightly simpler: You just replace n with n_2.

Inline Exercise 27.6: Show that in the case of conductors, the correct form is

$$R_F(0) = \frac{\kappa^2 + (n-1)^2}{\kappa^2 + (n-1)^2}, \tag{27.34}$$

using the approximation of the Fresnel reflectance for conductors.

We sometimes want to render things like an underwater view of the surface of a swimming pool. In this case, the light rays are traveling in a medium of large refractive index, and the "other" side is air, which has lower refractive index. Of course, Fresnel's equations still hold, as does the Schlick approximation, but to make it work you must use θ_t, the angle of the transmitted ray (the one in the air, not the water) as the argument. The result is that the Fresnel reflectance approaches 1.0 as θ_i approaches the critical angle, which is generally much less than 90°. (For angles greater than the critical angle, R_F remains at 1.0.)

Inline Exercise 27.7: If you have looked up at the pool's surface while swimming, explain the appearance of the surface from below, and the difference in its appearance from above, by considering the Fresnel reflectance and the critical angle for total internal reflection.

27.8.2 The Torrance-Sparrow Model

The Phong model predicts that a surface illuminated in direction ω_i will reflect light in all directions, with a peak in the mirror-reflection direction ω_r. In actual observations of nonmirror materials, that's not the peak direction. Torrance and Sparrow provided a model to explain this off-mirror peak: They imagined that the surface was made up of **microfacets,** each of which was a tiny mirror reflector, but with random orientations. The microfacets were assumed to pair up to form "V" shapes, with identical slopes of each side of the "V" so that an edge-on slice of the material appeared as a collection of grooves of varying depth, all with their tops at the same height, as in Figure 27.13.

Note that we are implicitly assuming that the BRDF we're estimating is for a measurement area that's substantially larger than the scale of a single microfacet, or the analysis, which is based on average microfacets, is no longer reasonable.

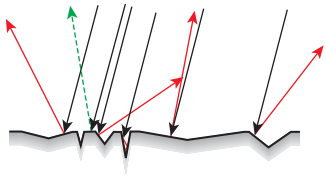


Figure 27.13: The symmetric grooves all have their tops at the same height. Light arriving at an angle (downward-pointing black arrows) can be reflected in the mirror direction (the dashed green arrow), or reflect back toward the source or in other directions (red).

For the same reason, when we use a measured BRDF in rendering it's important that an imager pixel correspond to a surface region whose area is as large or larger than the area used in gathering the BRDF in the first place.

Inline Exercise 27.8: The wavelength of visible light is between one-half and one micron; it's convenient to treat it as about one micron for back-of-the-envelope estimation. To prevent diffraction effects from being significant, microfacets should be at least a couple of wavelengths (say, five) in their minimum dimension. In a surface that's fairly flat (most microfacets have slope less than 45°), we can imagine each microfacet is a small disk or square, so its projected length, in any direction, is at least $.71 \approx \cos(45°)$ times its minimum dimension.
(a) Approximately how many such microfacets can fit into a 1 mm-diameter disk?
(b) If you had that many small mirrorlike facets in such a disk and illuminated them with a laser pointer whose beam covered just that disk, what would the pattern of reflected light look like? Get a laser pointer, a piece of polished metal, and a piece of white paper to "catch" the outgoing light and see whether the reflected light follows the predicted pattern.

For illumination arriving along the normal direction ($\omega_i = \mathbf{n}$), the scattered light goes in many directions: If the grooves are all shallow, most of it reflects back in the normal direction; if they're deep, there's much more scattering in off-normal directions. But for illumination arriving in an off-normal direction, multiple phenomena combine to generate the scattering pattern.

- Each peak "shadows" the next valley to some degree, so the amount of light reflected from a microfacet is no longer proportional to its area.
- The reflected light may hit yet another microfacet, and be further reflected, and thus not continue in the reflected direction; this is called **masking.**
- For certain illumination directions, we can get both masking *and* shadowing, as shown in Figure 27.14.

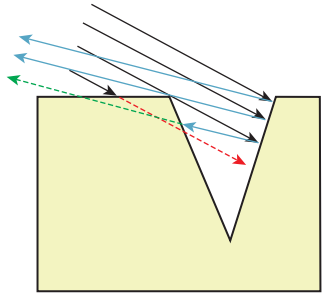


Figure 27.14: Incoming light misses the bottom part of the right-hand side of the V groove, which is shadowed (right-pointing red dashed ray); some of the light reflected from that right-hand side is masked by the left-hand side (left-pointing green dashed ray). (Courtesy of Ephraim Sparrow, "Theory for Off-Specular Reflection from Roughened Surfaces" by K. Torrance and E.M. Sparrow. It was printed in Journal of the Optical Society of America*, Vol. 57, No 9, 1105–1114, September 1967. Redrawn.)*

The detailed analysis of the effects of masking and shadowing, for various distributions of microfacet orientations, is quite complex [TS67], but the analysis predicts three important phenomena: The first is **backscattering,** in which some incoming light from off-normal directions is reflected back toward the source; the second is the off-specular peak—the peak value of the BRDF occurs not at the mirror-reflection direction, but at a more-grazing (i.e., less normal) direction. The third is that the value of the BRDF at grazing angles remains finite, which is in accord with experimental observation, but not with prior microfacet models that didn't account for masking and shadowing.

By the way, it's worth experimenting with a piece of ordinary office paper to see how very specular are the near-grazing-angle reflections from a supposedly matte surface. If in front of your face you hold a piece of paper by its bottom edge so that the top falls down, forming a "hill" that you can look across, and then you look at some fairly bright scene (the view out an office window on a sunny day works well), you can see quite distinct features reflected in the paper at or near the silhouette edge.

The Torrance-Sparrow model, like the Phong model, combines a diffuse term with the glossy term, and takes into account the Fresnel equations in adjusting the reflectivity as a function of incoming-light angle. The distribution of microfacet slopes is assumed to be exponential: The probability density at slope α is proportional to $\exp(-c^2\alpha^2)$, where the constant c is a parameter of the model.

The parameters for the model are the index of refraction (which is wavelength-dependent), the slope-distribution constant c, and the diffuse and glossy constants k_d and k_g, although they use the ratio $g = k_g/k_d$ as well, using a complex-number version of the index of refraction to represent both the ordinary index of refraction and the absorption. Torrance and Sparrow report that $c = 0.05$ works well, and approximately agrees with experimental observations of $c = 0.035$ and $c = 0.046$ for ground glass surfaces. They plot the predicted results for aluminum and magnesium oxide, and show good agreement with the observed data.

27.8.3 The Cook-Torrance Model

Cook and Torrance [CT82] developed an extension of the Torrance-Sparrow model that explicitly took into account the different nature of diffuse reflection (usually involving subsurface scattering, or multiple scattering from a sufficiently rough surface) and specular reflection, especially from metals, which is an almost entirely surface-based phenomenon. Since specular reflection from microfacets is again used to model glossy reflection, anything said about specular reflection here also applies to glossy reflection. The specular-diffuse difference means that the diffusely reflected and specularly reflected lights from a single surface may have quite different spectral distributions; plastics, for instance, tend to specularly reflect light with approximately the same spectral distribution as the illuminant, while metals (think of copper and gold) tend to have substantial spectral variation in reflectivity, so the reflected light "takes on the color of the material."

As with the Phong model, there are three parts: an ambient, a diffuse, and a specular term. The ambient term is considered to be an average of the diffuse and specular effects due to light coming from many different directions in the scene, which is assumed to be uniform. Because of this, the color of the ambient term is a combination of the diffuse and specular colors (where we are using the term "color" as a shorthand for "spectral distribution"). The diffuse term is assumed Lambertian. The complete model, ignoring the ambient term, has the form

$$f(\omega_\mathrm{i}, \omega_\mathrm{o}, \lambda) = sR_s(\omega_\mathrm{i}, \omega_\mathrm{o}, \lambda) + dR_d(\lambda), \text{where} \tag{27.35}$$

$$R_s(\omega_\mathrm{i}, \omega_\mathrm{o}, \lambda) = \frac{F(\omega_\mathrm{i}, \lambda)}{\pi} \frac{DG}{\pi(\mathbf{n} \cdot \omega_\mathrm{i})(\mathbf{n} \cdot \omega_\mathrm{o})}, \tag{27.36}$$

where s and d are the amounts of specular and diffuse reflectivity, respectively, R_s and R_d are the (spectral) BRDFs for specular and diffuse reflection, respectively, F is the Fresnel term, D is the microfacet slope distribution, and G is a geometric attenuation factor that accounts for masking and shadowing of facets; we've omitted the arguments for both D and G for now. The entire expression is evaluated at a point P of a surface with normal vector $\mathbf{n}(P)$, for which we'll write $\mathbf{n}$ for simplicity.

It's easiest to express the geometric term in terms of the half-vector

$$\mathbf{h} = \mathcal{S}(\omega_\mathrm{o} + \omega_\mathrm{i}). \tag{27.37}$$

Using **h**, the geometric attenuation is

$$G(\omega_i, \omega_o) = \min\left\{1, 2\frac{(\mathbf{n}\cdot\mathbf{h})(\mathbf{n}\cdot\mathbf{v})}{(\mathbf{v}\cdot\mathbf{h})}, 2\frac{(\mathbf{n}\cdot\mathbf{h})(\mathbf{n}\cdot\omega_i)}{(\mathbf{v}\cdot\mathbf{h})}\right\}. \tag{27.38}$$

The slope distribution function D describes the fraction of facets that are oriented in each direction; letting $\alpha = \cos^{-1}(\mathbf{n}\cdot\omega_i)$, we can write D as a function of α, thus implicitly assuming that it's symmetric around the surface normal, that is, that the microfacet distribution is isotropic. Cook and Torrance use the **Beckmann distribution function**

$$D(\alpha) = \frac{1}{m^2\cos^4\alpha}e^{-\left[\frac{\tan\alpha}{m}\right]^2}, \tag{27.39}$$

which has the single parameter m.

Inline Exercise 27.9: Convince yourself that if m is very small, then most facets are nearly perpendicular to the normal vector **n**, while if m is large, the surface is very rough with sharply angled facets.

They also note that a surface may be rough at several different scales; thus, the function D could be a weighted sum of multiple terms like the one in Equation 27.39.

Finally, they model the spectral distribution of the reflected light. For diffuse reflectance, they use measured reflectance spectra, which are typically measured with illumination at normal incidence; they note (as in our discussion of Fresnel reflectance above) that the diffuse reflectance spectra for most materials do not vary substantially for incidence angles up to 70° off normal, and even then vary relatively little. They therefore use the normal-incidence reflectance spectrum as the diffuse reflectance spectrum at all angles.

Inline Exercise 27.10: There are man-made materials that are designed to have reflectance spectra that vary with viewing angle; one example is a diffraction grating. Try to think of a diffuse material with this property. Hint: textiles.

For the specular term, they model the angle dependence of the reflectance spectrum as coming entirely from the Fresnel term, as above. The results represented a huge step forward in computer graphics: Because the color of the specular highlights could now be different from that of either the underlying surface's diffuse color *or* the color of the incident light, it became possible to plausibly simulate a much wider variety of materials (see Figure 27.15).

27.8.4 The Oren-Nayar Model

Closely related to the microfacet models for specular reflection is the Oren-Nayar model [ON94] for reflection from rough surfaces such as unglazed clay pots, tennis balls, or the moon's surface. Oren and Nayar observed that these diffuse reflectors did not actually follow Lambert's law very well at all; in particular, the areas near the silhouette tended to be much lighter than Lambertian reflectance would predict. This is particularly obvious with the moon, whose brightness appears almost uniform across the surface (except for surface-texture features). They suggest that this brightness near the silhouettes can be explained by noting that the

Figure 27.15: A single vase, illuminated by two lights, made of 12 different materials whose reflectance properties were determined with the Cook-Torrance model. (Courtesy of Robert Cook, ©1981 ACM, Inc. Reprinted by permission.)

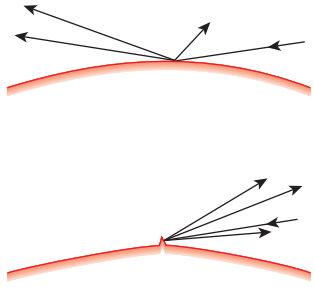


Figure 27.16: (Top) Near the edge of a smooth diffuse surface, very little light would be scattered back toward the viewer because the off-normal incidence of the light makes the irradiance small. (Bottom) If a small piece of surface near the silhouette is oriented more perpendicular to the incident light direction, this surface piece will scatter much more light in all directions, including the direction back toward the viewer.

surface roughness means that even when we are looking near a silhouette, *some* parts of the surface are facing back toward us, and thus reflect more light than might be expected (see Figure 27.16). In particular, they apply essentially the Torrance-Sparrow-Cook idea of microfacets, but in a situation where they assume that each microfacet acts like a *Lambertian* reflector rather than a mirror.

Furthermore, they consider not only single-bounce interactions of illumination with microfacets, but multiple inter-reflections, observing that for a quite rough surface, illuminated at a nearly grazing angle and viewed from the opposite side from the illumination, each lighted facet is invisible to the viewer, but the facets that are *not* directly lit may nonetheless be visible. If you look east toward a mountain range at dawn, you cannot see the sunlight on the east sides of each mountain, but the *west* side of some mountains will be illuminated by light reflected from the east side of more westerly mountains (see Figure 27.17).

The expression for the radiance (assuming a Gaussian distribution of facet slopes) from a facet is a rather messy integral. Oren and Nayar evaluated this integral for many different directions of arriving light, viewer direction, and Gaussian shape parameters, and developed a much simplified form that's suitable for more rapid computation of approximately correct values. The result is given in terms of θ_i, the angle between ω_i and $\mathbf{n}$, θ_o, the angle between ω_o and $\mathbf{n}$, and ϕ, the difference in azimuth between the incoming and outgoing light. If light arrives from the west and leaves to the east, then $\phi = 0$; if it leaves to the northeast, then $\phi = 45°$; if it leaves to the southeast, then $\phi = -45°$, etc. The model has two parameters: the slope-distribution constant σ, where the probability of a facet at angle θ is proportional to $\exp(-\frac{\theta^2}{2\sigma^2})$, and the albedo, ρ, of the surface (which may be a function of wavelength).

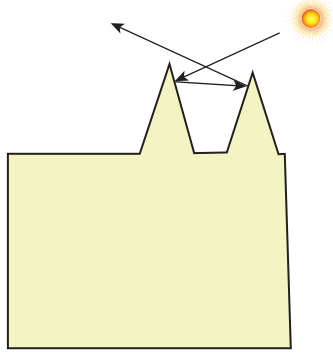


Figure 27.17: The west sides of mountains illuminated by the rising sun will be in shadow, but may be lit by light reflected from the east sides.

The full-generality result is seldom used; instead, the simplified version, which accounts for only single scattering, is

$$L_r(\theta_\mathrm{i}, \theta_\mathrm{o}, \phi; \sigma) = \frac{\rho}{\pi} E_0 \cos(\theta_\mathrm{i})(A + B \max(0, \cos\phi) \sin\alpha \tan\beta), \text{ where} \tag{27.40}$$

$$A = 1.0 - 0.5\frac{\sigma^2}{\sigma^2 + 0.33}, \tag{27.41}$$

$$B = 0.45\frac{\sigma^2}{\sigma^2 + 0.09}, \tag{27.42}$$

and $\alpha = \max(\theta_\mathrm{i}, \theta_\mathrm{o})$ and $\beta = \min(\theta_\mathrm{i}, \theta_\mathrm{o})$.

27.8.5 Wave Theory Models

Until now, all of our models have used geometric optics, ignoring the effects of the wave nature of light (such as interference and diffraction), except for the Fresnel term. One reason for this is that working directly with Maxwell's equations proves to be extremely difficult and computationally expensive. On the other hand, it does predict some effects that geometric optics models miss. The work of He [He93] makes the most compelling case for this, but the underlying physics is beyond the scope of this book; we refer the interested reader to the paper itself.

How important *are* wave effects? They certainly can matter, but as Pharr and Humphreys [PH10], p. 454, note:

> Nayar, Ikeuchi, and Kanade [NKK91] have shown that some reflection models based on physical (wave) optics have substantially similar characteristics to those based on geometric optics. The geometric optics approximations don't seem to cause too much error in practice, except on very smooth surfaces. This is a helpful result, giving experimental basis to the general belief that wave optics models aren't usually worth their computational expense for computer graphics applications.

27.9 Representation Choices

A BSDF can be represented in various ways—as a table of values to be interpolated, as we saw for measured models, or as a sum of "lobes," as in the Lafortune model, or even in a kind of "Fourier decomposition," using spherical harmonics, which are the analog, on the 2-sphere, of the powers of sine and cosine on the circle. It's also possible to represent BSDFs using sums of Gaussians, in wavelet bases, or many other possible forms. Each choice has its advantages and disadvantages, and graphics has not yet arrived at a definitive ideal model.

27.10 Criteria for Evaluation

We've discussed BSDFs and how to represent them with a general bias toward finding models that match measured data well, which certainly seems like a good thing. But we haven't discussed the precise criteria for "matching well." One obvious choice is the L^2 error: If f is an approximation to f_s, we can integrate $(f - f_\mathrm{s})^2$ over all of $\mathbf{S^2} \times \mathbf{S^2}$ to determine the goodness of fit. In the sense that the difference between f and f_s corresponds to the difference in what we see when we look at a directly illuminated piece of the material, this seems to make intuitive sense. But our perception is nonlinear as a function of radiance. A small error in the approximated reflectance at a $(\omega_\mathrm{i}, \omega_\mathrm{o})$ pair where f_s is small is far more perceptually important than the same difference at a place where f_s is large, but they're counted equally in measuring the goodness of fit. This argument suggests that we should

perhaps integrate something like $\log[(f - f_s)^2]$ over the domain, and use *that* as a measure of error. But that only makes sense if the BSDF is sending light to the eye. What if it's reflecting light onto *another* surface that will in turn reflect it to the eye? Then maybe the original L^2 difference is the better measure of goodness. After all, at some distance from the surface (especially if it's at all curved), the fine details of the BSDF are "blurred out" so that the reflected light distribution is fairly uniform, as we'll see in Section 31.20.

Some of the simplest models, like the Phong model, don't fit observed data very well, but they have good empirical characteristics (a large lobe in about the right direction for specular reflection, intuitive parameters). Others, like the Lafortune model, provide better L^2 fit, and have the additional benefit of being amenable to sampling in certain common ways (see Section 27.14). Still others, like the spherical harmonic representation, allow for efficient nonprobabilistic evaluation of the reflectance integral. In deciding which to use, you need to consider your eventual purpose.

27.11 Variations across Surfaces

The BSDF of a surface is typically not constant as a function of position. The BSDF for a piece of paper might be nearly constant, but when it has print on it the printed parts will have much smaller total reflectance. Objects like wood have structural texture like grain at a scale of about 1 millimeter, and further cellular texture at a scale perhaps 100 times smaller. The BSDFs for the different kinds of wood fiber are quite different. Some of the richness of wood comes from strong subsurface scattering by linear structures like wood fibers; if the orientation of these fibers varies, as it does in burls, for instance, this can introduce another kind of variation in the reflectance of the material.

Let's examine two approaches to modeling a wall painted with latex paint, such as you might find in any office. Typically such a surface has some texture, in the nongraphics sense: There are bumps on the order of 0.1 mm, separated by a typical distance of 2 mm. The paint surface, even on the bumps, is reasonably flat at a scale of 10 wavelengths of light, so it's reasonable to use a BSDF representation. We'll assume that the latex paint has a perfect Lambertian BSDF, but we'll need to record, in a texture map, the variation of the albedo from point to point, and the variation of the normal vector. That entails a total of three dimensions of high-resolution texture map (one for albedo, two for the normal vector variation), or perhaps a procedural texture.

Alternatively, we could imagine treating the wall as truly flat, and measuring the BSDF at each point of the wall. On the sides of the bumps, we'd find that the BSDF was different from what it is at the bottoms of valleys, etc.; if we represent each BSDF as a sum of spherical harmonics, say, 50 harmonics, then to represent the entire wall we'd need 50 dimensions of texture map to record each harmonic coefficient. (We're assuming a white paint, to avoid the worry of spectral dependence.)

Clearly the first model is preferable in this case. But if we instead consider something like a granite wall surface, where the material is made up of an aggregate of other materials, each with a different reflectance property, a spatially varying BSDF might be a completely reasonable approach: Perhaps a suitably factored model would be a good solution; perhaps the variation of the BSDF will occur mostly in one or two factors so that the others can be treated as constants, saving a great deal of space.

Inline Exercise 27.11: One of the implicit assumptions in the definition of the BRDF is that it is measured and used at a scale larger than the scale of the largest variation in the underlying material. Thus, it makes some sense to measure the BRDF of granite for use in aerial sensing applications, where a single pixel sensor may record light reflected from many square meters of granite surface, but it does not make sense to use that same BRDF in trying to predict the appearance of a microscopic picture of granite. Suppose that we have modeled some object with local variation in appearance—a piece of paper with printing on it, or a flat metal tray with fingerprints around the rim—and we wish to make a picture of it from a distance so that the entire object will occupy just a few pixels on the imaging sensor. It's natural to use MIP mapping for this.
(a) Argue why it is reasonable to average the spatially varying BRDF over a region of the surface to estimate a BRDF for the larger surface region, at least in the case of the paper and the flat metal tray.
(b) Argue that even in the case of a flat surface, it's *not* generally reasonable to average the model parameters (such as the Phong exponent, or the Cook-Torrance specular color, or the index of refraction), and then use these averaged values to estimate the BRDF of the larger surface region.
(c) Suppose that your surface has a fairly constant BRDF (like the curved tile on a Spanish tile roof), but the underlying surface has substantial curvature at a smaller scale than one imager pixel (i.e., a Spanish tile rooftop that projects to just a few imager pixels). How would you compute a BRDF for the larger surface?

27.12 Suitability for Human Use

One benefit to using an explicit physically or empirically based model in representing a BSDF is that such models often have a few parameters that may be amenable to intuitive understanding. For example, the Phong exponent can be described as representing "shininess," and the diffuse and specular reflectivity as representing the "lightness" of the surface. Of course, the parameters may not match our intuition completely; the Phong exponent, for instance, affects the appearance of a surface a great deal when it's changed from 1 to 2, but hardly at all when it's changed from 51 to 52. (Offering the user an adjustment for the *logarithm* of the Phong exponent proves to be far more intuitive: 0 corresponds to diffuse, and 6 to "very shiny.") Similarly, the index of refraction of a material and its dielectric properties are not intuitively understood by most people, but we can offer an intuitive control that ranges from "metallic" to "plastic" by combining parameters in the Cook-Torrance model [Str88].

One more reason for using models with intuitive parameters is that we sometimes want to measure a material, and then create a new material that's quite similar, but not exactly the same. Fitting a model to the measured data, and then giving the designer intuitive controls to adjust, is far more likely to produce good results than giving the designer the opportunity to edit the measured data directly.

27.13 More Complex Scattering

We've discussed models for scattering from surfaces, which is a fairly good approximation for metals, for instance, but is increasingly inadequate as the materials we encounter become less surfacelike. In this section, we'll briefly discuss volumetric materials, which are sometimes called participating media, and subsurface scattering, which helps determine the appearance of materials like human skin.

27.13.1 Participating Media

We'll now give a very brief description of how light interacts with participating media like colored water, or fog.

When you sit in a dark room on a sunny day, with sunlight streaming through a window, shafts of sunlight typically fill the room. These appear because the sunlight hits tiny particles in the air of the room (usually dust), and is scattered by these particles to the eye. Even though the particles are scarce and small, the intensity of the sunlight is such that the net reflected light may be substantial compared to that reflected by the dark walls of the room. The result is that we "see the beam of light." The same kind of "volumetric scattering" explains the appearance of the rings of Saturn, and the dark regions at the bottom of cumulus clouds. Because we usually consider the air in a room as a medium through which light passes untouched, until it's scattered by a surface, our ordinary model no longer applies. Now the medium (air with dust particles) *participates* in the scattering process, and so the term **participating media** is often used in connection with such situations.

The exact modeling of participating media requires the precise measurement of several physical properties [Rus08]; even with these necessary constants, the associated computations are quite complex. In broad strokes, however, for sparsely distributed scatterers uniformly distributed in a medium (e.g., dust in the air of a room), the light passing through the medium is exponentially attenuated, that is, its radiance, after passing through a distance d in the medium, is multiplied by $\exp(-\sigma d)$ for some small constant $\sigma > 0$. This attenuation is explained by thinking of the probability of a bit of light making it through the whole medium without encountering a particle. Suppose that the probability of making it through one millimeter of the medium is 0.95. Then the probability of light making it through the first millimeter will be 0.95, while the probability of making it through *two* millimeters will be 0.95^2, etc., leading to an exponential decay. For a nonuniform participating medium, the decay rate σ becomes a function of location, and the amount of light exiting the medium ends up being the amount entering, multiplied by a constant that's an integral of a function of σ along a ray.

What we've just described is **absorption,** and it can be used to describe the interaction of light with materials like soot which absorb light and convert it to heat, but hardly emit or reflect it. Figure 27.18 schematically shows absorption at the orange particle just to the right of center. Absorption generally depends on wavelength, so this analysis really only applies to light of a single wavelength.

Three other phenomena arise for general participating media. The first of these is **emission,** shown by the red particle in the bottom left of the figure: Just as in the analysis of light's interaction with surfaces, we can encounter media that glow (think of the liquid in glow sticks, or the heated soot in flame). The second is

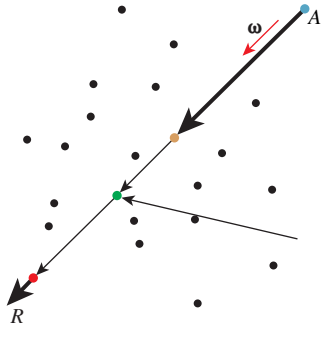


Figure 27.18: The ray R starts at A in direction ω and passes through a participating medium.

outscattering: Light hitting the medium may be scattered in various directions, causing things like the aforementioned shafts of light you see on a sunny day in a dusty room. This outscattering may not be uniform. Typically it's assumed that the outscattering is constant as a function of course-deviation: If the light is traveling in direction ω, the probability of scattering to a new direction ω' is assumed to be a function of $\omega \cdot \omega'$ only. The third phenomenon is inscattering (shown at the green particle, just to the left of center in the diagram): Light scattered from other parts of the medium may arrive at some point and scatter so as to add the light we're already observing. (Outscattered and inscattered light are therefore analogous to surface radiance and field radiance.)

These four effects can be combined to characterize how the radiance $L(P, \omega)$ at some point P in some direction ω is related to the radiance at $P + \epsilon\omega$ in direction ω; dividing the difference by ϵ and taking a limit as $\epsilon \to 0$ gives an equation that L must satisfy, analogous to the rendering equation. Pharr and Humphreys [PH10] and Rushmeier [Rus08] give a great deal more detail on this subject.

The absorption constant σ has units of inverse meters. If light travels a distance of $1/\sigma$ through the absorbing medium, it's attenuated by a factor of e. Perhaps more intuitive is that if the light travels $2.303/\sigma$, it's attenuated by a factor of 10. Even so, σ is not an intuitively easy-to-understand item. An artist is more likely to find $2.303/\sigma$, the distance it takes to decrease by a factor of 10, to be a natural control for absorption.

The coefficient of extinction, κ, mentioned in Chapter 26 and sometimes used as the imaginary part of a complex index of refraction, is closely related to the absorption constant σ: It represents the exponential decay of the electromagnetic wave as it enters a homogeneous material. If we compute the absorption constant for the material at a particular wavelength λ, the two are related by

$$\kappa = \frac{\lambda\sigma}{4\pi}. \tag{27.43}$$

Thus, the explicit dependence of absorption on wavelength is naturally incorporated by using κ. Of course, neither the idea of a complex index of refraction nor the coefficient of extinction is likely to be readily understood by a user; once again, the inverse of κ is likely to be more natural as a user control for material design. And for a material like smoke or fog, the absorption phenomenon is not *really* being governed by index-of-refraction effects, but by larger-scale phenomena, such as light entering and leaving the individual droplets of water in the fog.

27.13.2 Subsurface Scattering

In addition to mirrorlike reflections (and more diffuse reflections arising from multiple microfacets) and volumetric scattering, there's a third kind of scattering that is very common: subsurface scattering.

In human skin, for instance, light meeting the skin at some point P may leave the skin at another point Q. To see this, sit in a dark room facing a mirror and place a small flashlight firmly against your cheek. You will see the skin around the flashlight glowing red because of subsurface conduction of the light. This cannot be modeled by a surface-based BSDF. Instead, we use a bidirectional subsurface scattering distribution function (BSSRDF), which has the form $f_{ss}(P, Q, \omega_i, \omega_o)$, which represents the light leaving in direction ω_o from the point Q in response to light arriving from direction ω_i at point P (see Figure 27.19). It's convenient to

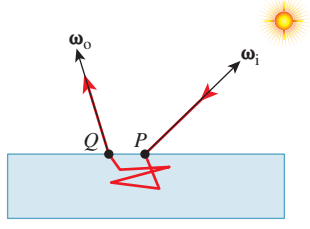


Figure 27.19: The arguments to the BSSRDF. The light enters the material at P from the source pointed to by ω_i, travels along the red path, bouncing inside the material, and exits at Q in direction ω_o.

simplify things by assuming that the material is homogeneous and anisotropic so that f_{ss} depends only on the distance from P to Q rather than on their absolute positions in the material; this substantially reduces the memory required to represent the scattering. But for many materials (like the palm of your hand), the geometric structure of the subsurface material is neither isotropic nor homogeneous. Things like veins, arteries, capillaries, muscle, cartilage, and fat all affect the scattering of light differently.

How does the use of subsurface models influence the rendering equation? Recall that the basic model had the form

$$L(P, \omega_o) = \ldots + \int_{\omega_i} L(P, \omega_i) f_s(P, \omega_i, \omega_o) \ldots, \tag{27.44}$$

that is, the emitted radiance was basically the value of an integral over all incoming directions at P. But with subsurface scattering, light arriving at some other point Q might eventually be emitted at P, and the form becomes

$$L(P, \omega_o) = \ldots + \int_Q \int_{\omega_i} L(P, \omega_i) f_{ss}(P, Q, \omega_i, \omega_o) \ldots. \tag{27.45}$$

Computing subsurface scattering adds another integral. In practice, for most materials the new integral (over all points Q of the material) can be replaced by one over a bounded area (like a disk) around P: Light entering at your fingertip is unlikely to exit at your nose in any substantial amount. If the material is homogeneous and anisotropic, and the arriving radiance is nearly constant over the surface, the scattered radiance can be precomputed and stored in a lookup table, and the new computation is not much worse than that for the simpler rendering equation.

What are the practical effects of modeling subsurface scattering? First, it's possible for light to diffuse across shadow boundaries so that a "hard shadow" on skin, for instance, ends up slightly softened. Second, it allows color bleeding within objects: A cup of tea with milk has its color affected, near the edge of the tea's surface, by the color of the cup itself.

How does one model the subsurface scattering? In much the same way as we model surface scattering: either with acquired data, or by phenomenological or physical models. One can acquire data by modifying a gonioreflectometer to allow the sensor to be adjusted so that it measures light received from some location Q that's not the center, P, of the sample stage—either by translating the entire sensor assembly in the plane of the sample stage, or by making a small, two-axis rotation of the sensor about its center. One could equally well translate or rotate the illumination source, of course, and/or rotate or translate the sample stage. Alternatively, one can illuminate the sample stage and then replace the usual sensor with a camera. This allows the measurement of light from many material points and directions at once. This approach is described by Jensen et al. [JMLH01] in some detail.

As for physical modeling of the subsurface scattering, it involves physics and mathematics well beyond the scope of this book. Jensen et al. describe some feasible approximations, and give a general overview and pointers to the physics literature.

27.14 Software Interface to Material Models

For our applications of material modeling to rendering, we'll restrict our discussion to a particular class of models: those composed of a finite part and an impulse part, which we'll now characterize.

The "impulse" part of scattering describes phenomena like transmission through boundaries between media, and mirror reflection: Radiance arriving from one single direction generates radiance leaving in a single other direction (or, in the case of simultaneous transmission and reflection, two other directions). These are the phenomena for which the measurement of the BSDF is impossible, as we described in Section 26.10.2. "Impulsive" scattering from ω_i to ω_o is described by a single factor by which incoming radiance is multiplied to produce outgoing radiance. We'll assume that for a given direction ω_i toward a light source, there are finitely many (usually two!) directions ω_o for which (ω_i, ω_o) is an impulse-scattering pair. For each such pair, we'll have a constant m, the **magnitude** of the impulse, which is the factor by which incoming radiance is multiplied to get outgoing radiance. Similarly, we'll assume that if we know ω_o, there are only finitely many directions ω_i such that (ω_i, ω_o) is an impulse-scattering pair. To simplify things, for the remainder of this section we'll say that there are exactly *two* impulse directions ι_1 and ι_2, with magnitudes m_1 and m_2.

When we want to ray-trace, it's essential to recover these impulse directions and the associated material properties; we'll need

```
ImpulseArray getImpulsesIn(surfel, ωo)
```

where `surfel` is a `SurfaceElement` data structure that stores the geometric normal at a point of some surface, the location of that point, the index of refraction of the material on the side pointed to by the normal and on the other side as well, and the shading normal (a vector used in shading computations, often interpolated from geometric normals at nearby points). The array of returned "impulses" contains a list of pairs (ω, ω_o, m) where all the ω_o values equal the input argument, and m is the magnitude of the impulse. It's also useful to have the dual form

```
ImpulseArray getImpulsesOut(surfel, ωi)
```

which returns an array of (ω_i, ω) pairs, together with their impulse magnitudes.

In fancier versions of ray tracing that can handle area lights and glossy surfaces, it's also important to be able to evaluate the finite part of the BSDF for any two inputs, that is, we need

```
float getBSDFFinite(ωi, ωo)
```

It's also useful to be able to "sample from the BSDF," that is, to ask, given an output direction ω_o, for an input direction ω_i where the probability density of selecting ω_i is proportional to $f_s(\omega_i, \omega_o)$. For a Lambertian surface, such a procedure would return a direction ω_i in the upper hemisphere uniformly at random.

Sampling from the BSDF doesn't make literal sense if the BSDF contains impulses, for in that case, certain BSDF values are infinite. Suppose, for instance, that we have a material and a direction-to-light ω_i in which 40% of the light is mirror-reflected in a direction ι_1, while 60% is Lambertian-scattered. In this case,

$f_s(\omega_i, \iota_1) = \infty$, making sampling "proportional to" the BSDF impossible. But what we'd like, in this case, is for the sampling procedure to return ι_1 40% of the time, and to return a vector ω_i uniformly at random on the hemisphere the remaining 60% of the time. To further extend the example, if the magnitude of the impulse remained at 0.4, but the material absorbed 30% of the light hitting it and scattered the remaining 20% in a Lambertian fashion, we'd expect the procedure to return ι_1 40% of the time, NONE 30% of the time, and a uniformly distributed random vector ω_i the remaining 20% of the time. For this, we need a procedure like

```
Vector3 getSampleIn(ω_o)
```

although such a procedure may, in the case of highly peaked BSDFs, prove to be very slow unless the material model has been designed in advance to make such sampling efficient.

Sometimes it suffices to get a sample where the probability density of a particular direction ω_i isn't exactly proportional to the BSDF $f_s(\omega_i, \omega_o)$, but whose probability distribution p is somewhat similarly shaped to the BSDF; in this case, we need to know not only the sample direction, but also a "factor" given by

$$\frac{f_s^0(\omega_i, \omega_o)}{p(\omega_i)}, \tag{27.46}$$

although such an adjustment isn't needed for the impulse terms, because they can be sampled from exactly. In this case, we need a procedure with a signature like

```
Vector3 getWeakSampleIn(ω_o, float &factor)
```

in which the adjustment factor is set when the sampled vector is returned.

Corresponding procedures for sampling *outgoing* directions in proportion to the BSDF, or for sampling either incoming or outgoing directions in proportion to a cosine-weighted BSDF, are also useful. Indeed, the cosine-weighted versions are the ones we'll primarily use in writing a path tracer and photon mapper in Chapter 32.

27.15 Discussion and Further Reading

Correctly modeling scattering is central to making renderings look realistic: For directly illuminated surfaces, our eyes essentially observe the BSDF, so making it right is important. Pharr and Humphreys [PH10] discuss the modeling of BSDFs, and a software interface to them, in extensive detail.

There's a huge literature on scattering models, and it's worth reading at least one or two of the early papers—perhaps the Torrance-Sparrow or Cook-Torrance or Blinn-Phong papers—to get an idea of all the complexities.

Lawrence [Law06] addresses the very practical question of how to make computational models for scattering that are (a) expressive enough to match measured data and (b) simple enough to admit relatively easy sampling strategies.

We began this chapter by discussing how objects transform light fields, and promptly shifted to talking about surfaces made of materials; this factorization is great for reducing the complexity of light transport (e.g., it lets us use the

reflectance equation). But further factorization in the form of texture mapping, with parameters that can range from color to, say, the roughness parameter in some scattering model, allows further simplification. **Appearance modeling** is the craft of making compact representations for lots of materials. Given the messiness of scattering described in this chapter's introduction, it should be no surprise that no general theory of appearance modeling has yet emerged, despite some substantial successes [GTR+06] [DRS08].

We briefly discussed volumetric scattering, with an implicit assumption that the distribution of particles in the scattering medium was uniform random. In cases where the distribution has some structure (e.g., the rings of Saturn), more sophisticated methods are called for. These were pioneered by Blinn [Bli82b], and advanced by Kajiya and von Herzen [KVH84], Miller [Mil88], and Kajiya and Kay [KK89], who introduced the notion of **texels**—three-dimensional arrays of parameters approximating the visual properties of microsurfaces like hair or fur—as a way to represent scattering of light from structured volumes of scatterers (see Figure 27.20). The complexities of that work demonstrate once again the point we made at the start of this chapter: Scattering tends to be messy and complex.

Figure 27.20: A teddy bear rendered by Kajiya and Kay's texel-rendering algorithm. (Courtesy of Jim Kajiya. ©1989 ACM, Inc. Reprinted by permission.)

Is scattering *too* complex? If, in the course of rendering, we need to compute the light leaving some object but *not* going directly to the eye, it's possible in many cases to use a simplified proxy for scattering: We can't really tell whether light has been scattered from a furry teddy bear, or from a brown paper sack of about the same shape. There are exceptions, of course. Light scattered from a crystal chandelier produces highlights all around a room; replacing the crystals with diffusely scattering reflectors would not be the same at all. Even so, *much* of the effect of scattering—the complex appearance of the teddy bear, for instance—disappears after one bounce. It would be nice to avoid all the extra work in these cases.

In our discussion of the Torrance-Sparrow and Cook-Torrance models, scattering involves light inter-reflecting among multiple surfaces, resulting in shadowed and masked parts. This is exactly the same behavior we'll see in studying global illumination algorithms, in which the geometry of a scene causes multiply reflected light to reach some places and not others. For real-time "solutions" (i.e., some games as of 2013), it turns out that we can approximate the effect of these complex global-illumination algorithms and replace them with the idea of **ambient occlusion** [Lan02], in which we make things darker when their surroundings are locally more convex, by setting an ambient term that's proportional to how much of the far field you can see locally. This creates higher-frequency intensity gradients than you get with a $1/r^2$ falloff in light intensity, makes corners dark, and highlights concavities and convexities to give the viewer a clue about material smoothness at a relatively large scale, since shadows are an important proximity cue.

Finally, it's worth standing back and looking at the microfacet models in a larger context. When we want to render a scene faithfully, we have to take into account how light from the luminaires scatters from each surface onto each other surface, and the resultant complex distribution of light energy reaching the eye or camera. The interconnectedness of all objects in the scene (or at least all mutually visible objects) leads to algorithmic complexity. Now look at microfacet models: *They're doing the same thing!* Light arriving at the surface reaches only part of one microfacet because another shadows it, and light scattered from that micro-

facet hits yet another microfacet, etc. Because we're assuming that the surfaces on which these microfacets reside are globally flat (at least relative to the scale of individual microfacets), we can assume that each microfacet interacts with only a few neighbors, thus reducing the complexity of the rendering problem. When we get to limiting cases like light arriving at a glancing angle, when many facets might shadow a single one, this simplifying assumption can break down.

27.16 Exercises

Exercise 27.1: In Gouraud shading, we have color values at the three vertices of a triangle, and we interpolate those values across the interior of the triangle. Since this is typically done on a raster screen, one approach is to work from top to bottom, linearly interpolating values along each edge, and then within a single row of pixels linearly interpolating between the two ends. In a typical triangle, there will be a top vertex, a bottom vertex, and a middle vertex. As we pass the middle vertex, we'll need to start traversing a different edge. Alternatively, we could do interpolation from the top to the middle row, and from the bottom to the middle row.

(a) Show that if we compute the intersection of an edge with a row center exactly (rather than rounding to the nearest pixel center), the result is exactly the barycentric interpolation of the vertex values.

(b) Show that as we move from one row to the next, working down from the top vertex to the middle vertex, the starting value for each pixel row differs from the starting value for the previous row by the same amount.

(c) Use the idea of part (b) to develop a low-operation-count implementation of Gouraud shading in the 2D tested, using "pixels" that are each small, colored rectangles to visualize your results.

(d) Suppose we were to apply the same idea to shade a convex quadrilateral: We work from top to bottom, computing interpolated values along the two edge points in each row, and then linearly interpolate along the row. If we rotate the quadrilateral (keeping the assigned color values at each vertex), does the interpolated shading appear to rotate as well?

Exercise 27.2: It's very common to photograph either the moon or a lighthouse being reflected in fairly calm (but not completely flat) water. The usual depiction shows the bright reflection appearing as a wedge that grows wider as it approaches the viewer, with the wedge point either near the lighthouse or (in the case of the moon) near the horizon. Find such a photograph, describe the notable features of the reflection (such as the shape of the wedge), and explain them in terms of the physical models you've seen in this chapter.

Exercise 27.3: A bookshelf holds three books, one with a white binding and two with black bindings. The shelf itself is made of polished wood. Looking down at the shelf, you can see the reflection of the spines of the books. Near the bottom of the spines, the division between the white and black reflections appears quite sharp. But if you look at the reflections of a region near the top of the spines, the division is quite blurry. Explain why. Is it a Fresnel effect, because of the difference in viewing angle? Why or why not? How could you test this idea? Hint: Actually set up the experiment!

Exercise 27.4: In the Phong model, as typically expressed, there's a diffuse color, expressed as an RGB triple, and a diffuse reflectance.

(a) Make a conjecture about why these are not combined into a single RGB triple, since they are, after all, multiplied together in the model.
(b) The diffuse and specular colors in the Cook-Torrance model are specified by RGB triples (or more generally, by spectral distributions). Why do you suppose the specular exponent does *not* get expressed as an RGB triple?
(c) Suppose that the entire model was divided up by wavelength so that the specular exponent *could* vary from color to color. Suppose the red exponent was 3, while the blue and green exponents were 5, and that both the diffuse and specular colors were pure white. What would be the appearance of the specular highlight from a white point light?

Exercise 27.5: Plot F_R, the Fresnel reflectance, as a function of θ_i, for an interface where $r = n_1/n_2$ is 1, 1.5, 2, and 3. To do so you'll need to compute θ_t using Snell's law.

Exercise 27.6: Compare the Schlick approximation of the reflectance with the Fresnel expression for the reflectance for aluminum and magnesium oxide at 500 nm. Magnesium oxide is nonmetallic, so the Schlick approximation should not necessarily be expected to work. Use 1.0 as the index of refraction of air, 1.44 for aluminum, and 1.74 for magnesium oxide.

Exercise 27.7: The Fresnel term used in graphics assumes that the arriving light is unpolarized; on the other hand, the different reflection constants for parallel and perpendicular polarization mean that the light *leaving* the surface *is* actually polarized. Yet we generally assume, when it reaches the next surface, that it's unpolarized. Try to imagine a physical situation in which this inconsistency would manifest itself in a visible artifact.

Chapter 28

Color

> Strictly speaking, the rays are not colored.
>
> *Optics,* Isaac Newton

28.1 Introduction

Most people are able to sense *color*—it's the sensation that arises when our eyes are presented with different spectral mixes of light. Light with a wavelength of near 400 nanometers makes most people experience the sensation "blue," while light with a wavelength near 700 nm causes the sensation "red." We describe color as a *sensation* because that's what it is. It's tempting to say that the light arriving at our eyes is colored, and we're just detecting that property, but this misses many essential characteristics of the perceptual process; perhaps the most significant one is this: Two very different mixes of light of different frequencies can generate the same perception of color (i.e., we may say "Those two lights are the same color green"). Thus, our notion of color, which we use to distinguish among lights of different wavelengths, is insufficient to distinguish among *mixtures* of lights at different wavelengths. It's therefore worth distinguishing between the physical phenomenon ("This light consists of a certain mixture of wavelengths") and the perceptual one ("This light looks lime green to me"). Furthermore, our observation of the same spectral mix may cause different perceptions at different times or different intensities.

As you read this chapter, you should keep the following high-level facts in mind.

- Color is a perceptual phenomenon; spectral distributions are physical phenomena.
- Everything you learned about red, green, and blue in elementary school was a simplification.

- The eye is approximately logarithmic: Each time you double the light energy (without altering the spectral distribution) arriving at your eye, the brightness that you perceive will increase by the same amount (i.e., the brightness difference between one unit of energy and four units of energy is the same as the brightness difference between 16 units and 64 units).

Most of what a majority of people "know" about color is false, or at the very least, it is true only under very restrictive conditions of which they are unaware. Try to read this chapter with an open mind, forgetting what you've learned about color in the past.

28.1.1 Implications of Color

Before we discuss the physical and perceptual phenomena involved in color, let's consider some implications of color: Because objects have different colors, and because you can tell the difference, you can use color in a user interface to encode certain things. For instance, you might choose to make all the icons in a text editor having to do with high-lighting be based on a yellow background, reflecting the idea that many highlighter markers are yellow. Similarly, you might choose to make all the high-priority items (or all the items with significant consequences, like "Close this document without saving changes") be drawn in red, to attract the user's attention.

But a significant number of people are colorblind (or, more accurately, color-perception deficient)—they perceive different wavelength mixes in a different way from the rest of us, and two lights that appear red and green to most people appear to be the same color to a red-green colorblind person. About 8% to 10% of men are red-green colorblind; there's also yellow-blue colorblindness (quite rare), and even total colorblindness, but this is very rare. Colorblindness is very rare (less than 1%) in women.

From a computer graphics point of view, the critical consequence of color-blindness comes in interface design: If you rely solely on color-coding to indicate things, about 5% of your users will miss the idea you're trying to indicate.

The effects of individual colors are important, but even more significant is the challenge of selecting groups of colors that "work well together." Such selections are in the domain of art and design rather than science. As you design a color palette for a user interface, consider the following.

- Someone else may have already developed a good set of colors; try starting from interfaces that you like and working with *their* colors.
- Use a paint program to see how each of your colors looks when placed atop or near each of your other colors, or in groups of three.
- Consider how your colors will look on various devices; certain colors that look good on an LCD screen may look bad when printed. If this matters in your application, you'll want to design with this in mind from the start.

28.2 Spectral Distribution of Light

We begin our discussion of color with the physical aspects. As we described in Chapter 26, light is a form of electromagnetic radiation; visible light has wavelengths between 400 and 700 nanometers. An ordinary fluorescent lamp

(see Figure 28.1) produces light at many wavelengths; the combination of these makes us perceive "white." By contrast, a laser pointer uses a light-emitting diode (LED) to create light of a single wavelength, usually around 650 nm, which we perceive as "red."

The **spectral power distribution** or **SPD** is a function describing the power in a light beam at each wavelength. It can take on virtually any shape (as long as it's everywhere non-negative). Filters are available that allow only certain wavelengths, or wavelength regions, to pass through the filters; clever combinations of these allow one to create almost any possible spectral power distribution. We can add two such functions to get a third, or multiply such a function by a positive constant to get a new one. Thus, the set of all spectral power distribution functions forms a **convex cone** in the vector space of all functions on the interval $[400\,\text{nm}, 700\,\text{nm}]$. The possibility of creating almost any function means that this cone is infinite-dimensional; in particular, the spectral power distributions

$$P_s(\lambda) = \begin{cases} 1 & \text{if } s \le \lambda \le s+1 \\ 0 & \text{otherwise,} \end{cases} \tag{28.1}$$

where s ranges over integers between 400 and 699, are all linearly independent, so the space is at least 299-dimensional. By making the "spikes" in the function narrower and the spacing closer, it's easy to see that the number of linearly independent functions is arbitrarily large.

By contrast, as we'll see in later sections, the set of color **percepts,** or color sensations, is three-dimensional; to the degree that the mapping from spectral power distributions to percepts is linear, it must be many-to-one. Indeed, for any given percept, there must be an infinite-dimensional family of SPDs that give rise to that percept.

Certain SPDs are both important and easy to understand: These are the **monospectral** distributions, in which nearly all the power is at or very near to a single wavelength (see Figure 28.2).

One reason that these are interesting is that all other SPDs can be written as (infinite) linear combinations of them, so they play the role of a basis for the set of SPDs.

A pure monospectral light cannot (in our model of light) carry any energy, because the energy is described in part by an integral over wavelength. So when we speak of "monospectral" lights, you should think of a light whose spectrum is entirely in the interval from 650 nm to 650.01 nm, for example.

Describing an SPD requires either tabulating its (infinitely many) values, or somehow presenting summary information. In practice, real SPDs are tabulated at finitely many values using a spectroradiometer, but even these tabulated values may need to be summarized. In **colorimetry,** the terms **dominant wavelength, excitation purity,** and **luminance** are used to present such summaries; these vary in utility depending on the shape of the SPD. For the highly contrived SPD of Figure 28.3, the *dominant wavelength* is 500 nm. The *excitation purity* is defined in terms of the relative amounts of the dominant wavelength and the broad-spectrum light: If e_1 is zero and e_2 is large, then the excitation purity is 100%; if $e_1 = e_2$, the excitation purity is zero. So excitation purity measures the degree to which the light is monospectral. (For more complex spectra, the precise definition of the "dominant wavelength" is subtler; it's not always the one with the highest value, which might be ill-defined if multiple peaks had the same height. These subtleties need not concern us.)

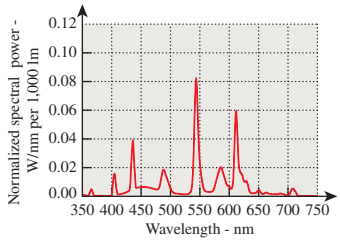


Figure 28.1: The spectral power distribution of a fluorescent lamp. The power emitted at each wavelength varies fairly smoothly across the spectrum, with a few high peaks. Figure provided courtesy of Osram Sylvania, Inc.

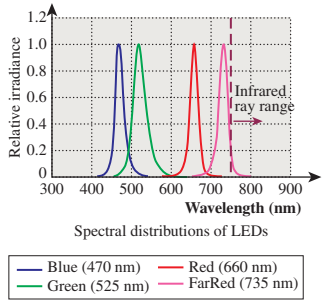


Figure 28.2: The spectral power distributions of several LEDs. The light is concentrated at or near a single wavelength for each kind of LED; an ideal monospectral source would have all energy at a single wavelength.

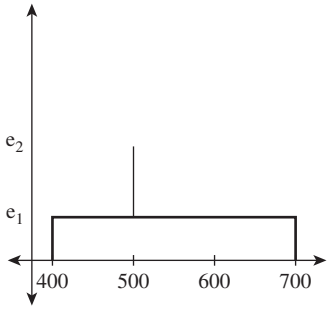


Figure 28.3: A contrived spectral power distribution with 500 nm as its dominant wavelength.

One last note about spectral power distributions: Ordinary incandescent lights (especially those with a clear glass bulb) have spectral power distributions that are quite similar to the blackbody radiation described in Chapter 26, because they produce light by heating a piece of metal (tungsten, typically) to a very high temperature—such as 2500°C—by pushing electric current through it; the resultant emission begins to approximate the blackbody curve, even though the tungsten itself is not matte black. (The sun, by contrast, has a surface temperature of around 6000°C.) One important characteristic of this radiation is that the SPD is quite smooth, rather than being very "spiky." This makes simple summary descriptors like "dominant wavelength" and "excitation purity" work quite well for such smooth SPDs.

28.3 The Phenomenon of Color Perception and the Physiology of the Eye

People with unimpaired vision perceive light; they describe their sensations of it in various terms like "brightness" and "hue" and with a great many individual words ("saffron," "teal," "indigo," "aqua," . . .) that capture individual sensations of color.

Our perception of color is also influenced by a gestalt view of the world: We use different words to describe the color of things that emit light and to describe those that reflect light. People will describe an object as "brown," but they will almost never speak of a "brown light."

This same gestalt view allows us to understand the "colors of objects." One might say that a yellow book, in a completely dark closet, is black, but people are more inclined to say that it's yellow but not lit right now. Certainly in a dimly lit room, the light leaving the yellow book's surface is different from that leaving the surface in a well-lit room, and yet we describe the book as "yellow" in both cases. Our ability to detect something about color in a way that's partly independent of illumination is termed **color constancy.**

Of course, one can imagine an experiment in which one looks through a peephole and sees something behind it. The something might be a glowing yellow bulb, or it might be a yellow piece of paper reflecting the light from an incandescent bulb. When the object is seen from a distance, and without other objects nearby for comparison, one cannot tell the difference between the two. So the distinction between "emitters" and "reflectors" is not one that's captured by the physics of the light entering the eye, but by the overall context in which the light is seen.

By the way, to experiment with color, it turns out that "color matching" is different from "color naming": Saying the name of a color is more complex than matching a color with another during an experiment.

It's commonplace to say that "intensity" is independent of "hue": One can have a bright blue light or a dim blue light, and the same goes for red and yellow and orange and green. In the same way, the degree of "saturation" of a color—Is it *really* red, or is it pinkish, or a grayish-red?—appears independent of both intensity and hue. But it's difficult to think of a fourth property of color that's independent of these three. This suggests that perhaps color is defined by three independent characteristics, which we'll later see is true. Just *which* three

characteristics is a matter of choice (just as choosing the coordinate axes to use on a plane is a matter of choice; any pair of perpendicular lines can work!). So some people choose to describe colors in terms of hue, saturation, and "value," while others prefer to describe mixes of red, green, and blue. We'll say much more about these in Section 28.13.

Careful physiological experiments have revealed much of the structure of the eye; Deering [Dee05] presents a good summary of the results of this work in the context of understanding what the retina can detect, which provides a guide to what is worth rendering in the first place. The key thing, from the point of view of understanding color, is the presence of two kinds of receptors: **rods** and **cones.** Rods are sensitive to visible light of all wavelengths, while the three types of cones are sensitive to different wavelengths of light: The first has its peak response at 580 nm, the second at 545 nm, and the third at 440 nm (see Figure 28.4). Detailed observations of the response curves for the receptors (including rods) are described by Bowmaker and Dartnall [BD80]. These are often described as "red," "green," and "blue" receptors, even though the red and green peaks occur at wavelengths commonly described as yellow, with the red peak being an orangy yellow and the green peak being a greener yellow. (To be more precise, a monospectral light of 580 nm wavelength causes, in most viewers, the percept "orangy yellow.") A better set of names is "long wavelength," "medium wavelength," and "short wavelength" receptors, and the names L, M, and S are often used for these. We'll generally use "red," "green," and "blue," however, to avoid the need to convert from wavelength to color.

One can read this graph by saying, for instance, that a certain amount e of light at 560 nm will cause a response in a red receptor, but that one would need twice as much light at wavelength 530 nm to generate the same response in that red receptor. (Of course, these lights provoke very different responses in the green and blue receptors, too.) Furthermore, the effects of different lights on the red receptor are additive: Sending in both e light at 560 nm and $2e$ light at 530 nm will generate the same red-receptor response as sending $2e$ at 560 nm. If we use $f(\lambda)$ to indicate the red receptor's response at wavelength λ and use $I(\lambda)$ to indicate the incoming light's intensity at wavelength λ, then the total response from the receptor will be

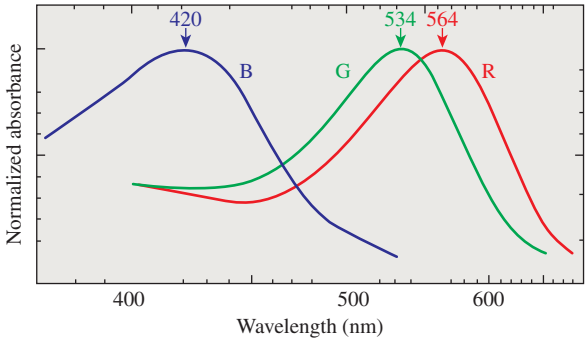


Figure 28.4: The approximate spectral response functions of the three types of cones in the human retina; the labels R, G, and B are misleading, because the peaks of the R and G curves both correspond to monospectral lights that most people describe as in the "yellow" range.

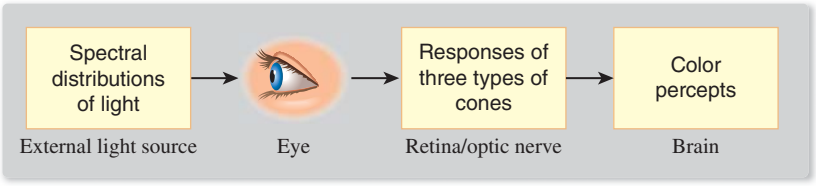


Figure 28.5: Light, described by its spectral power distribution, enters the eye; the three types of cones each respond and their individual responses are conducted by the optic nerves to the brain, resulting in a perception of color. These correspond to three distinct areas of study: physics, physiology, and perceptual psychology.

$$\int_{400\,\text{nm}}^{700\,\text{nm}} I(\lambda) f(\lambda)\, d\lambda. \tag{28.2}$$

In short, the total response is a linear function of the incoming light I, with the linear operation being "integrate against the response curve."

With this in mind, we can consider a system diagram (see Figure 28.5) tracing how a physical phenomenon (the spectral power distribution of light) becomes a perceptual phenomenon (the experience of color). Notice that this diagram is slightly simplified, in that it treats the incoming light without considering how the pattern of light is organized (i.e., what the person is actually seeing). This omission makes it impossible for this model to account for phenomena like spatial comparison of colors or color constancy, but the simplification—we can imagine that all the light arriving comes from a single, large, glowing surface surrounding the viewer—makes it easy to discuss the basic phenomena of color.

28.4 The Perception of Color

Given the three types of cones, it's not surprising that color perception appears to be three-dimensional. We begin with the examination of the aspect that's least related to color, which is **brightness**—the impression we have of how bright a light is, independent of its hue. By the way, the brightness we are referring to is not a quantity that has physical units; it's a generic and informal term used to characterize the human sensation of the amount of light arriving at the eye from somewhere (a lamp, a reflecting surface, etc.).

28.4.1 The Perception of Brightness

To determine relative brightness of light at different wavelengths, imagine an experiment in which you are shown two lights: a 555 nm reference monospectral light source, and a second monospectral light source whose wavelength λ will be varied over the range 400 nm to 700 nm. We fix a particular wavelength λ, and you are given a knob with which you can control a multiplier for the reference light source; you adjust it until it has the same brightness as the one at wavelength λ. We record the setting $g(\lambda)$ and reset λ to a new value and repeat. When we are done, we have a tabulation of how effective light at frequency λ is at seeming bright, compared to light at the reference wavelength 555 nm. For each value of λ, the number $g(\lambda)$ tells how much less effective light at wavelength λ is in

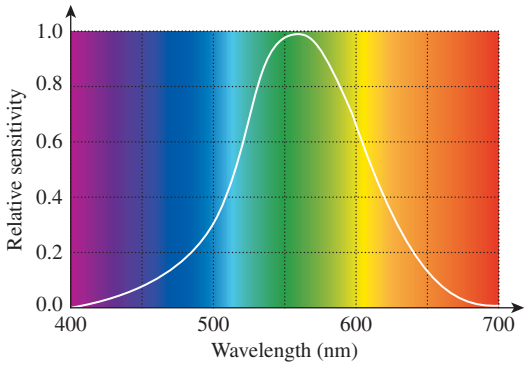


Figure 28.6: The luminous efficiency at each wavelength tells how much less bright light of that wavelength appears than light at the standard wavelength of 555 nm.

provoking a response than light at wavelength 555 nm. Scaling so that the largest value of $g(\lambda)$ is 100%, we can plot the resultant function $\lambda \mapsto e(\lambda)$ (see Figure 28.6); the resultant graph shows the **luminous efficiency function** for the human eye. ("Efficiency" here refers to how efficient energy at a particular wavelength is in provoking the sensation of brightness.)

The luminous efficiency graph actually varies from person to person, and varies based on the person's age as well; in view of this, a standard luminous efficiency curve was derived by averaging many observations.

This standardized tabulation can be used to define the **luminance** of a light source: We multiply the intensity at each of the tabulated wavelengths by the luminous efficiency value for that wavelength, and compute the sum, thus approximating the value

$$\int I(\lambda)e(\lambda)d\lambda, \tag{28.3}$$

where $I(\lambda)$ is the spectral intensity at wavelength λ. The resultant value has units of **candelas,** which is the SI unit for the measurement of **luminous intensity.** One candela is the luminous intensity, in a given direction, of a source that emits monochromatic radiation at a frequency of 540×10^{12} Hz (i.e., 555 nm wavelength),[1] and whose radiant intensity in that direction is $1/683$ watt per steradian. The international standards committee chose the peculiar numbers in this definition to make it closely match earlier measures that were based on the light from a single standard candle, or the light from a certain near-blackbody source (a square centimeter of melting platinum). Naturally, light at other wavelengths, of equal radiant intensity, produces fewer candelas of visible light than does light at 555 nm.

To give a sense of common illumination in terms of candelas, my LCD screen emits about 250 candelas per square meter (one candela per square meter is called a **nit;** it's the photometric term corresponding to radiance in radiometry), while the light from the screen at a movie theatre is about 40 candelas per square meter.

1. The definition is given in terms of frequency rather than wavelength because the speed of light varies in different media; in graphics, where we work primarily with light in air, this consideration is irrelevant.

A studio broadcast monitor has a reference brightness of 100 candelas per square meter.

It's tempting to say that since the human eye's sensitivity to light is captured by the candela, we could (if we wanted to do just grayscale graphics) represent all light in terms of the candela. As mentioned in Chapter 1 and Chapter 26, this would be a grave error. In doing so, we'd need to assign a reflectivity to each surface; assuming diffuse surfaces, this would be a single number indicating what fraction of incoming light becomes outgoing light. Suppose we have a surface whose reflectivity is 50%. Then incoming light of a particular luminous intensity would become outgoing light of half that intensity. The problem is that real surfaces, with real light, may reflect different wavelengths differently. A surface might, for instance, reflect the lower half of the spectrum perfectly, but absorb all light in the upper half. If it's illuminated by two sources, one that's in the lower half and one that's in the upper half, with equal luminous intensity, the reflected light in the first case will have the same luminous intensity, while in the second case it will have none at all. In other words, there are cases where this "summary number" captures information about human perception, but masks information about the underlying physics that brought the light to the eye. One could argue, therefore, that luminous intensity of light should only be examined for light that arrives at some person's eye.

Counter to this position is the fact that much of the light we encounter every day (like that from incandescent lamps) is a mixture of many wavelengths, and most surfaces reflect some light of every wavelength, so in practice we can use a summary number like luminous intensity, and a summary reflectivity, and the reflected light's luminous intensity will turn out to be the incoming intensity multiplied by the reflectivity. This summary-number approach only causes problems in cases where the spectral distribution of energy (or of reflectivity) is peculiar. But with the advent of LED-based interior lighting, such peculiar distributions are becoming increasingly commonplace; many of today's "white LED flashlights" are actually based on multiple LEDs of different frequencies, and have highly peaked spectral distributions, for instance. This discussion is another example of the Noncommutativity principle.

We've said that because photometric quantities represent weighted averages, and the weighted-averaging process does not commute with various other operations (like multiplication), these photometric quantities will be of little use to us except when applied to the light arriving at the human eye. To clarify the statement about weighted averages, consider the following example. We take two lists of numbers,

$$L = (1, 3, 1, 5, 6) \text{ and} \tag{28.4}$$

$$R = (.33, .33, .33, 0, 0), \tag{28.5}$$

and consider the weighted sum of each under the weights

$$w = (0.2, 0.2, 0.3, 0.3, 0.0). \tag{28.6}$$

The results are 2.6 and .233, respectively.

Now consider the term-by-term product of L and R; it is

$$(0.33, 1, 0.33, 0, 0), \tag{28.7}$$

and its weighted sum, using w, is 0.33. Notice that 0.33 is *not* $2.6 \times .233 = .6058$; in other words, computing the weighted sums and then multiplying is different from multiplying and then computing a weighted sum. Now imagine that L represents the light energy at five chosen frequencies, and R represents the reflectivity of a surface at those five frequencies. Then the term-by-term product represents the frequency distribution of the reflected light. But if we computed the weighted sum of each thing (i.e., the thing corresponding to the photometric measurements), then the product of the aggregate incoming light and the aggregate reflectivity is not the aggregate outgoing light. (There's an argument to be made that we should not multiply the reflectances by the weights, but we should weight them evenly; even with this approach, commutativity fails.)

You may encounter other photometric terms; each of them can be thought of as a radiometric quantity, recorded per wavelength and then integrated against the luminous efficiency curve. Table 28.1 shows this correspondence.

Table 28.1: Comparison of radiometric and photometric terms.

Concept	Radiometric Units	Photometric Units	Photometric Name (abbr.)
Spectral radiance	$\mathrm{W\,m^{-2}\,sr^{-1}\,nm^{-1}}$	$\mathrm{lm\,m^{-2}\,sr^{-1}}$	Nit
...integrated over an area	$\mathrm{W\,sr^{-1}\,nm^{-1}}$	$\mathrm{lm\,sr^{-1}}$	Candela (cd)
...integrated over a solid angle	$\mathrm{W\,m^{-2}\,nm^{-1}}$	$\mathrm{lm/m^2}$	Lux (lx)
...integrated over an area and a solid angle	$\mathrm{W\,nm^{-1}}$	lm	Lumen (lm)

Note that *radiance* is also an integrated form of spectral radiance, but it's simply an integral over wavelength, without the weighting factor provided by the luminous efficiency curve. Because of this, you cannot compute photometric quantities from nonspectral radiometric quantities. If someone asks, "I've got a source that's $18\,\mathrm{W\,m^{-2}\,sr^{-1}}$, how many nits is that?" there is no correct answer!

28.4.1.1 Scotopic and Photopic Vision

The rods (the other kind of receptor in the eye) are also sensitive to light, but in a different way than the cones. The cones are the dominant receptors in high-light situations (e.g., daytime), while the rods dominate in low-light situations (e.g., outdoors at night). The first of these is called **photopic** vision, and the second **scotopic** vision. The scotopic response curve is different from the photopic response curve (see Figure 28.7), having a peak at a lower wavelength and dropping to zero by about 650 nm. This means that the rods cannot detect the sort of light we perceive as "red." Because both kinds of receptors perform some adaptation to average light levels, this makes red a good color for instruments that will be used in low-light situations: The red light from the instruments does not affect the

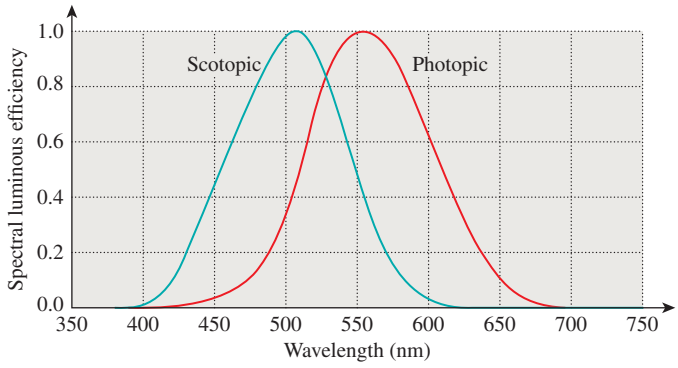


Figure 28.7: The luminous efficiency in photopic vision has a peak at 555 nm; the peak for scotopic vision is closer to 520 nm.

average-light-level adjustment for the rods, which are the primary receptors in use for seeing things in the dark.

28.4.1.2 Brightness

Our discussion so far has addressed the issue of how light of different wavelengths is perceived. There's a separate issue: how light of different intensities (but constant spectral distribution) is perceived. In other words, if we have a diffuser—a piece of frosted glass, for example—and it can be lit from behind by 100 identical lamps, and we turn on one, or ten, or all 100 so that the diffuser appears to be a variable light source, how will our eyes and brains characterize the change in brightness? Given the wide range of intensities that we encounter in daily life, it's hardly surprising that the response can be modeled as logarithmic: The change from one lamp to ten is perceived as being the same "brightness increase" as the change from ten lamps to 100. (Here we are using *brightness* in a purely perceptual sense, not as something physical to be measured, but as a description of a sensation.) That is to say, this model says that the perceptual strength associated to seeing a light of luminous intensity I is

$$S = k \log(I). \tag{28.8}$$

In support of the idea of a logarithmic model of our sensitivity to luminance, we can display two lights of the same luminous intensity and then adjust one until it becomes just noticeably different from the other. By doing this over and over at different starting sensitivities, we find that the **just noticeable difference** (or **JND**) is about 1% (i.e., $1.01I$ is noticeably different from I) for a wide range of intensities. In very dark and very bright environments, the number increases substantially, but for a range that includes the intensity ranges of virtually all of today's displays, it is about 1%. So if we adjusted the intensity repeatedly by 1%, we might expect to say that the brightness had increased by several "steps," and that k steps of increase would be achieved by multiplying by $(1.01)^k$. (This reasoning makes the assumption that each JND seems to the viewer to be of the same "size," however.) This implies that the response is proportional to the logarithm of the intensity.

An alternative model (Stevens' law) says that the response should be modeled by a power law:

$$S = cI^b, \tag{28.9}$$

where b is a number slightly less than 1. The shapes of the graphs of $\log$ and $y = x^b$ are somewhat similar—both concave down, both slowly growing—so it's no surprise that both can be used to fit the data decently. Each model has its detractors, but from our point of view, the important feature is that either one can be used to generate a good fit to the data, *particularly* when the range of brightnesses being considered is relatively small. In fact, as we'll discuss later, the eye adapts to the prevailing light in an environment, and intensities that differ from this by modest amounts can be compared to one another. But our sensation of lights that are very bright or very dim compared to the average is quite different ("too dark to see" or "too bright to look at").

The Commission Internationale de l'Éclairage (CIE), a group responsible for defining terms related to lighting and color, chose to use a modified version of Stevens' law to characterize perceptual responses to light, and it's this model that we'll use in further discussing the perception of brightness. To be explicit, the CIE defines **lightness** as

$$L^* = \begin{cases} 116(Y/Y_n)^{\frac{1}{3}} - 16 & \frac{Y}{Y_n} < 0.008856 \\ 903.3Y & \frac{Y}{Y_n} \geq 0.008856 \end{cases}, \tag{28.10}$$

where Y (called **luminance**) denotes a CIE-defined quantity that's proportional (for any fixed spectrum) to the energy of the light, and Y_n denotes the Y value for a particular light that you choose to be the "reference white."

You can see that L^* is defined by a $1/3$ power law that's been shifted downward a little (the -16 does this), and which has had a short linear segment added to deal with very low light values. In practice, this linear segment applies only to intensities that are a factor of more than 100 smaller than that of the reference white; in a typical computer graphics image, these are effectively black, so the linear segment is mostly irrelevant.

In practice, this logarithmic or power-law nature of things is somewhat confounded by "adaptation" of the rods and cones. The luminance we encounter in ordinary experience ranges over a factor of 10^9 between a moonless overcast night and a snowy region on a sunny day. Both rods and cones react to arriving light with chemical changes, which in turn generate an electrical change that is communicated to the brain. Plotting the output of the various sensors against the log of the luminance, we get a graph like the one shown in Figure 28.8; the rods react to varying luminance by changing their output . . . up to a point. After that point, any further increase in luminance doesn't affect the rods' output, and they are said to be **saturated.** The cones, on the other hand, begin to change their output substantially at about that point, so differences in brightness are detected by the photopic system. The placement of the cones' curve on the axes, though, is not fixed: Upon exposure to light of a certain level, like D on the chart, the cones, which were near the limit of their output, will gradually *adapt* and shift their response curve so that it's centered at D, thus responding to light changes at or near D. This ability to adapt is limited—at some point, all light begins to seem "very bright." The function of the "reference white" in the CIE definition is to characterize the interval of intensities over which we want to characterize lightness.

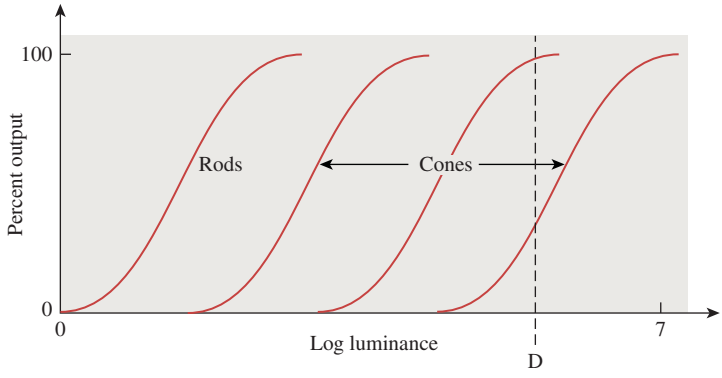


Figure 28.8: The percent output of rods and cones. The rods' output flattens out at modest luminance levels, and no change in output occurs even in very bright scenes; the cones' response also flattens out, but the absolute position of the curve may vary along the log luminance axis substantially as the cones adapt to the light present.

28.5 Color Description

Among lights of constant brightness, there is considerable variation in spectral power distribution: Those with greater power in the long wavelengths tend to have one appearance; those with greater power in the short wavelengths have another. We associate these appearances with the notions of "red" and "blue." Indeed, there's a whole vocabulary associated with describing color. Because it's natural to think of color as an intrinsic property of surfaces or lights, and only after a recognition of the mechanism of color perception is it clear that color is in fact a perceptual phenomenon, most discussions of color talk about the colors of objects, especially paints. We'll begin by introducing the terms used, and then consider their meanings in light of our system view. Such terms as "hue," "lightness," "brightness," "tints," "shades," "tones," and "grays" are all used to describe our perception of things. **Lightness** is used to describe surfaces, while **brightness** usually describes light sources. **Hue** is used to characterize the quality that we describe with words like "red," "blue," "purple," "aqua," and so on, that is, the quality that makes something appear to not be a blend of black and white. Blends of black and white are called **grays;** blends of white and pure colors are called **tints,** while blends of black and pure colors are called **shades.** Colors that are blends of black, white, and some pure color are called **tones** (see Figure 28.9). (Properly speaking, we should say "The percepts arising from various combinations of stimuli that produce the percepts 'black' and 'white' are called 'grays'," but such language rapidly becomes fairly cumbersome.)

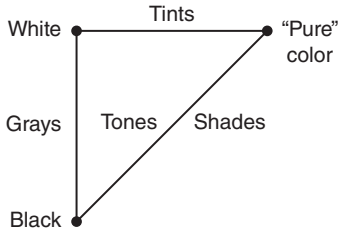


Figure 28.9: Tints, tones, and shades, as commonly used to describe colors.

What constitutes a "pure color," though? Among all lights of a given luminance (monospectral or otherwise), we can form combinations by blending 50% of one light with 50% of another, or blending with a 70:30 ratio, etc. Doing so takes lights whose colors we've experienced and produces new ones, whose colors may be new to us or may be ones we've experienced before. As we experiment with more and more spectral power distributions, we find that certain distributions have colors that are "at the edge," in the sense that they never appear as the color of any combination of other spectra. Such colors can be called "pure." Indeed, experiment shows that such a designation of pure spectra leads to labeling

precisely to the monospectral sources as "pure." Our understanding of the cones can tell us why.

The sensitivities of the three cones to various wavelengths of light mean that a monospectral light arriving at the eye generates a signal to the brain consisting of the outputs of the three kinds of sensors, which can be read off the response chart. In Figure 28.11 you can see how light of 440 nm generates lots of blue-cone (i.e., short-wavelength) response, less green (i.e., medium-wavelength), and even less red (i.e., long-wavelength). Similarly, at 570 nm, both red and green cones produce large responses, while the blue cones generate almost none. One can make a three-dimensional coordinate system labeled with S, M, and L, and plot the curve defined by such responses (see Figure 28.10).

Light that is a mix of these monospectral lights will (approximately, and within certain bounds) provoke a response that is a linear combination (with positive coefficients) of the responses to the monospectral lights, that is, the set of all responses will form a **generalized cone** in this space of possible cone responses (see Figure 28.12). The responses to monospectral lights are the points on the boundary of this cone, as predicted, in the sense that each of them cannot be produced as a combination of other responses. There's one exception: The start and end of the monospectral response curve are points representing pure red and pure violet. Combinations of these form a line; the collection of rays from the origin through this line is a planar region constituting a part of the cone's boundary. Points on this part of the boundary *are* representable as combinations of other response points; they are the "purples," and are not "pure" colors. (Note that the geometry of this response cone—the mostly convex shape of its cross section, in particular—is a consequence of the shapes of the response curves for the three types of cones in the eye. In the exercises in this chapter, you'll study what the

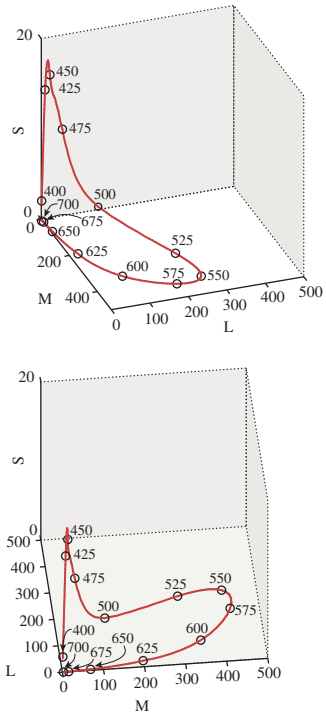


Figure 28.10: The response curve for monospectral visible light. Note that the short-wavelength-response axis has a different scale. We show the curve from two different views.

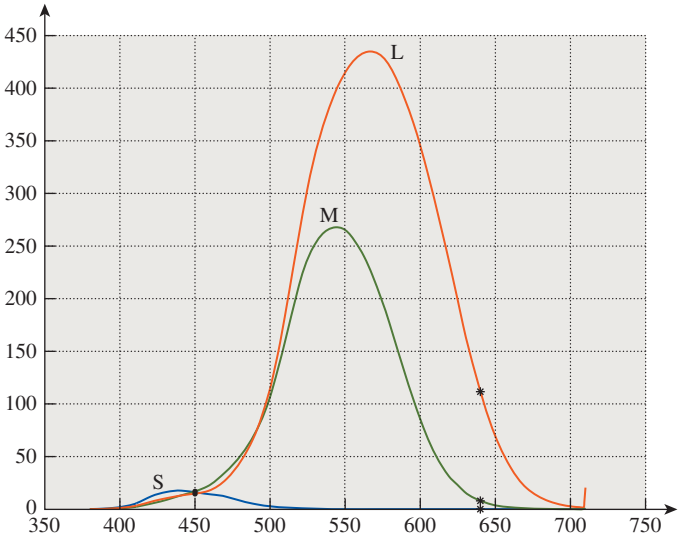


Figure 28.11: The response of the three cone types to a monospectral light can be read from the plot of their sensitivities. Light at 450 nm, for instance, generates about equal short- and medium-wavelength responses (shown in blue and green and labeled "S" and "M"), but a slightly smaller long-wavelength response (shown in red and labeled "L"). Light at 640 nm generates a large red response, a small green response, and almost no blue response.

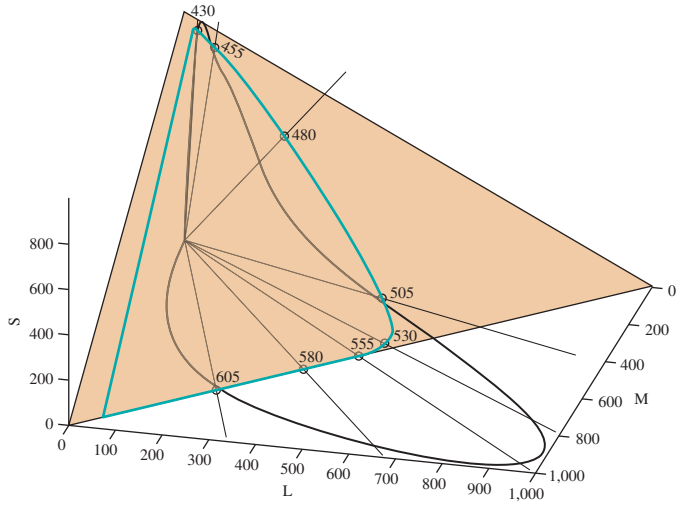


Figure 28.12: The set of all possible responses from combinations of monospectral lights (i.e., all possible spectral power distributions) forms a generalized cone in the space of response triples. The cone's intersection with the $S+M+L=1000$ plane (tan) is the area bounded by the aqua curve.

shape of this curve might be if the sensors' response curves were different, and where the monospectral curve would lie in those cases.)

28.6 Conventional Color Wisdom

Knowing how spectral information is converted to perceive color (at least in the absence of gestalt influences) allows us to understand something about the conventional wisdom surrounding color. We'll discuss a few common claims here.

28.6.1 Primary Colors

We often hear that "red, green, and blue are primary colors" (usually without a definition of "primary"), which we take to mean that they are colors that cannot be made from others, while all other colors *can* be made from them. Anyone who has tried to make orange from red, green, and blue paint knows this is false. But you *can* create a wide range of colors (or, to be pedantic, of paints which, when illuminated by sunlight or similar spectra, produce a wide range of color percepts) from red, green, and blue—far wider than you can produce from pink, yellow, and orange, for instance.

If we consider the aqua curve in Figure 28.12, but we adjust each monospectral light using the luminous efficiency curve so that they all have the same perceptual brightness, we get a curve in the plane of constant brightness that looks something like Figure 28.13 and on which we can identify points corresponding to the percepts "red," "green," and "blue." The responses associated to other spectral power distributions of the same brightness fill in this horseshoe shape, resulting in other percepts of "less saturated" colors, including white near the center.

The triangle generated by the colors red, green, and blue occupies much of this horseshoe shape, partially justifying calling them "primary," although this corresponds to the addition of lights (i.e., we can say that red, green, and blue are primary light colors).

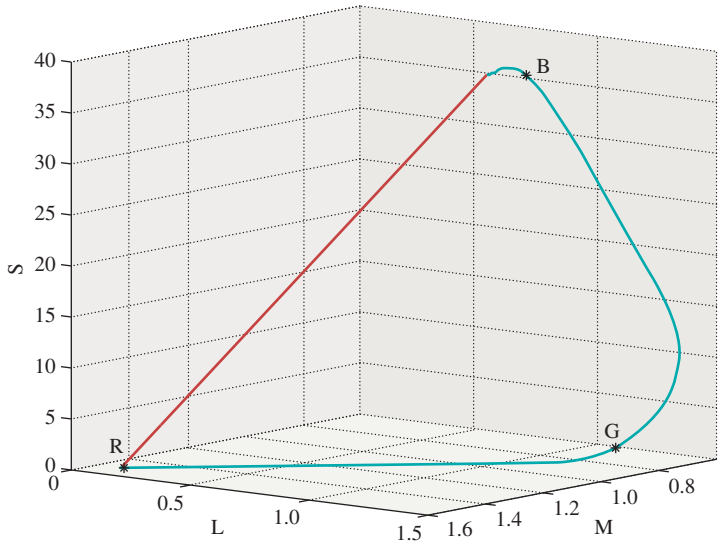


Figure 28.13: The set of responses associated to monospectral lights of equal brightnesses. These form a curve in a plane of constant brightness. Three points, corresponding to percepts of "red," "green," and "blue," are marked on the curve.

For paints, there's something else going on: Red paint absorbs most light with short wavelengths and reflects most light with longer wavelengths; when illuminated with white light, it appears red. Similar statements apply to green and blue paint. When we mix red and green paint, the red paint absorbs much of the green light, and the green paint absorbs much of the red light, and what's reflected is a spectral mix of red and green, but not much of either—we see brown.

This, by the way, is the formal explanation of the claim that "lights mix additively, while paints mix subtractively."

But the statement that red, green, and blue are primary is only part true. Indeed, choosing any three points on the curve above covers *some* portion of all possible perceptual responses and misses others. To really do the job, you'd need infinitely many "primaries" consisting of all the monospectral lights.

28.6.2 Purple Isn't a Real Color

People are sometimes told that "purple isn't a real color," because it doesn't appear in the rainbow. (Purple, as we mentioned, is how we describe the color sensation produced by a mix of red and blue-to-violet light, i.e., near the straight edge of the horseshoe shape.) It's true that it's not a color sensation corresponding to a monospectral source, but it certainly is a color sensation.

28.6.3 Objects Have Colors; You Can Tell by Looking at Them in White Light

The claim that objects have colors that are revealed by exposing them to white light can perhaps be better stated by saying that objects illuminated by sunlight reflect light with a spectral distribution that provokes a color response in our brains. But there are many kinds of "white light," and every actor knows that the white lights on stage and the white light of the sun are very different, and require

different cosmetics. Furthermore, for objects with highly peaked reflectance spectra, the existence of peaks or valleys in the illuminant spectrum can have drastic effects on the reflected light. It's perhaps better to say the following: "Objects have reflectance spectra, and the human brain is surprisingly good at predicting, for natural objects with not-too-peaked reflectance spectra, which are common, how an object seen under unusual illumination (shade, 'colored' light) will look under illumination by sunlight. This fairly consistent prediction could be called the 'color' of the object."

28.6.4 Blue and Green Make Cyan

Various claims about how colors mix are commonplace. In the case of paint color mixes, they're often misleading. For instance, painting with a blue watercolor, letting it dry, and then painting a red stripe over the blue leads to one thing; doing this in the opposite order leads to another. Mixing the colors before painting leads to a third. So any claims about mixing of colors must include the mixing process to be testable. In the case of colors atop others (see Figure 28.14), one can think of light as being reflected from the top color, from the bottom color after passing through the top, or from the underlying surface after passing through both. If we assume that each time light passes through a color-layer, some fraction of the energy at certain wavelengths is absorbed, this last kind of light passes twice through each paint layer, while the first kind never passes through any paint layer. The **Kubelka-Munk** coloring model [Kub54] carries out this analysis in detail.

This mixing problem is further compounded by the difference in the way lights mix and pigments mix; the distinction here is purely *physical*. If I shine a red and a green light onto a uniformly reflective piece of white paper, the reflected light will appear yellow. By contrast, if I have a red paint or dye and apply it to a white piece of paper, it absorbs colors outside the long-wavelength part of the spectrum so that only light we perceive as "red" gets reflected. If I mix this with a green paint or dye that absorbs all light except that in the green-percept part of the spectrum, the two together will absorb almost all light. If the paints or dyes were ideal, the result would be black paint; in practice, as noted earlier, we often get a muddy brown, indicating that very little light is reflected. These two phenomena are given the misleading names **additive color** and **subtractive color,** respectively; in fact, it's spectra that are being added or filtered, and the color perception mechanism remains unchanged, as we said in Section 28.6.1.

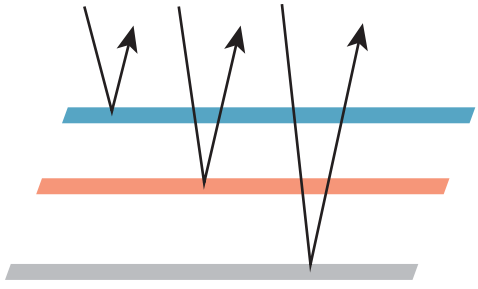


Figure 28.14: One color painted atop another. Light can be reflected from the top, from the bottom after passing through the top, or from the substrate on which the bottom is painted. Assuming some attenuation for each time the light passes through a paint layer, we get a model of the reflected light.

28.6.5 Color Is RGB

As computer displays have become commonplace and dialog boxes for choosing colors with RGB sliders have proliferated, one sometimes hears that color is just a mix of red, green, and blue. As we've seen already, there are many colors that cannot be made either by mixing red, green, and blue dyes/inks/paints or by mixing red, green, and blue lights. It *is* true that such mixes can generate a great many colors, but not all.

28.7 Color Perception Strengths and Weaknesses

The physiological description of sensor responses to light is still a step away from the perception of color; that happens in the brain. When those perceptions *do* occur, we confidently say that we saw something red, or blue, or yellow. But numerous optical illusions show that we may be overconfident. We can summarize a few key things. We're good at

- Detecting differences between adjacent colors
- Maintaining our sense of the "color of an object" in the presence of changing illumination (see Figure 28.15)

We're not very good at telling whether two widely separated colors are the same, or remembering a color from one day to the next. Then again, given the changing lighting circumstances we constantly encounter, this is probably an advantage rather than a limitation.

28.8 Standard Description of Colors

With the goal of having a common language for describing color, there's been a great deal of work in providing standards. The Pantone™ color-matching system is a naming system in which a wide variety of color chips are given standard numbers so that a printer can say, for instance, "I need Pantone 170C here." The numbers refer to calibrated mixes of certain standardized inks.

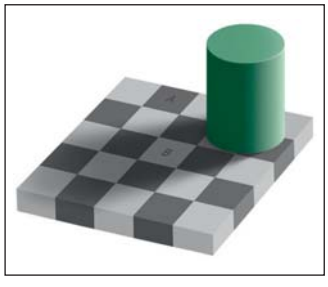


Figure 28.15: The squares labeled A and B have identical gray values, but we perceive them as very different shades of gray; indeed, we're inclined to call one a "white square" and the other a "black square." One may regard this as a failure of the visual system to "recognize the same color," but it's more appropriate to regard it as the success of the visual system in detecting color constancy in the presence of varying illumination: We perceive all the black squares to be black even though the actual gray values in the image vary substantially. (Courtesy of Edward H. Adelson.)

There's also the widely used Munsell color-order system [Fi76], in which a wide range of colors are organized in a three-dimensional system of hue, value (i.e., lightness), and chroma (i.e., saturation or "color purity"), and in which adjacent colors have equal perceived "distance" in color space (as judged by a wide collection of observers).

28.8.1 The CIE Description of Color

We have observed that monospectral lights provoke a wide range of sensor responses, plotted on the horseshoe-shaped curve. We've also seen that choosing three monospectral lights in the red, green, and blue areas of the spectrum (we'll call these primaries for the remainder of this section) allows us to produce, by combining them, many familiar color sensations, but not by any means *all*. As we said earlier, when we consider a color like orange, we find that no combination of our red, green, and blue primary lights gets us light that we perceive as orange. We can, through subterfuge, still express the orange light as a sum of the red, green, and blue primaries, however. What we *really* want is to say that "orange looks like about a half-and-half mix of red and green, and then move *away* from blue." In equations, we'd write something like

$$orange = .45red + .45green - 0.1blue. \tag{28.11}$$

Of course, we can't take away blue light that isn't there, but we can add blue light to the orange. If we find that

$$1.0orange + 0.1blue = .45red + .45green, \tag{28.12}$$

in the sense that the color mixes on the left and right produce the same sensor responses, then we'll express that numerically with Equation 28.11. In this way, we can find what mixes of our primaries are needed to match *any* monospectral light L, and plot the result as a function of the wavelength of L; the result has the shape shown in Figure 28.16. These three "color matching functions," $\bar{r}$, $\bar{g}$, and $\bar{b}$, tell us how much of our red, green, and blue primaries need to be mixed to generate each monospectral light. For example, to make light that looked like 500 nm monospectral light, we'd have to combine about equal parts of blue and green, and subtract quite a lot of red (i.e., we'd use $\bar{r}(500)$, $\bar{g}(500)$, and $\bar{b}(500)$ as the mixing coefficients). To make something resembling 650 nm light, we'd use lots of red, a little green, and no blue.

What about a 50-50 mix of 500 nm and 650 nm light? We'd use a 50-50 mix of the two color matches above. Because such a mix has all coefficients positive, it's actually possible to make it with our red, green, and blue standard monospectral lights. In general, if we have a light with a spectral power distribution P, we can find the "mixing coefficients" by applying the idea above to each wavelength, that is, we compute

$$c_r = \int_{400}^{700} P(\lambda)\bar{r}(\lambda)\, d\lambda, \tag{28.13}$$

$$c_g = \int_{400}^{700} P(\lambda)\bar{g}(\lambda)\, d\lambda, \text{ and} \tag{28.14}$$

$$c_b = \int_{400}^{700} P(\lambda)\bar{b}(\lambda)\, d\lambda, \tag{28.15}$$

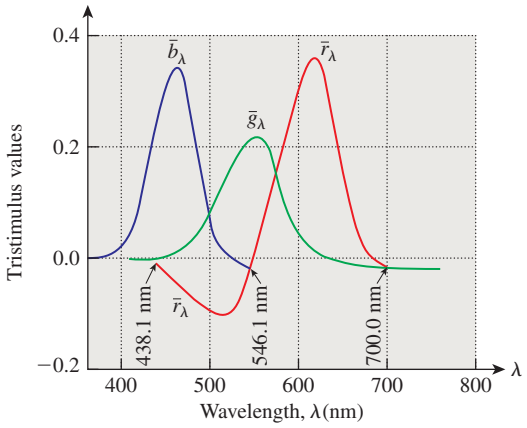


Figure 28.16: The color-matching functions, which indicate, for each wavelength, how much of a standard red, green, and blue light must be mixed to produce the same sensor responses as a monospectral light of wavelength λ. At least one mixing coefficient is negative for many monospectral lights, indicating the impossibility of making those colors as mixes of red, green, and blue.

and use these as the amounts of our red, green, and blue primaries. (Of course, if any of the three computed coefficients is negative, we cannot reproduce the color with our sources.)

Unfortunately, the set of all convex combinations of our three primaries doesn't include all possible colors; geometrically, the triangle whose vertices correspond to our primaries is a proper subset of the horseshoe-shaped set of sensor responses.

In 1931, the CIE defined three standard primaries, which it called X, Y, and Z, with the property that the triangle with these three as vertices actually includes all possible sensor responses. To do so, the CIE had to create primaries that had *negative* regions in their spectra, that is, they did not correspond to physically realizable light sources. Nonetheless, these primaries have certain advantages.

- The Y primary was defined so that its color-matching function was exactly the luminous efficiency curve; this means that for any spectral light source, T, written as a combination

$$T = c_x\mathbf{X} + c_y\mathbf{Y} + c_z\mathbf{Z}, \tag{28.16}$$

  the number c_y will be the perceived intensity of the light. This was significant in developing black-and-white televisions: The signal had to transmit in some form the Y-component of the lights that the camera was seeing.[2] Later, when color signals began to be broadcast, the c_x and c_z data were sent in a different band; color televisions could decode these, and black-and-white televisions could ignore them.
- The color-matching functions for **X**, **Y**, and **Z** are everywhere non-negative (see Figure 28.17), so all colors are expressed as non-negative linear combinations of the primaries.

2. The value c_y itself is not what's transmitted; more on this later.

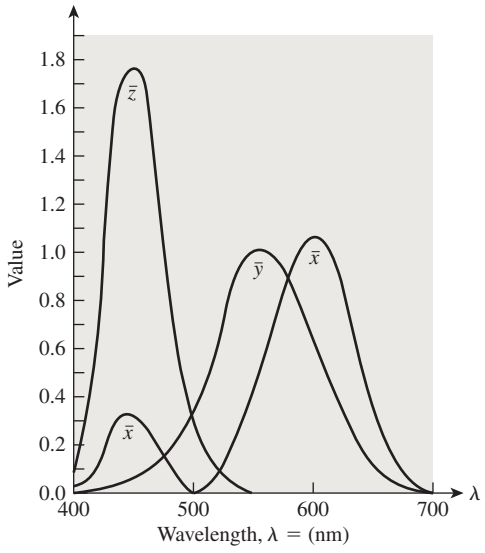


Figure 28.17: The color-matching functions $\bar{x}, \bar{y}$, and $\bar{z}$ for the 1931 CIE primaries.

- Because the red, green, and blue primaries can be identified as points in *XYZ*-space (i.e., as a linear combination of **X**, **Y**, and **Z**), any combination of them can be so expressed as well; thus, there's a direct conversion from *XYZ* to *RGB* coefficients (and vice versa).

In analogy with the color-matching functions for red, green, and blue, a light whose spectral power distribution is P can be expressed as

$$X\mathbf{X} + Y\mathbf{Y} + Z\mathbf{Z}, \tag{28.17}$$

where

$$X = k \int P(\lambda)\bar{x}(\lambda)\, d\lambda, \tag{28.18}$$

$$Y = k \int P(\lambda)\bar{y}(\lambda)\, d\lambda, \text{ and} \tag{28.19}$$

$$Z = k \int P(\lambda)\bar{z}(\lambda)\, d\lambda. \tag{28.20}$$

(More precisely: The light with power distribution $X\mathbf{X} + Y\mathbf{Y} + Z\mathbf{Z}$ and the light with power distribution P will evoke the same color response.)

In practice, such integrations are computed numerically, using the values of the matching functions tabulated at 1 nm intervals that are found in texts such as [WS82, BS81]. The constant k is 680 lm W^{-1}. But we also sometimes compute the "colors" for the reflectance spectrum of some reflecting object. In this case, one must choose a standard light source as a reference for "white" and illuminate the surface. The values are usually scaled so that a completely reflective surface has a Y-value of 100; thus,

$$k = \frac{100}{\int W(\lambda)\bar{y}(\lambda)\, d\lambda}, \tag{28.21}$$

where W is the spectral power distribution of the standard white light we're using.

Suppose that the light C produces the same sensor responses as

$$X\mathbf{X} + Y\mathbf{Y} + Z\mathbf{Z}. \tag{28.22}$$

In that case, we write

$$C = X\mathbf{X} + Y\mathbf{Y} + Z\mathbf{Z}. \tag{28.23}$$

The CIE defines numbers that are independent of the overall brightness by dividing through by $X + Y + Z$; doubling the incoming light doubles each of X, Y, and Z, but also doubles their sum, so the quotients

$$x = \frac{X}{X + Y + Z}, \tag{28.24}$$

$$y = \frac{Y}{X + Y + Z}, \text{ and} \tag{28.25}$$

$$z = \frac{Z}{X + Y + Z} \tag{28.26}$$

remain unchanged. Note that the sum $x + y + z$ is always 1, so if we know x and y, we can compute z. Thus, the collection of intensity-independent colors can be plotted on just the xy-plane; the result is the **CIE chromaticity diagram** shown in Figure 28.18. Notice that **X** and **Y** were chosen so that the diagram is tangent to the x- and y-axes.

Near the center of the "horseshoe" is **illuminant C,** which is a standard reference "white," based on daylight. Unfortunately, it doesn't correspond to

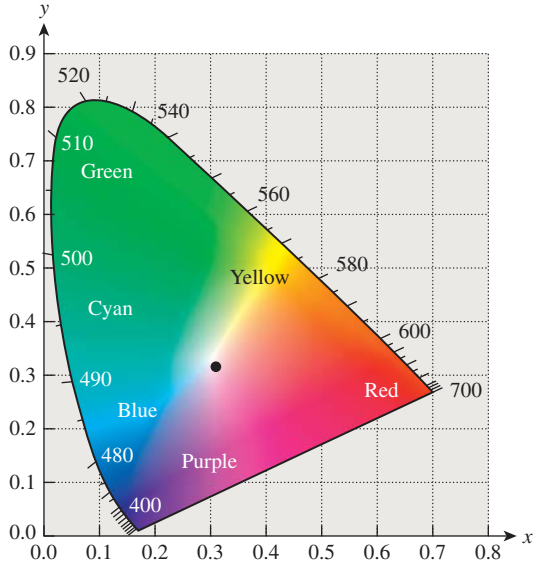


Figure 28.18: The CIE chromaticity diagram. The boundary consists of chromaticities corresponding to monospectral lights of the given wavelengths, shown in nanometers. The dot in the center is a standard "white" light called "illuminant C."

$x = y = z = 1/3$, although it is close. (Other reference whites are described in Section 28.11.)

Note that if we know x and y, we can compute $z = 1 - (x + y)$, but this does not allow us to recover X, Y, and Z; for that we need at least one more piece of information (all xyz-triples lie on a planar subspace of XYZ-space). Typically we recover XYZ from x, y, and Y (the luminance value). The formulas are

$$X = \frac{x}{y} Y, \tag{28.27}$$

$$Y = Y, \text{ and} \tag{28.28}$$

$$Z = \frac{1 - (x + y)}{y} Y. \tag{28.29}$$

28.8.2 Applications of the Chromaticity Diagram

The chromaticity diagram has several applications.

First, we can use the diagram to define **complementary colors:** Colors are complementary if they can be combined to form illuminant C (e.g., D and F in Figure 28.19). If one requires a half-and-half mix in the definition, then some colors, like B, have no complement.

Second, the diagram lets us make precise our notion of **excitation purity:** A color like the one indicated by point A in Figure 28.18 can be represented by combining illuminant C with the pure-spectral color B. The closer A is to B, the more spectrally pure it is. So we can define the excitation purity to be the ratio of the length AC to the length BC. We extend this definition to C by saying that its excitation purity is zero. For some colors, like F, the ray from C through F meets the boundary of the horseshoe at a nonspectral point; such colors are called **nonspectral;** but the ratio CF to CG still makes sense, and we can define excitation purity this way. The dominant wavelength, however, is more problematic; the standard is to say that the dominant wavelength is a "complementary" one at B, which would be denoted 555 nm c, where the "c" indicates complementarity.

A third use of the chromaticity diagram is the indication of **gamuts:** Any light-producing device (like an LCD monitor) can produce a range of colors that can be indicated on the chromaticity diagram. Colors outside this gamut cannot be produced by the device. (Similarly, printing devices have gamuts, once one defines a standard illuminant under which the printed page will be viewed.) A device that can produce two colors can also produce (by adjusting the amounts of each) chromaticity values that are convex combinations of the two. In Figure 28.20, lights whose chromaticity values are I and J can be combined to form chromaticity values on the line segment between them; adding a third color K gives a gamut consisting of a whole triangle. Clearly there's no triangle with vertices in the horseshoe that contains the entire horseshoe; thus, no three-color display, no matter how perfectly calibrated, can produce all color percepts.

Note that printer gamuts are typically far smaller than those of displays; in high-end printers, this can be partially remedied by the use of **spot color**—additional inks placed in the printer to expand the gamut so as to include a particular color. But in general, getting faithful print versions of images from a display is impossible. The problem of gamut matching (i.e., finding reasonable mappings from the gamut of one device to that of another) remains a serious challenge.

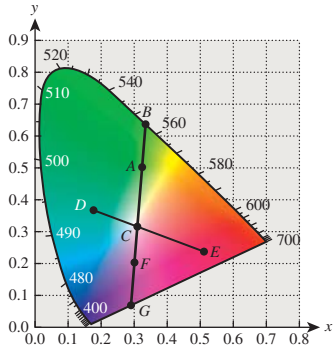


Figure 28.19: Colors on the chromaticity diagram. D and E are complementary.

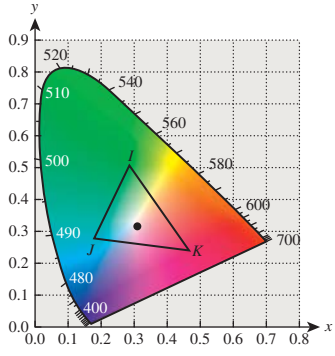


Figure 28.20: Mixing of colors in the chromaticity diagram. Colors on the line IJ can be created by mixing the colors I and J; all colors in the triangle IJK can be created by mixing the colors I, J, and K.

28.9 Perceptual Color Spaces

The CIE color system is remarkably useful; it's so standard that colorimeters measure X, Y, and Z values of light, for instance. In the CIE system, each color has XYZ-coordinates; it's tempting to measure the "distance" between two colors $C_1 = X_1\mathbf{X} + Y_1\mathbf{Y} + Z_1\mathbf{Z}$ and $C_2 = X_2\mathbf{X} + Y_2\mathbf{Y} + Z_2\mathbf{Z}$ by computing the Euclidean distance between the triples (X_1, Y_1, Z_1) and (X_2, Y_2, Z_2). Unfortunately, this does *not* correspond to the *perceived* color distance: If C_1 and C_2 have the same Euclidean distance as C_3 and C_4, the perceived distance between them may be very different.

Fortunately, one can transform the XYZ-coordinates, *nonlinearly,* to get new coordinates in which the Euclidean distance *does* correspond to perceptual distance. The 1960 CIE *Luv* color coordinates were developed to meet this need, but they were superseded by the 1976 CIE $L^*u^*v^*$**uniform color space.** Letting X_w, Y_w, and Z_w denote the XYZ-coordinates of the color to be used as white, the $L^*u^*v^*$coordinates of a color with **XYZ**-coordinates (X, Y, Z) are defined by the formula for L^* given in Equation 28.10, and

$$u' = \frac{4X}{X + 15Y + 3Z}, \tag{28.30}$$

$$v' = \frac{9Y}{X + 15Y + 3Z}, \tag{28.31}$$

$$u'_w = \frac{4X_w}{X_w + 15Y_w + 3Z_w}, \tag{28.32}$$

$$v'_w = \frac{9Y_w}{X_w + 15Y_w + 3Z_w}, \tag{28.33}$$

$$u^* = 13L^*(u' - u'_w), \text{ and} \tag{28.34}$$

$$v^* = 13L^*(v' - v'_w). \tag{28.35}$$

The CIE has also defined $L^*a^*b^*$ color coordinates (sometimes called "Lab" color) by

$$a^* = 500\left[(X/X_w)^{\frac{1}{3}} - (Y/Y_w)^{\frac{1}{3}}\right] \text{ and} \tag{28.36}$$

$$b^* = 500\left[(Y/X_w)^{\frac{1}{3}} - (Z/Z_w)^{\frac{1}{3}}\right], \tag{28.37}$$

where X_w, Y_w, and Z_w denote the XYZ-coordinates of the white point. Both $L^*u^*v^*$and $L^*a^*b^*$can be used to measure "distance" in color space, and both see frequent use in computer graphics, although $L^*a^*b^*$seems to be more widely used in the description of displayed colors.

28.9.1 Variations and Miscellany

The CIE diagram we've shown is based on the 1931 tabulation of colors, in which samples subtended a 2° field of view on the retina. There's also a 1964 tabulation for a 10° field of view, emphasizing larger areas of constant color. For much of computer graphics, the narrower field of view is more relevant.

The mapping from the space of all spectra (which is infinite-dimensional) to the space of response triples (which is three-dimensional) is more or less linear

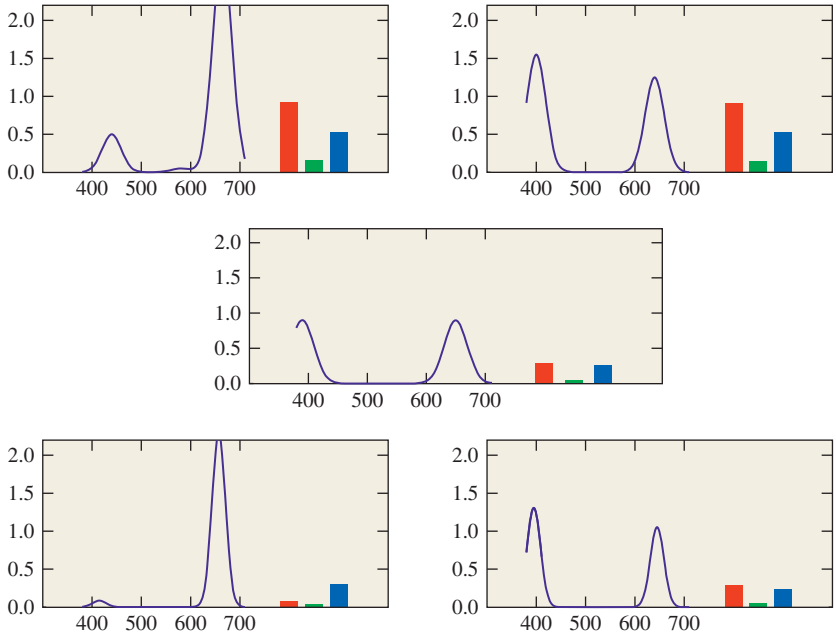


Figure 28.21: Two metameric light spectra (top) are each multiplied (wavelength by wavelength) by the reflectance spectrum (middle). The resultant spectra are no longer metameric. (Next to the spectrum for each light are its corresponding RGB response values.)

(at least for not-too-bright lights, where saturation comes into play, and not-too-dim lights, where photopic/scotopic differences enter); this means that it's necessarily many-to-one. Different spectra that generate the same response values are called **metamers;** metameric lights are interesting because, upon reflection by a surface, they can become nonmetamers (see Figure 28.21). In practice, most reflectance functions are nonspiky enough that metameric effects like this are not significant, although with LED lamps, which tend to have spikier spectra, the problem may be more serious.

The colors in the $x+y+z=1$ plane of the CIE XYZ space are *not* all possible colors. As the sum $x + y + z$ varies, other colors appear (such as maroon). Furthermore, colors like brown, which are generally used to describe reflective color rather than emissive color, tend not to appear at all.

The colors purple and violet are often considered to be synonymous, but violet is the name for a pure spectral color (at about 380 nm, just on the edge of perceptibility), while as we said, purple is the name for points on or near the straight bottom edge of the CIE horseshoe.

28.10 Intermezzo

Let's pause and note the important points so far. First, color is a three-dimensional perceptual phenomenon evoked by the arrival of different spectral power distributions at the eye. Any color percept can be generated by a combination of the CIE

primaries **X**, **Y**, and **Z**; if a color C is generated by $X\mathbf{X} + Y\mathbf{Y} + Z\mathbf{Z}$, we can think of X, Y, and Z as "coordinates" for that color in the space of all possible colors.

There are other coordinate systems on the space of colors, such as the CIE $L^*u^*v^*$and $L^*a^*b^*$, in which L^*captures the notion of intensity, while the other coordinates encode chromaticity. In these systems, distances between color triples correspond to perceptual distances much more closely than do distances between (X, Y, Z)-coordinate triples. But the coordinates in these systems are *not* linear functions of the X, Y, Z-coordinates (which are in turn linear functions of radiometric quantities), so they are not suitable for computations with a physical basis.

In both of the "perceptual" coordinate systems, there's a free parameter, namely, the color chosen as "white." Without the knowledge of the white point, you cannot convert an $L^*u^*v^*$ coordinate triple into an XYZ triple, for instance.

We now move on from the description of color to the question of how to represent color in an image file, a television signal, etc. Considering that there's only a half-century of experience in this regard, a surprisingly large number of representation methods have arisen.

28.11 White

As we mentioned earlier, many spectral power distributions appear white, so picking a particular white point can be a challenge. And an SPD that looks white at one intensity may look yellow at another intensity, because of the adaptation of the eye. Furthermore, the surroundings may have a substantial impact on the appearance of a color; if we watch a slide show in a dark room, showing a scene illuminated by incandescent lamps, we rapidly accommodate so that the white point of the slides appears white. But if that same slide show is shown in a well-lit room with white walls, the "white" within the slides may appear yellow, for instance.

The CIE has defined several standard "whites"; the simplest (from the point of view of computation) is illuminant E, which has a constant SPD across the range of visible light. Illuminant C, now deprecated but still widely used, attempts to approximate the white of sunlight. More common in modern usage are the D series of illuminants, which are tabulated by the CIE in 5 nm increments. Many of the most useful are, at a gross level, quite similar to blackbody distributions, and the names indicate this: D65 is similar to 6500 K black body radiation, D50 is similar to 5000 K radiation, etc. The photography industry uses the D55 standard; either this or D65 is a good choice for much of computer graphics.

28.12 Encoding of Intensity, Exponents, and Gamma Correction

As mentioned above, the CIE standard for defining L^* uses a $\frac{1}{3}$-power law; the idea is that L^* is a reasonable measure of perceived brightness of light (at least within a modest range of luminances around the luminance of some reference white). Suppose that you wanted to store or transmit information about light without using too many bits. If you were engaged in physical measurements, you'd just want to choose some numeric representation of intensity. But if you were planning to use the information about light in some way that involved a human looking at it (e.g., if you were a television engineer trying to decide what information to encode in

the first black-and-white television signal!), you might argue that if a human can distinguish, say, 100 levels of intensity, then we should use 100 different numbers to represent these. It would be silly to use 200 different numbers, because we'd have different numbers representing different, but indistinguishable, intensities. If we were representing values in binary, we'd be wasting a bit by going from 100 values to 200 values. Similarly, if we encoded only 50 different intensity levels, we'd get nonsmooth intensity gradients in our display.

If you simply take all possible intensity values and divide them equally (i.e., you quantize the intensity signal), you'd find that to capture perceptual differences that were significant at the low-intensity levels you'd need to use very small buckets. But those same buckets would be redundant at high-intensity levels. In fact, you would be far better off encoding the number L^*, because each quantized range of L^* values would correspond to the same amount of perceptual variation. By choosing the bucket size correctly, you could most efficiently encode the brightness.

To recover the intensity at the receiving end of the channel, you would invert the formula for L^* (roughly, you'd take the third power of L^*, and multiply by the constant Y_n) and arrange for your television screen to emit the corresponding intensity.

As it happens, the cathode ray tubes (CRTs) that were used in early televisions have an interesting characteristic: The intensity emitted is proportional to the $\frac{5}{2}$ power of an applied voltage. Since $\frac{5}{2}$ is fairly close to three, this meant that you could take the L^* value and use it as a voltage to determine the color of each pixel, approximately.

To be clear: The visual system's response to intensity is nonlinear and looks approximately like $I^{1/3}$; the CRT's output intensity in response to applied voltage is also nonlinear and looks like $I = kV^{5/2}$. Combining these two results in a nearly linear overall effect (a $\frac{5}{6}$ power law).

In fact, video engineers defined a "signal *representative of* luminance" (which has later, in some video literature, been incorrectly called "luminance"); this signal approximately encodes the 0.42 power of luminance. Why use 0.42 instead of 0.33? One answer is that if you used 0.4 instead, then the $\frac{5}{2}$ power law of the CRT would cancel it exactly: This allows you to simplify the electronics in a consumer television, and at the cost of only a minor inefficiency in the encoding of the signal. The use of 0.42 instead of 0.4 has been explained by the observation that the viewing circumstances for television (much less bright than outdoors) are not the same as the circumstances under which the signal was captured (often bright lights or outdoors in daylight); the slight adjustment is meant to help compensate for this.

You can experience the distinction between high-light and low-light perception of intensity by considering a garden at midday on a slightly overcast day (so that the lighting is reasonably diffuse), and the same garden just after sunset on that day. Only the light levels change. Because our perception of "lightness" is supposed to be approximately logarithmic, the difference in lightness between the leaves of a plant and its flower should be the same at midday and at twilight. In practice, they are not, appearing to be lower-contrast at twilight, and we must do some adjusting to compensate.

To experience this effect directly, we can use the area surrounding some gray values as a proxy for the ambient illumination. Figure 28.22 show three gray squares surrounded by white and black borders. The gray squares in each column

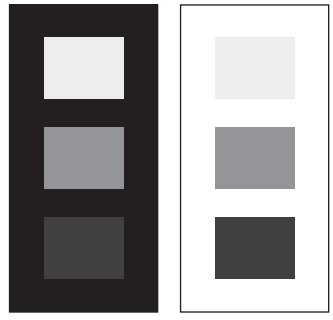


Figure 28.22: Surrounding context can vary our perception of tones. (Figure concept from Poynton [Poyb].)

are identical, but the contrasts in the left column appear less than the contrasts in the right column.

The signal representative of luminance (the 0.42 power of luminance) seems as if it should be a part of a video signal; in fact, a video signal starts out as three values, r, g, and b representing the amounts of red, green, and blue light in a way that's linear in the intensity (if you double the intensity, then each of r, g, and b will double). The **luma** is then a weighted sum of $r^{0.42}$, $g^{0.42}$, and $b^{0.42}$. The difference between these values and the values determined by computing luminance directly and raising *that* to the 0.42 power is generally insignificant (another application of the Noncommutativity principle) and luma is used as the Y' component of the $Y'IQ$ color model described below. The prime on Y' indicates that this coordinate does *not* vary linearly as a function of the light intensity. Ordinary video cameras compute R, G, and B values that, for a given aperture and white balance, are proportional to incoming intensities in the appropriate wavelength ranges, and raise these to the 0.45 power; the values they produce should therefore be called R', G', and B', following the naming convention. To recover the original R, G, and B values, these must be raised to the 2.2 power. And to transform to other color spaces, we typically must first recover R, G, and B, and then perform the conversion, since most color transformations are described in terms of things like R, G, and B that vary linearly with energy.

The exponent 2.2 that is used to convert video $R'G'B'$ values back to RGB is often called **gamma,** and the process of raising values to some power around 2.2 is known as **gamma correction.** The number 2.2 is by no means universal; other gamma values have been used in various image formats over the years, and many image display programs allow the user to "adjust gamma" to modify the exponent used in the display process.

28.13 Describing Color

In computer graphics, we often need to describe color mathematically. Because the physical interaction of light and surfaces occurs in ways determined by their spectra rather than their colors, we don't use the $L^*u^*v^*$description of light when we want to model this physical interaction. And because the values we compute while rendering are typically spectral radiance values (possibly for some fairly broadband spectra, i.e., the radiance for the bottom, middle, and top thirds of the visible spectrum), which then must be converted to values that govern three display brightnesses, it's best to separate the physical models used in rendering from our description of colors that appear on our displays or printers.

So we'll now present several color models used to describe the colors that our devices can produce. Typically these color models are **bounded,** in the sense that they can only describe colors up to a certain intensity (or generally only a subset of the colors up to some intensity value). This matches the physical characteristics of many devices: An LCD monitor cannot produce more than a certain brightness; the light reflected from a printed page cannot exceed the light arriving at the page, etc.

The choice of a color model may be motivated by simplicity (as in the RGB model), ease of use (the HSV and HLS models), or particular engineering concerns (like the $Y'IQ$ model used for the broadcast of color television signals or the CMY model for printing). And with the widespread interchange of imagery among different devices, there are color models whose design is based on lossless

exchange of imagery, in which not only are color coefficients included in one's data, but so are descriptions of the model used to represent the data. The International Color Consortium notion of *profiles* is one of these [Con12], used to describe a device's color space, and thus support reproduction of similar colors across different devices and media; a far simpler (but less rich) approach is sRGB, a single standardized RGB color space discussed below.

In this section, we'll discuss several color models, their goals, and methods for interconversion.

We'll mostly follow the convention that says that quantities that vary linearly with the intensity of the light that they represent are denoted by unprimed letters, while those that vary nonlinearly are denoted with primes. Since primes get used for other reasons, and because of historical precedent, we won't be absolutely rigid in this.

You should understand, however, that conversion among models may not, in general, make sense, because of the context in which the model is described. CMY (a system used to describe ink amounts in printing) is based on the ideas of inks being applied to a certain white paper and illuminated by a certain light; the reflected light cannot be brighter than that illuminant. Converting a color used in an ultrabright display to CMY therefore cannot be done: No CMY value represents that bright a color. There is a fine art in mapping the gamut of one device to that of another; appropriate mappings may depend on intended uses. The message to take away from this is that when you produce images in computer graphics, you should attempt to store them losslessly, with important information (What white point is being used? What primaries?) recorded in the image file so that they can later be converted to other formats. In general, conversion from format A to format B and back again may end up corrupting an image.

28.13.1 The RGB Color Model

Most displays, whether LCDs, CRTs, or DLPs, describe each pixel in terms of three numbers called r, g, and b, which in turn correspond to the degree to which three lights contribute to the appearance of that pixel. In an LCD, the three lights are in fact three filters, each filtering a backlight and allowing differing amounts of red, green, and blue light through; the three filters are vertically aligned as stripes to form a square "pixel." In the case of a CRT, the three are phosphors that glow when struck by an electron beam; they're typically arranged in a pattern in which each pixel consists of three colored dots in a closely spaced triangle. The precise spectra of the red, green, and blue lights being blended are not necessarily specified in RGB image data, so the numbers r, g, and b have only a vague display-specific meaning. Still, the general shape of the set of displayable colors within the CIE XYZ-space can be seen in Figure 28.23.

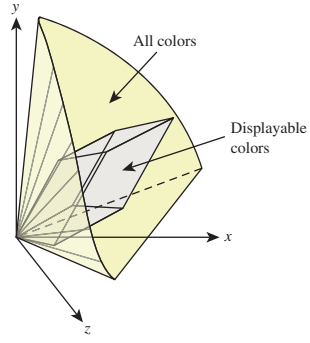


Figure 28.23: The color gamut for a typical display within the CIE XYZ color space. Note that white can be displayed very brightly, while red, green, and blue have much less intensity. Note, too, that many colors are not within the display gamut at all, particularly bright and dim ones.

The good news is that with the development of video standards and HDTV standards, a particular set of three colors has come to be fairly standard; these are used in the sRGB standard, described below. But for older graphics images, it's a mistake to assume that the RGB values have any particular meaning; it may be best to experiment with adjusting the meaning (in the sense of *XYZ*-coordinates) of R, G, and B until the image looks best, and then transform the result into sRGB for future use.

The RGB color cube is usually drawn not as it embeds in the CIE XYZ space, but instead with red, green, and blue as the coordinate axes, as in Figure 28.24.

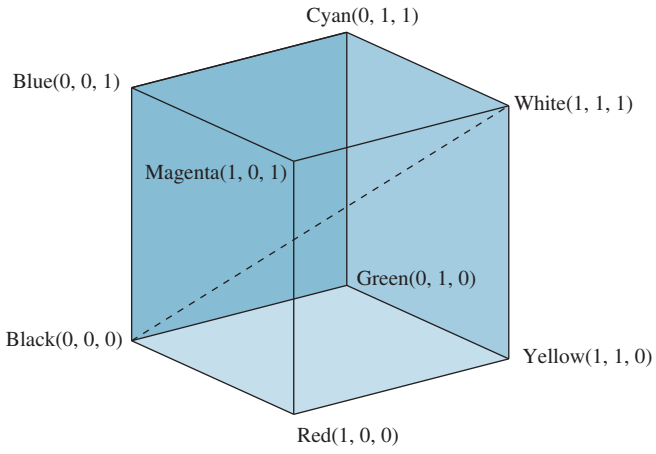


Figure 28.24: The RGB cube. Grays lie along the main diagonal.

In this form, grays lie along the main diagonal; moving away from this diagonal gives increasingly saturated colors. Viewed this way, we are taking a part of the space of colors and transforming it so that it looks like a rectilinear cube (which is a skewed parallelepiped in *XYZ*-coordinates). For this reason, people sometimes refer to an RGB color space, rather than RGB coordinates on colors.

To return to the general (prestandards) case: The color gamut associated with the RGB color cube depends on the primary colors producible by the display (the LCD's color stripes or the CRT's phosphors). So an RGB triple like $(0.5, 0.7, 0.1)$ may represent rather different greenish-yellows on different devices.

Fortunately, we have a universal description—CIE XYZ values—to which we can convert. Unfortunately, the conversion requires knowing something about the primary colors of our device. These can be measured with a colorimeter by making all pixels red, observing the color in XYZ space, that is, (X_r, Y_r, Z_r); then making all pixels green, observing the XYZ color (X_g, Y_g, Z_g), and then doing the same for blue to get (X_b, Y_b, Z_b). If we then display

$$rR + gG + bB, \tag{28.38}$$

the resultant XYZ color coefficient triple will be

$$r(X_r, Y_r, Z_r) + g(X_g, Y_g, Z_g) + b(X_b, Y_b, Z_b) = \begin{bmatrix} X_r & X_g & X_b \\ Y_r & Y_g & Y_b \\ Z_r & Z_g & Z_b \end{bmatrix} \begin{bmatrix} r \\ g \\ b \end{bmatrix}. \tag{28.39}$$

In other words, the result will be the coefficients of **X**, **Y**, and **Z** in the CIE XYZ description of the color. If we have two displays with corresponding matrices $\mathbf{M}_1$ and $\mathbf{M}_2$, we can convert the colors of each display to XYZ space with the respective matrices. Starting with the color

$$rR + gG + bB \tag{28.40}$$

on display 1, we get to the XYZ color

$$\mathbf{M}_1 \begin{bmatrix} r \\ g \\ b \end{bmatrix}, \tag{28.41}$$

which in turn corresponds to the color triple

$$\mathbf{M}_2^{-1}(\mathbf{M}_1 \begin{bmatrix} r \\ g \\ b \end{bmatrix}) = (\mathbf{M}_2^{-1}\mathbf{M}_1) \begin{bmatrix} r \\ g \\ b \end{bmatrix} \tag{28.42}$$

for display 2, so the matrix $\mathbf{M}_2^{-1}\mathbf{M}_1$ will take RGB color descriptions for display 1 to those for display 2.

It will often happen that some RGB color triple for display 1, after multiplication by the transition matrix, will produce a color triple for display 2 some of whose entries are greater than one, or less than zero. This indicates that there's a color in display 1's gamut that is outside the gamut of display 2. What can we do in such a case? There are several solutions, ranging from the simple to the complex. We can simply ignore the transition matrix, replacing it with the identity; this fails to match colors, but avoids the gamut-overshoot issue entirely (indeed, this is mostly what's done in practice with images transferred over the Internet). We can clamp the resultant color values between 0 and 1; this produces unpleasant artifacts in the darkest and brightest areas of the image, but it's simple. Or we can take a more sophisticated approach like the ones described by Hall [Hal12], or those described in the ICC profile model's rendering intents [Con12], which include a strategy that maps the white point of the source image to the white point of the medium on which it is to be displayed, and then warps other colors accordingly, a strategy that attempts to map the most saturated colors to the most saturated colors, and then warps others to be consistent with this, and a strategy that attempts to capture the perceptual relations among colors in an image as faithfully as possible. (Of course, this depends on our knowing the white point for the medium.)

The **sRGB** standard proposed certain "standard" colors for R, G, and B, based on the observation that many displays were closely matched to these standards; their relationships to CIE XYZ coordinates are given by a linear mapping:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 3.2410 & -1.5374 & -0.4986 \\ -0.9692 & .18760 & 0.0416 \\ 0.0556 & -0.2040 & 1.0570 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}. \tag{28.43}$$

Of course, for two displays that both have RGB primaries described by this relation, the transition matrix *will* be the identity.

28.14 CMY and CMYK Color

Cyan-Magenta-Yellow (CMY) color descriptions are used for printers, where the inks are materials that reflect some portion of incoming light, absorbing other portions. Cyan ink absorbs red light, but reflects blue and green (i.e., its reflectance of long-wavelength visible light is low, but of short- and medium-wavelength light is high); magenta absorbs green, and yellow absorbs blue. Once again, the exact details of which wavelengths are absorbed must be based on measurement.

Colors are described as a mix of cyan, magenta, and yellow. When two inks are mixed, the light that's reflected is that which is not absorbed by *either* ink. So a mix of cyan and magenta absorbs both red and green, resulting in something that reflects blue light. Thus, for CMY colors, we write

$$U = cC + mM + yY \tag{28.44}$$

and then denote the color by the number triple (c, m, y). In this form, the CMY color $(0, 0, 0)$ is white, and $(1, 1, 1)$ is black.

Lacking exact measurement of inks, the usual conversion from RGB to CMY is that the RGB color (r, g, b) is the same as the CMY color $(1 - r, 1 - g, 1 - b)$. Because of the interactions between inks, the arrangements of dots in dot-based printing, and many other factors, this should be regarded as a *gross approximation*. Indeed, in computer graphics there's almost no situation in which you should represent an image by CMY colors. Most modern printers have software that accepts RGB colors and converts them, as well as possible for the particular printer technology, to amounts of ink to be used at each point.

In practice, the color $(1, 1, 1)$ is not really very black; the mix of cyan, magenta, and yellow inks doesn't really manage to absorb all light. So printers often have a fourth ink, which is black (denoted K). This is used to replace parts of the darker mixes of C, M, and Y.

28.15 The YIQ Color Model

The YIQ color model (which we should call $Y'IQ$ to follow the convention about nonlinear coordinates) is used in U.S. commercial television broadcast. It's a nice example of a color model designed with engineering constraints in mind; these constraints were (a) the need to broadcast a signal that could be used to drive both black-and-white and color television receivers, and (b) the desire to use bandwidth most efficiently.

To satisfy the first goal, the YIQ color model's Y value is, as described above, the **luma,** which is

$$Y' = 0.299r^{0.42} + 0.587g^{0.42} + 0.114b^{0.42}, \tag{28.45}$$

and is therefore distinct from the IE XYZ model's Y value. (We've used a prime both to emphasize this distinction and to indicate that it does not vary linearly as a function of R, G, and B.) The I and Q values then remain to encode chrominance information. They are essentially rotated and scaled versions of the u^* and v^* values. We omit the details, because with the rapid growth in the use of component video, the YIQ standard is less and less relevant. Perhaps the most significant aspect of it is that the Y' of $Y'IQ$ is *not* the same as the Y coordinate in **XYZ**-coordinates, but instead is roughly similar to the 0.42 power of Y.

The allocation of bandwidth to the transmission of the three channels (which corresponds to the number of bits of precision with which each is communicated) is carefully chosen: 4 MHz is assigned to Y, 1.5 MHz to I, and 0.6 MHz to Q; this corresponds to our strong sensitivity to luminance and to sharp discontinuities in luminance (the result of using too few bits of precision), and to the relative sensitivities of the visual system to color variation along the I and the Q axes.

28.16 Video Standards

Modern component video is encoded in various ways that are similar to $Y'IQ$, in the sense that one component carries intensity information while two others carry chromaticity information. Following Poynton [Poya], let's examine one encoding a decoding process, starting from the unambiguous XYZ description of a color (see Figure 28.25).

The entire process is described by the HDTV standard [Uni90], informally known as Rec. 709. The transformation from XYZ to RGB (as specified in Rec. 709, so we append the subscript 709) is (to three decimal places)

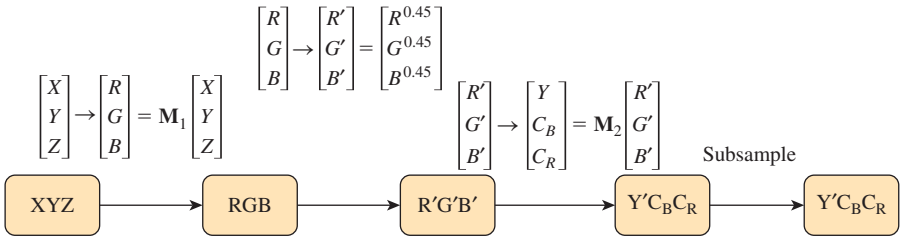


Figure 28.25: Converting from XYZ values to $Y'C_BC_R$ values. XYZ is converted to RGB by multiplication by a matrix $\mathbf{M}_1$; the RGB values are then nonlinearly encoded by a 0.45 *power function; the resultant values are then transformed by another matrix, $\mathbf{M}_2$, and shifted slightly, to form Y', C_B, and C_R, where Y' approximately represents intensity and the other two encode chrominance information. Finally, the resultant values are digitized by a step called the subsampling filter. Conversion to analog component video is similar, except that the subsampling filter is replaced by band-limiting.*

$$\begin{bmatrix} R_{709} \\ G_{709} \\ B_{709} \end{bmatrix} = \mathbf{M}_1 \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 3.24 & -1.54 & -0.5 \\ -0.97 & 1.88 & 0.04 \\ 0.06 & -0.20 & 1.06 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}. \tag{28.46}$$

The conversion to R', G', B' is very simple:

$$\begin{bmatrix} R'_{709} \\ G'_{709} \\ B'_{709} \end{bmatrix} = \begin{bmatrix} R_{709}^{0.45} \\ G_{709}^{0.45} \\ B_{709}^{0.45} \end{bmatrix}. \tag{28.47}$$

A second matrix operation converts the primed values into a luminance value and two chrominance values, while adding an offset to make the chrominance values lie in the range of 8-bit positive integers.

$$\begin{bmatrix} Y' \\ C_B \\ C_R \end{bmatrix} = \mathbf{vM}_2 \begin{bmatrix} R'_{709} \\ G'_{709} \\ B'_{709} \end{bmatrix} = \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} + \begin{bmatrix} 65.481 & 128.553 & 24.9965 \\ -37.797 & -74.203 & 112 \\ 112 & -93.786.20 & -18.214 \end{bmatrix} \begin{bmatrix} R'_{709} \\ G'_{709} \\ B'_{709} \end{bmatrix} \tag{28.48}$$

As R', G', and B' range from 0 to 1, the value Y' ranges from 16 to 255, while C_B and C_R go from $128 - 112 = 14$ to $128 + 112 = 240$.

If you happen to have R', G', and B' ranging from 0 to 255 (as you might in some computer representations of images), you'll need to first scale them appropriately (dividing by 255) before converting using Equation 28.48.

There is another standard for video—studio video—that requires a different transformation (albeit similar in form). Before converting to or from video, you must know which video format is in use.

28.17 HSV and HLS

The RGB cube is not ideal as a color-selection tool. For one thing, with its limited range (0 to 1 in R, G, and B) it's best suited to selecting *reflectances* in three wavelength bands, that is, it's well suited to "material colors" but not "colored lights," where the intensity can be arbitrarily large. Even for selecting reflectance colors,

RGB is not very convenient; the individual controls don't match our sense of the "independent" characteristics of a color like "how light it is" or "how saturated it is" or "what hue it is." As you adjust from red toward orange by slightly increasing the green component, your color also gets brighter, when all you really wanted was to change the hue.

Two alternative interfaces are widely available for color selection: the hue-saturation-value (HSV) interface, and the hue-lightness-saturation (HLS) interface. In each, hue is varied independently of the other qualities of color (such as how light it seems). These are useful tools, and they are somewhat more intuitive for users of paint programs than RGB mixes, but they're the wrong tool to use when you need to specify a colored light for rendering, or even need to specify a color for printing, since the conversion from HSV to RGB produces *only* an RGB specification; without a precise definition of RGB (sRGB might be a good choice), such a specification is ambiguous. The web material for this chapter contains a discussion of the conversion among RGB, HSV, and HLS.

28.17.1 Color Choice

No single color specification system is best for all users; even among systems designed for usability like HSV and HLS, preferences vary. Many programs wisely allow the user to pick colors with a dialog that can be toggled among several different modes, allowing direct RGB specifications with sliders or typed-in text values (typically 0 to 255), HSV selection via sliders, click-to-pick selection from a disklike display of colors (often adjustable with a third slider to adjust from dark to light), etc.; users quickly find which method best suits them in various circumstances.

28.17.2 Color Palettes

Our discussion of color has concentrated almost entirely on the description and selection of single colors. But when multiple colors are displayed together, interactions between them can be important. A number of peculiar optical illusions are based on "tricking" our color-perception system. For example, it's well known that the color surrounding a particular region can influence our perception of the color of that region (see Figure 28.26).

Figure 28.26: The simultaneous contrast effect. The two gray squares at the top appear to be different colors, but are in fact identical. The gray stripe on the bottom is a single color across its whole length.

When one chooses colors for a user interface, for instance, it's important to choose them so that they are harmonious, and so that effects like the simultaneous contrast artifacts of Figure 28.26 don't mask important design decisions. If, for instance, we're creating a drawing program, and all interface elements relating to *drawing* are in one color and all elements relating to *text* are in another, there will be a problem if some elements of each kind are displayed on different backgrounds that makes them appear to be unrelated.

Meier et al. [MSK04] have studied this problem extensively, and have developed interfaces for selecting color palettes rather than individual colors.

28.18 Interpolating Color

We often need to interpolate between colors in graphics, in situations ranging from simple design ("I'd like a color gradient from aqua to magenta on this background rectangle") to rendering ("I know the colors of this triangle at the three vertices,

but I need to interpolate colors along the edges and at the interior points"). Unfortunately, this isn't easy. For colors that are very similar (e.g., when their difference is just a little more than the just-noticeable difference), almost any interpolation scheme will work, including interpolating the RGB coefficients (or whatever other tuples you might be using to represent the colors). But for distant colors (e.g., a saturated green and a medium-brown), there are many possibilities, and no one of them is right for all circumstances.

You can even show that certain reasonable assumptions about color interpolation cannot be met by *any* interpolation scheme on any three-dimensional color space.

If, for instance, one insists

- That the color $C(\alpha, C_1, C_2)$, interpreted as "α of the way between colors C_1 and C_2," be a *continuous* function of α, C_1, and C_2, and should be C_1 when $\alpha = 0$ and C_2 when $\alpha = 1$
- That two colors of equal saturation and brightness should be interpolated by intermediate colors of equal saturation
- And that the color interpolated as "α of the way from C_1 to C_1" should always be C_1

then one has a contradiction. If we restrict our attention to colors of saturation 1 and brightness 1, we have a circle, which we'll denote $\mathbf{S^1}$, and which we'll parameterize by hue, ranging from 0 to 1 (so a hue of 0 and a hue of 1 both denote totally saturated red, for example). Restricting the function C to just this circle gives a function from $[0, 1] \times \mathbf{S^1} \times \mathbf{S^1}$ to $\mathbf{S^1}$; the properties above translate to

- $C : [0, 1] \times \mathbf{S^1} \times \mathbf{S^1} \to \mathbf{S^1}$ is continuous
- $C(0, x, y) = x$ for all $x, y \in \mathbf{S^1}$
- $C(1, x, y) = y$ for all $x, y \in \mathbf{S^1}$
- $C(\alpha, x, x) = x$ for all $x \in \mathbf{S^1}$

Now consider the functions

$$p_0 : [0, 1] \to \mathbf{S^1} : t \mapsto C(0, 0, t), \text{and}$$
$$p_1 : [0, 1] \to \mathbf{S^1} : t \mapsto C(1, t, t).$$

The second property tells us that p_0 is a constant, that is, its winding number around the circle is zero. The final property tells us that p_1 wraps once around the circle. But these two loops can be joined together by a family of intermediate curves,

$$p_s : [0, 1] \to \mathbf{S^1} : t \mapsto C(s, st, t), s \in [0, 1]$$

each of which starts and ends at the same place. This is impossible, for $t \mapsto p_s(t)$ is a continuous function of both s and t, and hence the winding number of p_s is a continuous function of s. But a continuous integer-valued function is constant, so the winding number cannot change from 0 to 1 as we vary s.

This is not to say that color interpolation is impossible or wrong; it merely indicates that many seemingly natural constraints cannot be met. It's therefore often best to carefully consider the application domain, and ask what's actually needed: Do you *really* need to be able to interpolate between colors of opposite hue without passing through white? Will the colors you're interpolating between

ever be far apart, or will they generally be close together? Indeed, an interesting experiment to perform when you want to write a color-interpolating program is to create some sample inputs for your program, and try to approximate the outputs that you'd like to get. If *you* can't approximate the output, you're unlikely to be able to write a program that does so.

28.19 Using Color in Computer Graphics

We use color for aesthetics, to establish tone or mood; for realism, to identify associated groups of entities; and for coding of types of interaction. With care, color use is effective in many of these roles. Careless use of color, however, can be disastrous; in one experiment, the introduction of meaningless color to a monochrome interface reduced user performance by about two-thirds [KW79]. Color should be employed conservatively; decorative uses should be subordinate to functional uses so that color cannot be misinterpreted as having functional meaning. The use of color, like any other aspect of a user interface, must be tested with real users to identify and resolve problems. One conservative approach is to design first for a monochrome display, which ensures that the color use is purely redundant (and guarantees usability by color-deficient users).

There are many books about the use of color for aesthetics, including [Bir61]; we state here just a few of the simpler rules to produce color harmony. The most fundamental rule is to select colors according to some method, typically by traversing a smooth path in some color model, or by restricting colors to a plane in some color space. This might mean using colors of constant saturation or value, for instance. Furthermore, it's wise to choose colors at equal perceptual spacing, which is not the same as equal coordinate distance in whatever color model we're using: Conversion to CIE $L^*u^*v^*$coordinates, or some other system in which perceptual distances are accurate, is essential.

A random selection of hues and saturations is usually quite garish; grouping colors so that those of similar hue or similar saturation are nearby is more attractive (but the distinction between colors may be less obvious).

If a chart or table contains just a few colors, the complement of one of them makes a good choice for the background; a neutral gray is a good background for photographic or similar imagery. If adjoining colors are not harmonious, a thin black border between them will often help resolve the contrast. In general, a parsimonious approach to choosing a color palette is wise (except in the case of realistic images, of course).

Color can be used to code data (indeed, this is a standard tool in scientific visualization applications), but several cautions are in order. First, color codes can carry unintended meanings. If we display the earnings of company A in red and those of company B in green, we may suggest to the viewer that company A is in financial trouble because of our learned association of "red" with "in debt" in financial situations. Bright, saturated colors stand out more than dim, pastel colors; this may give unintended emphasis. And two unrelated elements of an interface that have similar colors may be perceived as related, even if the color was intended as purely ornamental.

A number of color-usage rules are based on physiological rather than aesthetic considerations. For example, because the eye is more sensitive to spatial variation in intensity than it is to variation in chromaticity, lines, text, and other fine detail

should vary from the background not just in chromaticity, but in brightness (perceived intensity) as well—especially for colors containing blue, since relatively few cones are sensitive to blue. Thus, the edge between two equal-brightness colored areas that differ only in the amount of blue will be perceived as fuzzy.

Blue and black differ very little in brightness, so this is a particularly bad combination. Similarly, yellow on white is hard to distinguish.

The eye cannot distinguish the colors of very small objects, so color coding should not be applied to small objects. Judging the color of objects subtending less than 20 to 40 minutes of arc is error-prone [BCfPRD61, Hae76]; at a typical viewing distance of 24 inches, objects 0.1 inch (i.e., many pixels) tall subtend about this angle. The color of a single pixel on a modern monitor is almost impossible to discern.

The color of a region can affect our perception of its size. Cleveland and McGill discovered that a red square is perceived as larger than a green square of equal size [CM83]. This could well cause the viewer to attach more importance to a red object than to a green one of similar size.

If you stare at a large area of saturated color and then look away, afterimages appear. This effect is disconcerting and distracting, so the use of large areas of saturated colors is unwise.

For a number of reasons, red objects appear closer than do blue ones; therefore, simultaneously using blue to represent foreground objects and red to represent background ones is unwise. The opposite coding is fine (although the use of saturated red text on a saturated blue background is particularly annoying to many people).

With all these perils and pitfalls of color usage, is it surprising that one of our first rules was that you should apply color conservatively?

28.20 Discussion and Further Reading

The workings of the human eye and the human visual system, and they way in which they combine to provide the perception of color, cover many fields; the application of color for aesthetic or persuasive or communicative goals occupies others. For more information on the human eye, presented in a manner particularly suited to computer graphics researchers, see Glassner's book [Gla94]; other useful references are [BS81, Boy79, Gre97, Hun05, Jud75, WS82] and [Poya]. For more background on artistic and aesthetic issues in the use of color for graphics, see [Fro84, Mar82, Mei88, Mur85, MSK04]. For more information on calibration and cross-calibration of displays, see [Cow83, SCB88, Con12, Int03].

28.21 Exercises

Exercise 28.1: You want to interpolate between two similar colors. They're represented in RGB space. You've heard that YIQ space is more naturally related to the human eye, so you convert to YIQ, interpolate there, and convert back. Explain why you get exactly the same result as if you'd interpolated in RGB. For which other color-description systems in this chapter will this turn out to be true, and why?

Exercise 28.2: The chapter claims that if you interpolate colors using whatever color triples they're represented by, *and* if the colors are nearby, then it

won't matter what color system you're using, in the sense that the results will be very similar. Verify this in the case of RGB and $L^*u^*v^*$versions of the colors $(r, g, b) = (0.7, 0.4, 0.3)$ and $(r, g, b) = (0.7 + \epsilon, 0.4 - 2\epsilon, 0.3 - \epsilon)$, by finding the 50-50 mix of the two colors in both RGB and $L^*u^*v^*$and comparing. Do this for $\epsilon = 0.01, 0.05$, and 0.25.

Exercise 28.3: Consider points with $Y = 1$ and chromaticity values that range over the entire CIE diagram. Compute the $L^*u^*v^*$coordinates of these points, and plot them on axes labeled L^*, u^*, and v^*.

Exercise 28.4: For $400 < \lambda < 700$, consider two monospectral lights, with $Y = 1$, one with wavelength λ and one with wavelength $\lambda + 1$; they are separated by 1 nm in "wavelength space." Plot their distance in *XYZ*-space as a function of λ; plot their distance in $L^*u^*v^*$-space as a function of λ. At what wavelength is this latter difference largest? Smallest? Note: You'll need to find a table of the *xy*-coordinates of the monospectral points on the CIE horseshoe.

Exercise 28.5: No three-color display can faithfully reproduce all color percepts. Suppose you wanted to design a three-primary display with the largest possible gamut (measured in terms of area on the chromaticity diagram).
(a) Argue why all three primaries should be on the boundary of the horseshoe.
(b) Find the *xy*-coordinates of the horseshoe boundary and then search for the optimal location for the three primaries.
(c) Approximately what percentage of the area can you cover with three primaries? With four? With five?

Exercise 28.6: Derive Equation 28.29 from Equation 28.26.

Exercise 28.7: (a) Suppose that the sensitivities of the receptors in the eye were not shaped like Gaussian bumps, but were instead triangular, the graph of the red receptor being an equilateral triangle with base between 600 nm, and 700 nm, the green having its base between 500 nm and 600 nm, and the blue having its base between 400 nm and 500 nm (all three equilateral triangles having the same heights). What would the CIE diagram look like? How many primaries would be needed for perfect color reproduction?
(b) Suppose instead that the domains overlapped so that red was defined on $[500, 600]$, green on $[450, 550]$, and blue on $[400, 500]$. What would the chromaticity diagram look like? How many primaries would be needed to faithfully reproduce every color percept?

Exercise 28.8: We said that if you're asked to convert a source that's $18\,\mathrm{W\,m^{-2}\,sr^{-1}}$ to nits, it's impossible. Suppose you were told in addition that it was a blackbody source at a particular temperature. Describe how you *could* compute the corresponding number of nits in this case (given a tabulation of the luminous efficiency function).

Exercise 28.9: Write formulas to convert $L^*a^*b^*$ coordinates for a color back to the *XYZ* triple for the color. You may assume that X_w, Y_w, and Z_w are known.

Exercise 28.10: We claimed above that a colorimeter could be used to measure the *XYZ* values for each of the red, green, and blue primaries of a display. Suppose, though, that the colorimeter only produces the CIE *xy*-values, but you can also measure the luminances Y_r, Y_g, and Y_b of the full-brightness red, green, and blue primaries. Express the *XYZ* coefficients of a color with *RGB* coefficients r, g, and b in terms of the observed *xy*-values and the full-brightness luminosities.

Exercise 28.11: (**Peripheral color perception.**) Stand with one arm pointing outward, and fixate on a point in front of you. Have a friend place a playing card in your outstretched hand so that the card faces toward your head. Move your hand

to gradually bring the card into your field of vision. (Continue fixating straight ahead.) Try to tell whether the card is red or black. Move it so that it's 45° off-axis, and try again. Move it to about 30° and try again. Continue until you are certain of the color, and then confirm that you're correct. (Thanks to Pascal Barla for suggesting this exercise.)

Chapter 29

Light Transport

29.1 Introduction

In this chapter we develop the **rendering equation,** which characterizes the light transport in a scene. We do this first for a scene in which there is no transmissive scattering, only reflection, and then generalize to handle transmission as well. In all but the simplest cases, the rendering equation is impossible to solve exactly. Approximation methods are therefore essential tools. The dominant approximation method involves estimating certain integrals by so-called Monte Carlo integration, that is, randomized integration algorithms, which we discuss in the next chapter. To assist with understanding the convergence properties of such algorithms, we can consider various kinds of light transport, some of which are amenable to study with one technique, some with another. For instance, a sequence of mirror reflections that conveys light from a point source to the eye must be treated rather differently from a sequence of diffuse reflections of illumination from an area light source. In fact, the phenomena produced by various kinds of light-transport paths can also be quite different at a perceptual level; we discuss this briefly, as well.

29.2 Light Transport

With the notion of the radiance field in hand, and that of the bidirectional reflectance distribution function (BRDF), we can now discuss light transport in general. We begin with the case of a scene consisting of empty space and purely reflective objects (i.e., there is nothing that's partly transparent, and light reflects from the surfaces of objects—there's no subsurface scattering to consider). The scattering of light from an object is therefore described by the BRDF, which we'll denote f_{r} (the "r" standing for "reflection"). This special case conveys the main ideas but avoids some complexities. Having developed this first situation, we'll generalize to other kinds of scattering, but with very few important changes.

We'll continue with the assumptions of Chapter 26, that the scattering of light by a material comes in two parts: mirror and Snell-transmissive scattering, which

we'll call **impulses,** and everything else. Impulse scattering is characterized by the idea that radiance along some incoming ray is transformed to radiance along a small number (one or two, typically) of outgoing rays. Such scattering cannot be represented by an integral like the one in the reflectance equation unless we admit the possibility of "delta functions" in the scattering function f_s. Nonetheless, we'll continue to write the transformation from incoming to scattered light in the form of the scattering equation (i.e., as an integral), and will consider, in Section 29.6, the consequences of impulses in f_s after developing the main ideas.

Similarly, although the emitted light in a scene typically comes from physical objects like lamps or the sun, which have nonzero size, it's convenient (and traditional) to allow point lights in a scene as well. These amount to impulses in L^e, and must also be handled specially. These, too, will be discussed in Section 29.6.

We'll be discussing **light,** the flow of photons in a scene, extensively. But we also want to talk about light *sources*, which are informally called lights in expressions like "point light" and "area light." To keep these two notions distinct, we'll use the term **luminaire** to mean a light source throughout this chapter.

To discuss light transport, we need to use quite a lot of notation, which we'll reuse in subsequent chapters. We summarize these symbols in Table 29.1, even though some are given full definitions only later in the chapter.

Table 29.1: Symbols used in light transport and rendering.

Symbol	Meaning
E	The eye point.
P	A surface point in the scene, often the first one encountered by a ray from the eye, but sometimes used generically.
Q, Q_j	A point on the surface of a luminaire or some other source of light arriving at P, such as an illuminated reflective surface.
$\mathcal{M}$	The set of all surfaces in the scene.
$\mathbf{n}_P, \mathbf{n}_Q$	The unit normal vector at P, which we've denoted $\mathbf{n}(P)$ previously, or the same thing for Q; using $\mathbf{n}_P$ slightly reduces the complexity of equations.
ω_i	A ray pointing from P toward some source of light.
ω_o	A ray pointing from P in the direction in which reflected light from ω_i exits, typically toward E.
ω	A generic name for a unit vector, typically based at P.
$L(P, \omega)$	The radiance at a surface point P in direction ω.
$L^e(P, \omega)$	The light emitted at point P in direction ω; zero except when P is a point of a luminaire.
$L^{ref}(P, \omega)$	The light reflected at P in direction ω.
$L^r(P, \omega)$	The light reflected or transmitted (refracted) at P in direction ω. $L = L^e + L^r$.
f_s	The bidirectional scattering distribution function.
f_r	The bidirectional reflectance distribution function.
f_s^∞	The "impulse" part of f_s, corresponding to transmission or mirror reflection.
f_s^0	The finite part of f_s, corresponding to nonmirror reflection.

For mathematical convenience, we're going to consider a scene that is *finite,* in the sense that it's contained within some sufficiently large sphere around the origin; the interior of this sphere we'll assume to be coated with a nonreflective material so that all light hitting it is absorbed. In a real scene, once light "leaves" the scene, we ignore it. But for this chapter, it's very useful to have a ray-casting function that takes a point P and direction $\mathbf{d}$ and returns the first surface point Q along the ray starting at P and going in direction $\mathbf{d}$. If the scene isn't surrounded by the large sphere, then a ray headed "out of the scene" doesn't hit anything, and the ray-casting function's value is not defined. So the large black sphere is completely for mathematical convenience, and it has no impact on the actual transport of light.

To keep the notation simple, we're going to further assume that we're studying a steady-state situation, one in which there is no time dependence: The luminaires have all been illuminated for long enough to allow light to scatter throughout the scene and reach a steady state. Furthermore, we're going to ignore the wavelength dependence, and study just radiance rather than *spectral* radiance.

Thus, our starting point is a collection of surfaces whose union is the set $\mathcal{M}$ of all surface points in the scene (including the large enclosing black sphere). For each point $P \in \mathcal{M}$, we also know $f_r(P, \omega_i, \omega_o)$, the bidirectional reflectance distribution function at P, which describes how much light arriving at P traveling in direction $-\omega_i$ becomes light leaving in direction ω_o (see Chapter 26 for the formal definition). The symbols ω_i and ω_o will be reserved, for this section, to be unit vectors that point in the same half-plane as the normal vector $\mathbf{n}_P$ at P, that is, $\omega_i \cdot \mathbf{n}_P \geq 0$ and $\omega_o \cdot \mathbf{n}_P \geq 0$.

In addition to the scene geometry and reflectance, we assume that we're given the **illumination** in the scene, described by the emitted radiance at every point of every luminaire, in every direction; in other words, we're given a function

$$L^e : \mathcal{M} \times \mathbf{S^2} \to \mathbf{R} : (P, \omega) \mapsto L^e(P, \omega), \tag{29.1}$$

where $L^e(P, \omega)$ denotes the *emitted* radiance leaving the point P in direction ω, the radiance you'd measure if every other luminaire and surface in the scene were removed, so that no light at all *arrived* at the point P, and hence none was reflected.

For a typical point P of a typical area luminaire, $L^e(P, \omega)$ is zero if $\omega \cdot \mathbf{n}_P < 0$; in other words, light radiates only toward the "outward" side of the luminaire. For luminaires like typical white incandescent lightbulbs, the radiance in all such outward directions is the same, or $L^e(P, \omega) = C$ for $\omega \cdot \mathbf{n}_P > 0$. Because of the analogy with the light reflected from a Lambertian surface, we'll refer to such a luminaire as **Lambertian.**

As we said above, we'll also assume we have a *ray-casting function* (see Figure 29.1),

$$R : \mathcal{M} \times \mathbf{S^2} \to \mathcal{M}, \tag{29.2}$$

where $R(P, \omega)$ is the first point hit by a ray starting at P in direction ω. If $\omega \cdot \mathbf{n}_P \leq 0$, then $R(P, \omega) = P$, that is, a ray *into* a shape hits the shape immediately. If $\omega \cdot \mathbf{n}_P > 0$, then $R(P, \omega)$ is the point we see by looking in direction ω from P. More precisely, $R(P, \omega)$ is the farthest point Q on the ray from P in direction ω with the property that all points of the ray strictly between P and Q are in empty space.

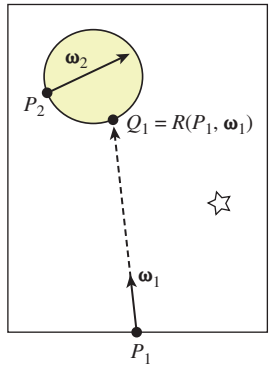


Figure 29.1: $R(P_1, \omega_1) = Q_1$, *but* $R(P_2, \omega_2) = P_2$.

The algorithmic representation of the ray-casting function and the associated **visibility function**—see Exercise 29.1—is the subject of Chapters 36 and 37, but you've already seen basic examples in Chapter 15.

With our assumptions clearly characterized, we can now analyze the light transport in our scene.

29.2.1 The Rendering Equation, First Version

Consider once again the reflectance equation, Equation 26.80, rewritten without time or wavelength dependence:

$$L^{\text{ref}}(P, \omega_o) = \int_{\omega_i \in \mathbf{S}^2_+(p)} L(P, -\omega_i) f_r(P, \omega_i, \omega_o)(\omega_i \cdot \mathbf{n}_P)\, d\omega_i. \tag{29.3}$$

This expresses the radiance reflected at point P in *outward* direction ω_o in terms of the light arriving at P in other directions.

If P happens to be a point of a luminaire, light may also leave P in direction ω_o because it is *emitted* at P rather than because it is reflected from there; that is,

$$L(P, \omega_o) = L^e(P, \omega_o) + L^{\text{ref}}(P, \omega_o) \tag{29.4}$$

$$= L^e(P, \omega_o) + \int_{\omega_i \in \mathbf{S}^2_+(p)} L(P, -\omega_i) f_r(P, \omega_i, \omega_o)(\omega_i \cdot \mathbf{n}_P)\, d\omega_i. \tag{29.5}$$

This is a basic version (e.g., it handles only reflection) of the **rendering equation,** which characterizes the function L, given the functions L^e and f_r. It was first described in computer graphics by Kajiya [Kaj86] and Immel et al. [ICG86], in slightly differing forms. It is completely analogous to similar equations developed in subjects like radiative transfer. While we'll concentrate on Kajiya's description and derivation in subsequent chapters, the form presented by Immel et al. is particularly well suited to the "sampling" needed in Monte Carlo rendering.

You'll notice that the unknown radiance function L appears on both sides of the equation, once under an integral, just as the unknown function h appears on both sides of the differential equation

$$h'(x) = 2h(x - 1), \tag{29.6}$$

once within a derivative. Equation 29.4 is called an **integral equation,** and solving such an equation is generally more difficult than solving a differential equation. The next chapter discusses various approaches to finding approximate solutions.

Equation 29.4 expresses the radiance function L, considering both the radiance *leaving* the point P and the radiance arriving there. Arvo [Arv95] calls the first of these the **surface radiance** and the second the **field radiance.** The rendering equation tells us how to compute *surface* radiance from field radiance, because it's restricted to the case where $\omega_o \cdot \mathbf{n}_P > 0$. But to evaluate the right-hand side, we must know how to compute the *field* radiance as well.

The idea that "closes the loop" in this equation is that any light arriving at a point P, traveling in direction $-\omega_i$, must have *departed* from some other point $Q \in \mathcal{M}$ traveling in direction $-\omega_i$. The point Q must be the point visible from P in direction ω_i. These observations allow us to write the **transport equation:**

$$L(P, -\omega_i) = L(R(P, \omega_i), -\omega_i), \tag{29.7}$$

for any ω_i with $\omega_i \cdot \mathbf{n}_P > 0$.

Substituting Equation 29.7 into Equation 29.4, we get the form of the rendering equation that's most useful in practice:

$$L(P, \omega_o) = L^e(P, \omega_o) + L^{ref}(P, \omega_o) \tag{29.8}$$

$$= L^e(P, \omega_o) + \int_{\omega_i \in \mathbf{S}^2_+(p)} L(R(P, \omega_i), -\omega_i) f_r(P, \omega_i, \omega_o)(\omega_i \cdot \mathbf{n}_P)\, d\omega_i. \tag{29.9}$$

This equation expresses the **surface radiance** function defined on all surfaces in the scene, in terms of the known luminaires (L^e), and an integral of the known BRDFs of all surface points (f_r), the ray-casting function R, and the surface radiance itself.

29.3 A Peek Ahead

The rendering equation is very nice and self-contained, but how do you *do* anything with it? Let's take a quick look ahead at code from Chapter 32 to see. The large-scale structure of a basic path tracer is:

```
foreach pixel (i, j)
  C = location of pixel on image plane
  r = ray from eye to C
  image[i, j] = pathTrace(r, true)
```

Listing 29.1 shows the central `pathTrace` procedure for such a path tracer. Given a ray (i.e., a point U and a direction ω) this procedure traces a ray into the scene and hits at some point P, and then estimates either $L(P, -\omega)$ or $L^{ref}(P, -\omega)$, depending on the boolean `isEyeRay`. The point P is represented by the variable `surfel` (for "surface element") in the program.

Listing 29.1: The core procedure in a path tracer.

```
Radiance3 App::pathTrace(const Ray& ray, bool isEyeRay) {
    Radiance3 radiance = Radiance3::zero();
    SurfaceElement surfel;

    float dist = inf();
    if (m_world->intersect(ray, dist, surfel)) {
        if (isEyeRay)
            radiance += surfel.material.emit;

        radiance+= estimateDirectLightFromPointLights(surfel, ray);
        radiance+= estimateDirectLightFromAreaLights(surfel, ray);
        radiance+= estimateIndirectLight(surfel, ray, isEyeRay);
    }
    return radiance;
}
```

As you can see, the outgoing radiance at P is the sum of the emitted light (`surfel.material.emit`) and the light reflected at P, estimated in the last three procedures. The first estimates the light arriving directly from point sources that's reflected at P; the second the light directly from area luminaires that's reflected

at P; and the third all other light. Thus, the term $L(P, -\omega_i)$ in the rendering equation has been split into a sum of three terms.

The first of these terms is computed (see Listing 29.2) by converting the integral over all possible incoming directions into a sum over the point luminaire sources. This change of domain in the integral means we have to alter the integrand as well, using the change of variable from Section 26.6.5.

Listing 29.2: Reflecting illumination from point lights.

```
Radiance3 App::estimateDirectLightFromPointLights(surfel, ray){
    Radiance3 radiance(0.0f);
    for (int L = 0; L < m_world->lightArray.size(); ++L) {
        const GLight& light = m_world->lightArray[L];
        // Shadow rays
        boolean visible = m_world->lineOfSight(surfel.geometric.location +
                                               surfel.geometric.normal * 0.0001f,
                                               light.position.xyz())
        if (visible){
             Vector3 w_i = light.position.xyz() - surfel.shading.location;
             const float distance2 = w_i.squaredLength();
             w_i /= sqrt(distance2);
             // Attenuated radiance
             const Irradiance3& E_i = light.color / (4.0f * pif() * distance2);

             radiance += (surfel.evaluateBRDF(w_i, -ray.direction()) *
                          E_i * max(0.0f, w_i.dot(surfel.shading.normal)));
        }
    }
    return radiance;
}
```

As you can see, the procedure loops through all the point luminaire sources, and for each one, it checks whether the source is *visible* from P, using `m_world->lineOfSight()`, the implementation of the visibility function for the world we're rendering. The reflected radiance is computed as a product of the BRDF, the dot product $\omega_i \cdot \mathbf{n}_P$, and a term E_i, which is the incoming radiance adjusted by the change-of-variable factor mentioned above.

The second term is similar, involving estimates of how area luminaires are reflected. The third term is the most interesting, however. Before we look at it, we need one more bit of mathematics (which we'll develop extensively in Chapter 30).

The idea is this: The integral of any function h over any domain D is the product of the *average value* of h on that domain with the *size* of the domain. When the domain is the interval $[a, b]$, the size is $b - a$; when the domain is a unit hemisphere, the size is 2π, etc. The "average value" can be estimated by evaluating h at n points of the domain and averaging. As n gets large, the estimate gets better and better. But even $n = 1$ works! That is to say, we can estimate the integral of h over the domain D by evaluating $h(x)$ for a randomly chosen point $x \in D$ and multiplying by the size of D. The estimate won't generally be very good, but if we repeat the estimation procedure multiple times, the *average* will be quite a good estimate.

We'll apply this to the situation where h is the integrand in the reflectance equation, and D is the upper hemisphere. Let's look at the code (see Listing 29.3).

Listing 29.3: Estimating the indirect light scattered back along a ray.

```
Radiance3 App::estimateIndirectLight(surfel, ray, bool isEyeRay){
    Radiance3 radiance(0.0f);
    // Use recursion to estimate light running back along ray from surfel that arrives from
    // INDIRECT sources, by making a single-sample estimate of the arriving light.

    Vector3 w_o = -ray.direction();
    Vector3 w_i;
    Color3 coeff;

    if (surfel.scatter(w_o, w_i, coeff)) {

        newRay = Ray(surfel.geometric.location, w_i).bumpedRay(
           0.0001f * sign(surfel.geometric.normal.dot(w_i));
        // the final "false" makes sure that we do not include direct light.
        radiance = coeff * pathTrace(newRay, surfel.geometric.normal), false);
    }
    return radiance;
}
```

Without worrying too much about the details, what's happening here is that we're picking a random direction `w_i`, using `scatter`, on the outgoing hemisphere. We're then using `PathTrace` to estimate the radiance arriving at P from that direction, that is, we're estimating $L(P, -\omega_i)$. The coefficient by which we multiply this radiance includes the area of the hemisphere and an adjustment for the fact that we did not pick our direction uniformly from all possible directions, but instead biased our choice based on the BRDF, for reasons you'll learn about in Chapter 30.

To summarize: The recursive nature of the rendering equation is exactly reflected in the recursive nature of the program. You might reasonably ask whether the recursion will ever terminate, since there seems to be no stopping condition. The answer is yes, because of the design of the `scatter` procedure: If a surface's hemispherical reflectance is 0.7, then 30% of the time `scatter` will return `false` and the recursion will terminate. The other 70% of the time the coefficient `coeff` is adjusted to take into account the probability of nonscattering.

29.4 The Rendering Equation for General Scattering

In formulating the rendering equation, we assumed that our scene contained only reflective materials rather than ones that could transmit light, or participating media like fog that can scatter light as it passes through them.

We'll now generalize to handle transmissive materials as well. This will require almost no new concepts, but we'll need to slightly revise the way we represent the radiance in a scene. A further revision is needed to handle participating media, which we will not discuss here. Instead, we'll consider just one special case, in which light is attenuated by absorption as it passes through a medium but is never scattered in any new direction. More general models of scattering, discussed briefly in Chapter 27, can be used to generate quite striking renderings like that shown in Figure 29.2 from 1987, one of the earliest high-quality synthetic images of participating media.

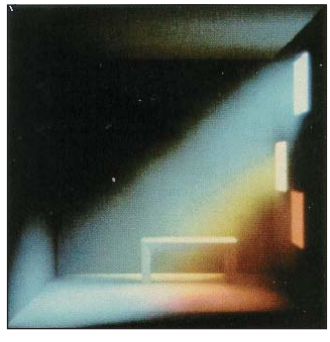


Figure 29.2: Rendering with a participating medium (dusty air). (Courtesy of Holly Rushmeier, ©1987 ACM, Inc. Reprinted by permission.)

The critical problem, when we want to include transparency into the rendering equation, is that a pair $(P, \omega) \in \mathcal{M} \times \mathbf{S^2}$ is no longer associated to a unique radiance value.

Because we are only doing surface rendering rather than volume rendering (i.e., we're ignoring participating media), we only care about $L(P, \omega)$ for points P that are on some surface. That surface must have a surface normal $\mathbf{n}$ at P. We can therefore define $L(P, \omega, \mathbf{n})$ by the rule that if ω and $\mathbf{n}$ have a positive dot product, then $L(P, \omega, \mathbf{n})$ represents the radiance leaving the surface in direction ω; if their dot product is negative, then it represents the radiance *arriving* at the surface in direction ω. By using either the unit outward surface normal *or its opposite* as $\mathbf{n}$, we can handle both reflection and transmission. In practice, this turns into an additional `if` statement at every ray-surface interaction: A surface element has two **sides** (one with a normal pointing each way), and we treat the vector ω differently depending on whether it points in the same or the opposite half-space as the normal vector to the "side." In mathematical terms, we can say that L is defined on the orientation double cover [Lee09] of the set of all scene surfaces.

The three-argument version of L is awkward to write. As an alternative, we can replace L altogether with two new functions: $(P, \omega) \mapsto L^{\text{in}}(P, \omega)$ and $(P, \omega) \mapsto L^{\text{out}}(P, \omega)$, which represent light arriving at P traveling in direction ω and light leaving P in direction ω, respectively; these are Arvo's field and surface radiance functions. The reflectance equation then becomes a **scattering equation:**

$$L^{\text{r, out}}(P, \omega_{\text{o}}) = \int_{\omega_{\text{i}} \in \mathbf{S^2}(p)} L^{\text{in}}(P, -\omega_{\text{i}}) f_{\text{s}}(P, \omega_{\text{i}}, \omega_{\text{o}}) |\omega_{\text{i}} \cdot \mathbf{n}_P| \, d\omega_{\text{i}}, \tag{29.10}$$

where five things have changed.

- The integral is now over *all* directions of incoming light.
- The result is now L^{r} rather than L^{ref} (recall that L^{r} denotes light either reflected *or* transmitted).
- The BRDF f_{r} has been replaced with the BSDF f_{s}.
- The dot product now has an absolute value.
- The annotations "in" and "out" have been added to the radiance and scattered radiance.

The transport equation now links incoming and outgoing light. We write the equation in two forms, one suitable for use in ray tracing, the other for use in photon mapping. The distinction is simply one of tracing rays in the direction of photon propagation or in the opposite direction. The version used in raytracing is this:

$$L^{\text{in}}(P, -\omega_{\text{i}}) = L^{\text{out}}(R(P, \omega_{\text{i}}), -\omega_{\text{i}}) \tag{29.11}$$

while the one that's useful in photon mapping is this:

$$L^{\text{out}}(Q, \omega_{\text{o}}) = L^{\text{in}}(R(Q, \omega_{\text{o}}), \omega_{\text{o}}) \tag{29.12}$$

Dividing the radiance field into two parts has further advantages. When we write the radiance field this way, it naturally extends to all points P rather than

limiting us to those points that lie on surfaces—we define radiance at (P, ω) for a nonsurface point P by letting $Q = R(P, -\omega)$, and setting

$$L(P, \omega) = L^{\text{out}}(Q, \omega), \tag{29.13}$$

which results in radiance that's constant along rays in empty space. In Equation 29.13, we defined $L(P, \omega)$ rather than L^{in} or L^{out} because at points of empty space, these two functions agree; they only differ at points of $\mathcal{M}$.

The rendering equation now becomes

$$L^{\text{out}}(P, \omega_{\text{o}}) = L^{\text{e}}(P, \omega_{\text{o}}) + \int_{\omega_{\text{i}} \in \mathbf{S}^2(p)} L^{\text{in}}(P, -\omega_{\text{i}}) f_{\text{s}}(P, \omega_{\text{i}}, \omega_{\text{o}}) |\omega_{\text{i}} \cdot \mathbf{n}_P| \, d\omega_{\text{i}}. \tag{29.14}$$

The changes we've made to incorporate transmission seem fussy and likely to lead to code with multiple cases. In practice, however, they have almost no effect. That's partly because of the restricted model of scattering we use in representing materials in Chapter 32: Scattering at a surface point consists of a small number of impulses and an otherwise diffuse or glossy reflectance-scattering pattern. (Recall that an impulse is a phenomenon that is similar to mirror reflection or Snell-Fresnel refraction, where radiance arriving along one ray scatters out along just one or two other rays.) In particular, in the general rendering equation, the part of the integral representing transmission degenerates to something far simpler: We look at the radiance arriving along one particular ray, multiply it by a constant representing how much light is transmitted, and add the result to the outgoing radiance.

29.4.1 The Measurement Equation

Typically a renderer takes a scene description as input, and produces an image—a rectangular array of values—as output. These values might just be RGB triples in some fixed range, or they might be RGB radiance values representing radiance in W m^{-2} sr^{-1}, or something else. In general, a particular pixel value represents the result of a *measurement* process. For a typical digital camera, the red measurement, for one pixel, represents the total charge accumulated in one cell of a CCD device. For a synthetic camera, it might represent the integral of irradiance in the red portion of the spectrum over the rectangular corresponding to one pixel on the image plane. Or it might represent a weighted integral of this irradiance over a disk slightly *larger* than the rectangle usually associated to a pixel so that radiance along a single ray contributes to the value of more than one pixel in the final image. We express this idea by associating to each pixel ij a **sensor response** M_{ij}, which converts radiance along any ray into a numerical value that can be summed over all rays to get the sensor value. That is to say, we posit that the measurement m_{ij} associated to pixel ij is computed as

$$m_{ij} = \int_{U \times \mathbf{S}^2} M_{ij}(P, \omega) L^{\text{in}}(P, -\omega) |\omega \cdot \mathbf{n}_P| \, dP \, d\omega, \tag{29.15}$$

where U is the image plane. This is a purely formal description of the measurement process. The critical thing is that M_{ij} is zero except for points in a small area and directions in a small solid angle. For a camera with a small pinhole aperture, for instance, $M_{ij}(P, \omega)$ is nonzero only if both of the following are true.

- The ray $t \mapsto P + t\omega$ passes through the pinhole.
- P is within the part of the image plane associated to pixel ij.

In this case, the radiance $L^{\text{in}}(P, -\omega)$ arriving at P is multiplied by the sensor response associated to this ray.

For a real physical sensor, the sensor response is called **flux responsivity** and has units of W^{-1}. Since the radiance is in $\text{W m}^{-2}\ \text{sr}^{-1}$, and we integrate out square meters and steradians, this makes the measurement m_{ij} unitless.

For a typical sensor, M_{ij} has a form that's independent of the pixel ij. For example, it might have the constant value 1 on a small rectangle representing the pixel, and zero elsewhere. While the region on which it takes the value 1 changes with ij, the form of the function doesn't change.

When we consider the space of all paths along which light might travel in a scene, from the point of view of rendering, some are more important than others. In particular, the paths that end up entering the virtual camera matter more to us than do those that end up absorbed in some distant and invisible part of the scene. Thus, the function M_{ij} can help us decide which paths might be worth examining. Because of this, it's been called the **importance function** [Vea96].

29.5 Scattering, Revisited

In the introduction, we talked (broadly) about two types of scattering. The first is mirrorlike: Radiance arriving from some direction ω_i leaves in a single direction ω_o, perhaps after attenuation by some factor $0 \leq c \leq 1$. The two main examples of mirrorlike reflectance are (a) mirrors, and (b) Snell's-law refraction. The second type of scattering is diffuselike scattering, in which radiance arriving in some direction is scattered over a whole solid angle of directions (perhaps uniformly with respect to angle—the Lambertian case—or perhaps nonuniformly). In this second kind of scattering, the outgoing radiance in a direction ω_o due to radiance along a single ray in direction $-\omega_\text{i}$ is infinitesimal: To get a nonzero outgoing radiance, we must sum radiance scattered from a whole solid angle of incoming directions; the integral in the rendering equation expresses exactly this. For the integral formulation to apply to the *first* part requires the fiction of "infinite values" for f_s akin to delta functions.

We can treat the process of converting incoming radiance to outgoing radiance as an operation K that takes the incoming radiance and scattering functions as input and produces the outgoing radiance $L^{\text{out}} = K(L^{\text{in}}, f_\text{s})$. What we've said in the previous paragraph is that f_s should be written as a sum $f_\text{s}^0 + f_\text{s}^\infty$, the first being the "finite part" and the second being the "impulse part"; the rule for combining the field radiance with the finite part to get the surface radiance can then be legitimately expressed as an integral:

$$K(L^{\text{in}}, f_\text{s}^0)(P, \omega_\text{o}) = \int_{\mathbf{S}^2} f_\text{s}^0(P, \omega_\text{i}, \omega_\text{o}) L^{\text{in}}(P, -\omega_\text{i})\ |\omega_\text{i} \cdot \mathbf{n}|\ d\omega_\text{i}. \tag{29.16}$$

In the opaque (i.e., reflection-only) case, the integral would be over a hemisphere, and we'd write f_r instead of f_s.

The rule for combining the incoming radiance with the impulse part has the form

$$K(L^{\text{in}}, f_\text{s}^\infty)(P, \omega_\text{o}) = \sum_{\omega_\text{i} \in H(\omega_\text{o})} f_\text{s}^\infty(\omega_\text{i}, \omega_\text{o}) L^{\text{in}}(P, -\omega_\text{i}), \tag{29.17}$$

where $H(\omega_o)$ is the (finite) set of directions ω_i that undergo mirrorlike scattering at P to direction ω_o, and $f_s^\infty(\omega_i, \omega_o)$ denotes the constant by which incoming radiance is scaled to produce outgoing radiance, which we've previously called the **magnitude of the impulse.**

There's a slight subtlety here. The form given in Equation 29.17 is only valid if L^{in} is continuous at $(P, -\omega_i)$. Otherwise, the value must be determined by a limit. As far as we know, this detail is generally ignored in graphics, perhaps because almost all physical processes involve convolution, and convolution tends to produce continuous functions. That is to say, perhaps our pure mathematically modeled L^{in} has a discontinuity, but if we were in the real world, things like volumetric scattering would tend to make it actually continuous. Any picture whose appearance *depended* on the discontinuity of L^{in} would be nonphysical anyhow!

29.6 A Worked Example

Consider the situation in Figure 29.3. The surface is 50% Lambertian and 30% a mirror reflector (so 20% of arriving light is absorbed). We'll compute the light reflected from the point $P = (0, 0, 0)$ under two different lighting conditions.

1. The surface is bathed in light from all points of the postive-x hemisphere (see Figure 29.4). The radiance $L^{in}(P, -\omega_i)$ is 6 W m^{-2} sr^{-1} for all ω_i with $\omega_i \cdot \begin{bmatrix}1 & 0 & 0\end{bmatrix}^T \geq 0$.
2. The surface is illuminated by a uniformly radiating sphere (see Figure 29.5) of radius $r < 1$ at position $Q = (1, 1, 0)$; the total power of the luminaire is 10 W.

We'll examine the behavior of the second case as $r \to 0$ as well.

In each case, we'll compute the reflected radiance in the direction $\omega_o = \mathcal{S}(\begin{bmatrix}-1 & 1 & 0\end{bmatrix}^T)$.

We start with situation 1. Let's begin by computing the diffusely reflected light. This is

$$L^{ref,\,0}(P, \omega_o) = \int_{\mathbf{S}^2_+(P)} f_s^0(P, \omega_o, \omega_i) L(P, -\omega_i)(\omega_i \cdot \mathbf{n}_P)\, d\omega_i. \tag{29.18}$$

Rewriting ω_i in polar coordinates, $(x, y, z) = (\cos\theta \sin\phi, \cos\phi, \sin\theta \sin\phi)$, the radiance field $L(P, -\omega_i)$ is zero unless $x > 0$, that is, $\cos\theta > 0$, so we can restrict to $-\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}$. Similarly, since we're only considering reflectance, we can restrict to $0 \leq \phi \leq \frac{\pi}{2}$. Thus, our integral becomes

$$L^{ref,\,0}(P, \omega_o) = \int_{-\pi/2}^{\pi/2} \int_0^{\pi/2} f_s^0(P, \omega_o, \omega_i) L(P, -\omega_i)(\omega_i \cdot \mathbf{n}_P) \sin\phi\, d\phi\, d\theta, \tag{29.19}$$

where the factor of $\sin\phi$ comes from the change to polar coordinates. Within this restricted domain, the value of L is the constant 6, so the integral becomes

$$L^{ref,\,0}(P, \omega_o) = \int_{-\pi/2}^{\pi/2} \int_0^{\pi/2} f_s^0(P, \omega_o, \omega_i) 6 \cos\phi \sin\phi\, d\phi\, d\theta, \tag{29.20}$$

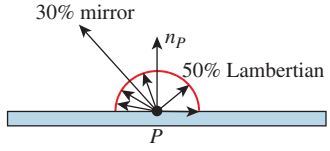


Figure 29.3: Scattering example: a 50% matte reflector that's also 30% mirror-reflective, but not transmissive.

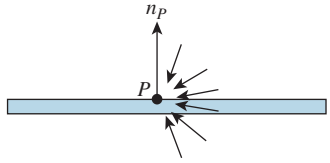


Figure 29.4: Light arrives from everywhere in the right half-space.

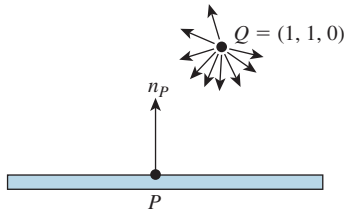


Figure 29.5: The point P is illuminated by a tiny, uniformly emitting, spherical lamp at location Q.

where we've replaced the dot product with $\cos\phi$. Finally, the finite part of the BSDF is the constant $\frac{0.5}{\pi}$ sr^{-1}, so the reflected radiance becomes

$$L^{\text{ref, 0}}(P, \omega_o) = \int_{-\pi/2}^{\pi/2} \int_0^{\pi/2} \frac{0.5}{\pi} 6 \cos\phi \sin\phi \, d\phi \, d\theta \tag{29.21}$$

$$= \frac{3}{\pi} \int_{-\pi/2}^{\pi/2} \int_0^{\pi/2} \cos\phi \sin\phi \, d\phi \, d\theta \tag{29.22}$$

$$= \pi \frac{3}{\pi} \int_0^{\pi/2} \cos\phi \sin\phi \, d\phi \tag{29.23}$$

$$= \frac{3}{2}, \text{ so that} \tag{29.24}$$

$$L^{\text{ref}}(P, \omega_o) = \frac{3}{2} + I \text{ W m}^{-2} \text{ sr}^{-1}, \tag{29.25}$$

where I is the impulsively reflected (i.e., mirror-reflected) radiance. Since the surface is 30% mirror-reflective, the incoming radiance of 6 W m^{-2} sr^{-1} is multiplied by the magnitude 0.3 to get the outgoing mirror-reflected radiance, 1.8 W m^{-2} sr^{-1}. Thus, the total reflected radiance is $\frac{3}{2} + 1.8$ W m^{-2} sr^{-1} = 3.3 W m^{-2} sr^{-1}.

As a result, we've converted the handling of an impulse in the scattering function from an integral to a simple multiplication by a constant, the impulse magnitude.

Now, as we look at the second situation, with illumination provided by a 10 W radiating small sphere, we'll see how each term (the diffuse and the impulse) behaves when there's an "impulse" in the incoming light field (i.e., a point light), by seeing what happens as the radius of the sphere approaches zero.

As we showed in Section 26.7.3, a uniformly radiating sphere of radius r and total power Φ produces radiance $\frac{\Phi}{4\pi(\pi r^2)}$ along every outgoing ray, and subtends a solid angle approximately $\frac{\pi r^2}{R^2}$ at a point at distance R, with the approximation growing better and better as R increases or r decreases.

The integral to compute the Lambertian-reflected light from this small spherical source is essentially the same as the one above, except that instead of integrating over all directions $\omega_i = \begin{bmatrix} x & y & z \end{bmatrix}^{\mathbf{T}}$ with $x \geq 0$, we now must integrate over just the small solid angle Ω subtended at P by the small spherical source. So

$$L^r(P, \omega_o) = \int_\Omega f_s^0(P, \omega_o, \omega_i) L(P, -\omega_i)(\omega_i \cdot \mathbf{n}(P)) \, d\omega_i + I, \tag{29.26}$$

where I as before represents the impulse-reflected radiance and f_s^0 again is the constant function $0.5/\pi$ sr^{-1}. Furthermore, for ω_i in the solid angle subtended by the luminaire, $\omega_i \cdot \mathbf{n}(P)$ is well approximated by $\mathbf{u} \cdot \mathbf{n}(P)$, where $\mathbf{u}$ is the unit vector from P to the center Q of the radiating sphere, that is, $\frac{\sqrt{2}}{2} \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}^{\mathbf{T}}$. Since the normal $\mathbf{n}(P)$ points in the y-direction, this dot product is just $\sqrt{2}/2$. Hence,

$$L^r(P, \omega_o) \approx \frac{\sqrt{2}}{2} \frac{0.5}{4\pi} \int_\Omega L(P, -\omega_i) \, d\omega_i + I. \tag{29.27}$$

The radiance along each ray is the constant $\frac{\Phi}{4\pi(\pi r^2)}$, so this becomes

$$L^r(P,\omega_o) \approx \frac{\sqrt{2}}{4\pi}\frac{\Phi}{4\pi(\pi r^2)}\int_\Omega d\omega_i + I \tag{29.28}$$

$$= \frac{\sqrt{2}}{4\pi}\frac{\Phi}{4\pi(\pi r^2)}\frac{\pi r^2}{R^2} + I \tag{29.29}$$

$$= \frac{\sqrt{2}}{4\pi}\frac{\Phi}{4\pi R^2} + I. \tag{29.30}$$

Since $R = \sqrt{2}$ and $\Phi = 10$, we get that the reflected radiance is $\frac{5\sqrt{2}}{16\pi^2} + I$ W m^{-2} sr^{-1}. Notice that this approximation of Equation 29.28 is independent of r so that even as the spherical luminaire shrinks to a point, the reflected radiance remains the same.

The constancy of the reflected radiance depends on two things: the assumption that the dot product $\omega_i \cdot \mathbf{n}(P)$ can be approximated by $\mathbf{u} \cdot \mathbf{n}(P)$, and the fact that the finite portion of the scattering function is constant. The first of these is justified because we're letting r approach zero. The second is *not* true for a general BSDF. But if f_s^0 is *continuous,* as it is in all the BSDFs that we consider, then the mean value theorem for integrals tells us that the integral we wish to compute is equal to

$$m(\Omega) \cdot f_s^0(P,\omega_o,\omega_i^*)L(P,-\omega_i^*)(\omega_i^* \cdot \mathbf{n}(P)) \tag{29.31}$$

for some ω_i^* in Ω. As the area of Ω goes to zero, this vector ω_i* must approach $\mathbf{u}$, so the integral approaches

$$\pi(r/R)^2 f_s^0(P,\omega_o,\mathbf{u})L(P,-\mathbf{u})(\mathbf{u} \cdot \mathbf{n}(P)). \tag{29.32}$$

To summarize the preceding argument, a point luminaire of power Φ, in direction ω_i from P, at distance R, produces reflected radiance (from the nonimpulse portion of scattering) in direction ω_o in the amount

$$\frac{\Phi}{4\pi R^2} f_s^0(P,\omega_i,\omega_o)\omega_i \cdot \mathbf{n}_P, \qquad \text{for } \omega_i \cdot \mathbf{n}_P > 0 \text{ and } \omega_o \cdot \mathbf{n}_P > 0. \tag{29.33}$$

Finally, let's consider the impulse reflection of the very small spherical source. The radiance leaving each point of the luminaire is again $\frac{\Phi}{4\pi(\pi r^2)}$. Because ω_i points to a point of the luminaire, the radiance arriving in direction $-\omega_i$ is $\frac{\Phi}{4\pi(\pi r^2)}$; to get the outgoing radiance in the mirror-reflected direction ω_o, we multiply by the impulse value 0.3 so that the outgoing radiance due to mirror reflection of the very small spherical source is $0.3\frac{\Phi}{4\pi(\pi r^2)}$. *Notice that this depends on the value of r!* As the size of the luminaire decreases, the radiance emitted must increase to keep the total power the same, with the result that the mirror-reflected radiance also increases without bound. If we try to take a limit, we end up with an answer that includes ∞, which is unsatisfactory.

There are several possible ways to address this problem.

1. Assert that no eye ray that we trace will ever "just happen" to hit a point luminaire, so this is a probability-zero event, and we can ignore the infinity that would arise if this event happened.
2. Say that for the sake of reflecting from diffuse surfaces, point lights are points, but for the sake of specular reflections, they have a nonzero radius r, which must be chosen by the user. Note that this makes the world in which we're trying to simulate light transport internally contradictory.

3. Say that point lights are in fact small spheres of a known radius, r, but that we'll restrict our scene to be sure that the distance from a point luminaire to any surface is much greater than r so that the diffusely reflected light can be very well approximated by treating the sources as point sources, but mirror-reflected light must be computed using r.
4. Observe that when we include both point lights and mirror reflections, both of which are approximations of physical phenomena made for convenience, the mathematics becomes intractable, and hence we'll abandon one or both.

Each of these approaches has its merits, although we prefer approach 3 to approach 2 (even though they both may result in identical programs). In Chapter 32, we choose the first option: We simply ignore mirror-reflected point lights.

29.7 Solving the Rendering Equation

It's natural to ask, having derived the rendering equation, how to solve it. That is to say, if we know the scene geometry and materials and illumination, how can we compute $L(P, \omega)$ for any point P and any vector ω, or for every P and every ω? We've already given you a taste of the answer in discussing a path tracer. But the general topic is the subject of the next three chapters. Because integration is central to the rendering equation, the first discusses probability and Monte Carlo integration. The second describes the ideas behind several techniques for solving the rendering equation. The third gives an implementation of two renderers. One of the shocking things is how very short the two programs are, given the length of this chapter and the next three. That brevity is partly due to the use of libraries for things like visibility testing, basic linear algebra, and material representation. But as you'll see, it's also due to the simplicity of basic Monte Carlo integration.

29.8 The Classification of Light-Transport Paths

In the course of the Monte Carlo integration used to estimate solutions to the rendering equation, we'll break up the integrals of the rendering equation into different parts—sometimes by breaking up the domain of integration into the categories of "directions in which we see luminaires" and "other directions," and sometimes by breaking up the integrand into a sum of a finite part and an impulse part, usually as a means of distinguishing between things like point luminaire and area luminaires, or between mirror reflections and diffuse reflections.

Because of this distinction in treatment, it's useful to be able to discuss the path that light took in getting from the luminaire to the receiver (a **light path**), or its reverse, the path we traced from the eye to eventually reach the light source (an **eye path**). Heckbert [Hec90] used a notation that's now universally accepted:[1] L denotes a luminaire, E the "eye," S a specular reflection, and D a diffuse reflection. Thus, LDE represents light that left the source, scattered from a diffuse surface,

1. Hanrahan [JAF+01] attributes this notation to Shirley, who claims [Shi10] that he's uncertain who first used it.

and reached the eye. Conventional regular-expression notation is useful here, with $+$ used to indicate "one or more," $*$ for "zero or more," ? for "zero or one," parentheses for grouping, and | for "or," so that $L(D|S)E$ denotes a one-bounce path from a source to the eye that involves either a diffuse or specular reflection, while LD^+E is a path involving one or more diffuse bounces.

Inline Exercise 29.1: (a) Write the notation for light that undergoes one or more diffuse bounces, and then a final specular bounce before reaching the eye.
(b) A very basic ray tracer might consider rays from the eye that undergo only repeated specular bounces, followed by zero or one diffuse bounce, before reaching the light. Write the notation for the path traveled by the light, from luminaire to eye.

Note that the notation describes a sequence of scattering events, *not* just the path. In the case of the half-mirror surface we just discussed, light could travel from a luminaire to the surface, be mirror-reflected, and reach the eye; it could also travel from the source, be *diffusely* scattered, and reach the eye. The light energy in the two cases travels along the same path, but the first is described by *LSE* while the second is described by *LDE*.

There are extensions that are fairly common. Veach uses D to mean Lambertian, G for any kind of glossy reflection, S for perfectly specular, and T for transmission, for instance [Vea97].

We can use this notation to characterize certain rendering algorithms in terms of the eye paths that they consider, following Hanrahan [JAF+01]:

- Appel's ray-casting algorithm: $E(D|G)L$
- Recursive ray tracing (Whitted): $E[S^*](D|G)L$
- Path tracing (Kajiya): $E[(D|G|S)^+)D|G)]L$
- Radiosity: ED^*L

Recursive ray tracing considers only light that's diffusely or glossily reflected, and then mirror-reflected to the eye. Radiosity only handles diffuse reflections. And Appel's ray-casting algorithm considers only direct lighting.

29.8.1 Perceptually Significant Phenomena and Light Transport

We conclude this largely mathematical chapter with a discussion of things we notice when we look at the world, for these phenomena—the things we take the trouble to name—are the things we must render effectively if we want to make compelling images. They thus serve as a guide to developing rendering algorithms.

The first phenomenon is the **shadow.** Figure 29.6 shows a hard shadow, created by a luminaire (the sun) that subtends a very small solid angle. The light from the sun is obstructed by the sharp edge of an object. Figure 29.7 shows a soft shadow. Soft shadows can be caused by many things: If the shadowing object is not sharply defined (e.g., a furry animal), the shadow may have no distinct edge, even if it's cast by a point luminaire. If the shadow is cast by a very small object, diffraction effects may dominate and effectively soften the shadow. But most often, soft shadowing is caused by nonpoint luminaires: A point on the receiving surface may be

Figure 29.6: A hard shadow.

Figure 29.7: A soft shadow.

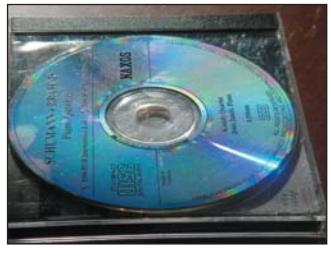


Figure 29.8: Diffraction under point lighting.

invisible from every point on the light source, in which case it's in the **umbra,** or it may be visible from just *some* of the points of the luminaire, in which case it's in the **penumbra,** or it may be visible to all points of the source, in which case it's not shadowed at all. To be more precise: A point P is in the penumbra if the set of points of the luminaire, L, visible from P when the occluder is removed, is different from the visible set when the occluder is present. Thus, if L is a typical incandescent lightbulb, we don't require that every point of the bulb be visible from P (which never happens), but only that every point that *could* be visible from P actually *is* visible.

The next important phenomenon is **boundaries** such as the contours discussed in Chapter 5. Boundaries between objects, between materials in a single object, etc., are all significant. (Indeed, shadow edges are yet another example of boundaries.)

Many surface properties are revealed through reflection of light. Under point illumination, diffractive objects like the surface of the CD in Figure 29.8 clearly reveal their reflectance structure; under area luminaires, the reflection is less focused but still shows the characteristic rainbow pattern (see Figure 29.9). Less extreme than diffraction are the stretched-out highlights that appear on brushed-metal surfaces (see Figure 29.10); the alignment of the brush marks generates these in much the same way that ripples on a pond generate an extruded reflection of moonlight.

And of course, for more common scattering functions, phenomena like edge highlights, generated by very glossy materials with high curvature (see Figure 29.11), are commonplace.

Polarization phenomena are less obvious in general, although anyone who has worn polarized sunglasses knows that mirror-reflected light tends to be somewhat polarized, and therefore can be attenuated by the polarizing lens in the sunglasses.

Caustics (see Figure 29.12) are bright areas that arise from the focusing of illumination by curved surfaces, through either reflection or refraction. Typically they are produced by point sources or small area sources like the sun. Figure 29.13 shows how caustics weaken or disappear under more diffuse illumination.

Assuming for the moment that most caustics arise from sunlight, that is, from light arriving on essentially parallel rays, caustics will, in general, appear close to the curved surfaces that cause them. Figure 29.12 suggests the reason: You can see several rays meeting at a bright spot, but beyond that spot, the rays diverge. Since caustics depend on convergence, once you are sufficiently distant from a surface (roughly, the maximum radius of curvature of the surface) the caustics can no longer appear. By studying the **focal points** of a surface (places at which nearby normal rays meet), you can determine how likely a surface is to cast a caustic on some distant object under random parallel illumination.

There's a dual phenomenon to caustics: Eye rays meeting a curved surface may converge on a single point, or a small range of points. A magnifying glass is designed to take advantage of this phenomenon, for instance, but it also appears in less intentional settings. For instance, the reflections of the blue windows in Figure 29.14 appear somewhat wavy and distorted because of the curvature of the windows in which they're reflected.

Transmission of light through transparent media also generates *refractive* phenomena, like the offset in the top and bottom parts of the red pen in Figure 29.15. Because spatial continuity (things that are straight should appear straight, things in a regular pattern should appear regular, etc.) is easily noticed, its interruption by refraction is perceptually significant.

Figure 29.9: Diffraction under more diffuse lighting.

Figure 29.10: Brushed metal generates stretched-out highlights.

Figure 29.11: The bright line along the edge of the bookshelf results from the glossy reflection of sunlight.

Figure 29.12: Caustics cast by a glass of water in sunlight.

If you are reading this indoors, then most of the light that you are seeing has been reflected multiple times. You can verify this by standing in an empty room with painted walls, illuminated by a single bulb. If you place a black occluder next to the bulb, some portion of the room will be "in shadow," but nonetheless remains remarkably bright. The importance of this multiply reflected light is not well measured by computing energies, because of the logarithmic sensitivity of the eye: Even comparatively dark areas are easy for us to see; these darker areas are ones where indirect illumination has taken multiple attenuating bounces. Even though little energy from them is reaching the eye, we can easily detect variations on surfaces, such as the pattern on a carpet in a dark corner of a room.

These variations in dark regions can, however, be lost in the presence of other light. Figure 29.16 shows how the view through a car window can be masked by reflected light from papers on the dashboard.

Keep these examples in mind as you read about rendering algorithms, and ask yourself which of them each algorithm is capable of replicating.

Figure 29.13: Under diffuse light, the caustics disappear.

Figure 29.14: Distorted reflections.

Figure 29.15: Refraction of light by water can break up spatial continuity.

Figure 29.16: The reflections from the dashboard obscure the view through the window.

29.9 Discussion

The key ideas in this chapter are the reflection equation and the transport equation, which combine to form the rendering equation, and the measurement equation, which is added as a final step in the process of image formation. Equally important is the notion of dividing various phenomena into "impulse" and "finite" pieces. This division applies to the BSDF, where impulses include mirror reflection and Snell's-law refraction, and to luminaires, where point lights act as a kind of impulse in the illumination. Such impulses require that we regard integrals with a skeptical eye, because integrals of finite pieces can be reliably estimated with Monte Carlo methods, as we'll see in the next few chapters, while those involving impulses must be treated separately.

The physical formulation of light transport is central to rendering, but it's also worth understanding the *phenomena*, that is, the human-perceived aspects, of light transport. The tiny "rainbows" cast by a chandelier on the walls of a well-lit room are insignificant in the total illumination by physical measures, but to a person sitting in the room, they are important characteristics that attract attention. Understanding these phenomena helps us understand which aspects of physical light transport may have greater impacts on the perceived correctness of an image.

29.10 Exercise

Exercise 29.1: Show that if you have the ray-casting function R, then you can build a **visibility function** $V : \mathcal{M} \times \mathcal{M} \rightarrow 0, 1$, where $V(P, Q)$ is 1 if all points of the line segment from P to Q that are strictly between the ends are in empty space, and is 0 otherwise. Informally, $V(P. Q)$ is 1 when Q is visible from P (and vice versa). Note that according to our definition, $V(P, P)$ is always 1. You may assume that you have a mechanism for perfect equality testing of points.

Chapter 30

Probability and Monte Carlo Integration

30.1 Introduction

In preparation for studying rendering techniques, we now discuss Monte Carlo integration. We start with a rapid review of ideas from discrete probability theory, and then generalize to continua like the real line or the unit sphere. We apply these notions to describe how to generate random samples from various sets. We then introduce Monte Carlo integration, treating all the basic ideas through the integration of a function on an interval $[a, b]$, where the ideas are easiest to understand. We then show how these ideas apply to integration on a hemisphere or sphere, and hence how they are used to find reflected radiance via the reflectance equation, for instance.

30.2 Numerical Integration

We'll start with a high-level overview of the use of randomization in numerical integration. Sometimes we need to integrate functions where computing antiderivatives is impossible or impractical. For instance, the integrand in the reflectance equation might not be described by an algebraic equation at all. In these cases, numerical methods often are the only workable solution. Numerical methods fall into two categories: deterministic and randomized. Here's a quick comparison of the two.

A typical deterministic method (see Figure 30.1) for integrating a function f over an interval $[a, b]$ is to take $n + 1$ equally spaced points in the interval $[a, b]$, $t_0 = a, t_1 = a + \frac{b-a}{n}, \ldots, t_n = b$, evaluate f at the midpoint $\frac{t_i+t_{i+1}}{2}$ of each of the intervals these points define, sum up the results, and multiply by $\frac{b-a}{n}$, the width of each interval. For sufficiently continuous functions f, as n increases this will converge to the correct value. Similar methods for surface integrals, which divide

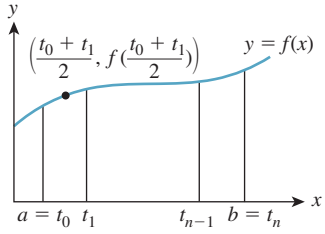


Figure 30.1: Integration using equal partitioning.

the surface into small rectangles, also work. Press et al. [Pre95] describe these and more sophisticated deterministic methods.

A probabilistic or **Monte Carlo** method for computing, or more precisely, for *estimating* the integral of f over the interval $[a, b]$, is this: Let $X_1, \ldots, X_n$ be randomly chosen points in the interval $[a, b]$. Then $Y = (b - a)\frac{1}{n}\sum_{i=1}^{n} f(X_i)$ is a random value whose average (over many different choices of $X_1, \ldots, X_n$) is the integral $\int_{[a,b]} f$. To implement this we use a random number generator to generate n points x_i in the interval $[a, b]$, evaluate f at each, average the results, and multiply by $b - a$. This gives an estimate of the integral, as shown in Figure 30.2. Of course, each time we compute such an estimate we'll get a different value. The variance in these values depends on the shape of the function f. If we generate the random points x_i nonuniformly, we must slightly alter the formula. But in using a nonuniform distribution of points we gain an enormous advantage: By making our nonuniform distribution favor points x_i where $f(x)$ is large, we can drastically reduce the variance of our estimate. This nonuniform sampling approach is called **importance sampling.** Figure 30.3 shows an example.

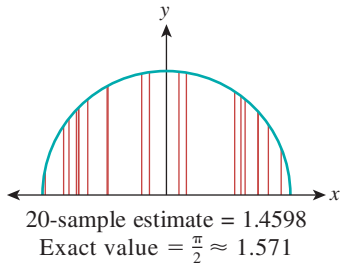


Figure 30.2: Estimating $\int_{-1}^{1} \sqrt{1 - x^2}\, dx$ with 20 samples approximates the correct result, which is $\frac{\pi}{2} \approx 1.571$.

We can integrate a function f over a surface region R by an analogous method. We choose points uniformly randomly on R, average the values of f at these points, and multiply by the area of the region. Or we can generate points nonuniformly, and compute a weighted average, again multiplied by the region's area, generalizing importance sampling. Our main application of numerical integration is when R is either the sphere or the outward-facing hemisphere at some surface point P where we're trying to compute the scattered light (i.e., trying to evaluate the integral in the reflectance equation). We'll return to this application at the end of the chapter.

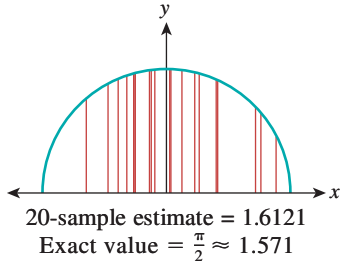


Figure 30.3: Estimating the same integral, again with 20 samples, but preferentially selecting samples near $x = 0$, and using appropriate weighting, gives a better approximation.

Randomized integration techniques form part of the working tools of anyone studying rendering, and much of the rest of graphics. Press et al. [Pre95] is a solid first reference for ideas beyond those discussed in this chapter.

30.3 Random Variables and Randomized Algorithms

Because the major shift in rendering methods over the past several decades was from deterministic to randomized, we'll be discussing randomized approaches to solving the rendering equation. To do so means using random variables, expectation, and variance, all of which we assume you have encountered in some form in the past. To establish notation, and prepare for the continuum and mixed-probability cases, we review discrete probability here. The approach we take may seem a little unusual. That's because our goal is not to *analyze* some probabilistic phenomenon that already exists (e.g., the chance of rainfall in Boston tomorrow, given that it's already been raining for two days), but rather to *construct* probabilistic situations in which something computable, like expected value, turns out to have the same value as some integral (like the reflectance integral) that we care about. That leads to emphasis on some things that are largely downplayed in introductory probability courses.

We often find that for our students, the connection between formal probability theory and programs that implement probabilistic ideas is unclear, and so we discuss this connection as well.

Those familiar with these concepts can skip forward to Section 30.3.8.

30.3.1 Discrete Probability and Its Relationship to Programs

A **discrete probability space** is a nonempty finite[1] set S together with a real-valued function $p : S \to \mathbf{R}$ with two properties.

1. For every $s \in S$, we have $p(s) \geq 0$, and
2. $\sum_{s \in S} p(s) = 1$.

The first property is called non-negativity, and the second is called normality. The function p is called the **probability mass function,** and $p(s)$ is the probability mass for s (or, informally, the probability of s). The intuition is that S represents a set of outcomes of some experiment, and $p(s)$ is the probability of outcome s. For example, S might be the set of four strings hh, ht, th, tt, representing the heads-or-tails status of tossing a coin twice (see Figure 30.4). If the coin is fair, we associate probability $\frac{1}{4}$ to each outcome.

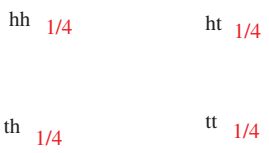


Figure 30.4: A four-element probability space, with uniform probabilities, shown as fractions in red.

Inline Exercise 30.1: Explain why, in a discrete probability space (S, p), we have $0 \leq p(s) \leq 1$ for every $s \in S$. The first inequality follows from the first defining property of p. What about the second?

An **event** is a subset of a probability space. The **probability** of an event (see Figure 30.5) is the sum of the probability masses of the elements of the event, that is,

$$\mathbf{Pr}\{E\} = \sum_{s \in E} p(s). \tag{30.1}$$

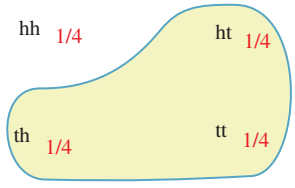


Figure 30.5: An event ("At least one 'tails' ") with probability $\frac{3}{4}$.

Inline Exercise 30.2: Prove that if E_1 and E_2 are events in the finite probability space S, and $E_1 \cap E_2 = \emptyset$, then $\mathbf{Pr}\{E_1 \cup E_2\} = \mathbf{Pr}\{E_1\} + \mathbf{Pr}\{E_2\}$. This generalizes, by induction, to show that for any *finite* set of mutually disjoint events, $\mathbf{Pr}\{\bigcup_i^n E_i\} = \sum_{i=1}^n \mathbf{Pr}\{E_i\}$. One of the axioms of probability theory is that this relation holds even for an infinite, but *countable,* collection of disjoint events when we're working with continua, like $[0, 1]$ or $\mathbf{R}$, rather than a discrete probability space.

A **random variable** is a *function,* usually denoted by a capital letter, from a probability space to the real numbers:

$$X : S \to \mathbf{R}. \tag{30.2}$$

The terminology is both suggestive and misleading. The function X is not a variable; it's a real-valued function. It's not *random,* either: $X(s)$ is a single real number for any outcome $s \in S$. On the other hand, X serves as a *model* for something random. To give a concrete example, consider the four-element probability space described earlier. There's a random variable X defined on that space by the notion "How many heads came up?" or, more formally, "How many hs does the string contain?" To be explicit, we can define the random variable X by

$$X(hh) = 2 \qquad X(ht) = 1 \qquad X(th) = 1 \qquad X(tt) = 0. \tag{30.3}$$

1. Countably infinite sets such as the integers are often included in the study of discrete probability, but we'll have no need for them, and restrict ourselves to the finite case.

This function takes on real values (either 0, 1, or 2), and it's defined on the set S; hence, it's a random variable. As in this example, we'll use the letter X to denote nearly every random variable in this chapter.

A random variable can be used to define events. For example, the set of outcomes s for which $X(s) = 1$, that is,

$$E = \{s \in S : X(s) = 1\} \text{ and} \tag{30.4}$$

$$= \{ht, th\}, \tag{30.5}$$

is an event with probability $\frac{1}{2}$. We write this $\mathbf{Pr}\{X = 1\} = \frac{1}{2}$, that is, we use the predicate $X = 1$ as a shorthand for the event defined by that predicate.

Let's look at a piece of code for simulating the experiment represented by the formalities above:

```
1  headcount = 0
2  if (randb()): // first coin flip
3      headcount++
4  if (randb()): // second coin flip
5      headcount++
6
7  return headcount
```

Here we're assuming the existence of a `randb()` procedure that returns a boolean, which is `true` half the time.

How is this simulation related to the formal probability theory? After all, when we run the program the returned value will be either 0, 1, or 2, not some mix of these. (If we run it a second time, we may get a different value, of course.)

The relationship between the program and the abstraction is this:

Imagine the set S of *all possible executions* of the program, declaring two executions to be the same if the values returned by `randb` are pairwise identical. This means that there are four possible program executions, in which the two `randb()` invocations return TT, TF, FT, and FF. Our belief and experience is that these four executions are equally likely, that is, each occurs just about one-quarter of the time when we run the program many times.

The analogy is now clear: The set of possible program executions, with associated probabilities, is a probability space; the variables in the program that depend on `randb` invocations are random variables. This is a quite clever piece of mathematical modeling of computation. You should be certain that you understand it.

 There's nice formalism in which a procedure that invokes `randb`-like functions is replaced by one with an extra argument, which is a long string of bits; these bits are used instead of `randb` calls. This revised procedure is **deterministic;** it becomes randomized only when we invoke it with a sequence of random bits. In this formalism, the probability space is the set of all possible bit strings that are provided as the extra argument.

30.3.2 Expected Value

The **expectation** or **expected value** or **mean** of a random variable X on a finite probability space S is defined as

$$\mathbf{E}[X] = \sum_{s \in S} p(s)X(s). \tag{30.6}$$

In the case of our "counting heads in a coin-flip" experiment, the expectation is

$$\begin{aligned}\mathbf{E}[X] &= p(hh)X(hh) + p(ht)X(ht) + p(th)X(th) + p(tt)X(tt), && (30.7)\\ &= \frac{1}{4}\cdot 2 + \frac{1}{4}\cdot 1 + \frac{1}{4}\cdot 1 + \frac{1}{4}\cdot 0, && (30.8)\\ &= 1. && (30.9)\end{aligned}$$

If we in fact run the coin-flipping program many times, we'll sometimes see a heads count of 0; sometimes 1; and sometimes 2. The average number of heads, over many executions of the program, will be about 1.

We can rewrite the expectation in Equation 30.7 by asking, for each possible value taken on by X (i.e., 0, 1, and 2), what fraction of the items in S correspond to that value. For the value 1, we have $X(ht) = 1$ and $X(th) = 1$, so two of the four items in S, in other words, half, correspond to the value 1. We then sum these fractions times the associated values, and thus compute

$$\begin{aligned}\mathbf{E}[X] &= \sum_{r=0,1,2} r \cdot \mathbf{Pr}\{X = r\}, && (30.10)\\ &= 0\cdot\frac{1}{4} + 1\cdot\frac{1}{2} + 2\cdot\frac{1}{4} = 1. && (30.11)\end{aligned}$$

This latter form is used in many applications of probability, because it depends only on the *values* of X and the associated probabilities, and the probability space S does not appear directly. In our applications, however, this form will not appear again.

Nonetheless, the function $r \mapsto \mathbf{Pr}\{X = r\}$, and its generalization in the continuum case, is of substantial interest to us. Note that it's defined on the *codomain* of X, not the domain. And for each codomain value r, it tells us (if we're thinking of X informally as "producing a random output") the probability that X will produce the output r. This function is called the **probability mass function** (or pmf) for the random variable X, and is denoted p_X.

One reason the pmf is interesting is that it can be generalized to apply to a function $Y : S \to T$, sending our probability space to *any* finite set T, not just the real numbers. For such a function, we define

$$p_Y(t) = \mathbf{Pr}\{Y = t\} \text{ for } t \in T. \tag{30.12}$$

This function, p_Y, is a probability mass function on T so that (T, p_Y) becomes a probability space.

Inline Exercise 30.3: (a) How do we know that p_Y is a probability mass function, that is, it satisfies the two requirements of non-negativity and normality? (b) If $E \subset T$ is an event, show that Equation 30.12 implies that the probability of the event E in the space (T, p_Y) is given by $\mathbf{Pr}_T\{E\} = \mathbf{Pr}_S\{Y^{-1}(E)\}$. (Here $Y^{-1}(E)$ denotes the set of all points $s \in S$ such that $Y(s) \in E$.)

We've now used the term "probability mass function" in two different ways: When we spoke of a probability space (S, p), we called p the probability mass function. But we've also described the pmf for a random variable $X : S \to \mathbf{R}$, or even for a function $Y : S \to T$ into an arbitrary set. We'll now show that

these uses are consistent. Consider the identity function $I : S \to S : s \mapsto s$. By Equation 30.12, we have

$$p_I(t) = \mathbf{Pr}\{I = t\} \text{ for } t \in S. \tag{30.13}$$

For a fixed element $t \in S$, what is the event $I = t$? It's the event

$$\{s \in S : I(s) = t\} = \{s \in S : s = t\} = \{t\}. \tag{30.14}$$

The probability of this event is just the sum of the probability masses for all outcomes in the event, that is, $p(t)$. Thus, the probability mass function for the identity map is the same thing as the probability mass function p for the probability space itself.

Functions like Y that send a probability space to some set T rather than $\mathbf{R}$ are sometimes still called random variables. In the case we're most interested in, Y will send the unit square to the unit sphere or upper hemisphere, and we'll use the terminology "random point." We'll use the letter Y to denote random-point functions, just as we use X for all the random variables in the chapter.

We conclude with one last bit of terminology. The function p_Y, whether Y is a random variable or a map from S to some arbitrary set T, is called the **distribution** of Y; we say that Y is distributed according to p_Y. This suggestive terminology is useful when we're thinking of things like student scores on a test, where it's common to say, "The scores were distributed around a mean of 82." In this case, the underlying probability space S is the set of students, with a uniform probability distribution on S, and the test score is a random variable from students to $\mathbf{R}$. You'll often see the notation $X \sim f$ to mean "X is a random variable with distribution f," that is, with $p_X = f$.

30.3.3 Properties of Expected Value, and Related Terms

The expected value of a random variable X is a constant; it's often denoted $\bar{X}$. Note that $\bar{X}$ can also be regarded as a constant *function* on S, and hence be treated as a random variable as well.

Expectation has two properties that we'll use frequently: $\mathbf{E}[X + Y] = \mathbf{E}[X] + \mathbf{E}[Y]$, and $\mathbf{E}[cX] = c\mathbf{E}[x]$ for any real number c. In short, *expectation is linear*.

> Expectation and variance (the latter which we'll encounter in a moment) are higher-order functions: Their arguments are not numbers or points, but actual functions—in particular, random variables.
>
> When we write $\mathbf{E}[X + Y]$, we mean the expectation of the sum of the two random variables X and Y, that is, the function $s \mapsto X(s) + Y(s)$. This means that when we write $X+X$, we mean the function $s \mapsto X(s)+X(s)$. If X happens to correspond to a piece of code that looks like `return randb(...)`, then you *cannot* implement $X + X$ by `X() + X()`: The two invocations of the random number generator may produce different values!

In general, when you look at a program with `randb` in it, your analysis of the program will involve a probability space with 2^k elements, where k is the number of times `randb` is invoked; each element will have equal probability mass. The same will be true when we look at `rand`: Any two numbers between 0 and 1 will

be equally likely to be the output of any invocation of `rand`. We call `randb` a **uniform random variable,** because all of its outputs are equally likely.

Note that if `randb` is invoked a variable number of times in the program, then some elements of the program's probability space will have greater probability than others.

Inline Exercise 30.4: Suppose we modify our coin-flipping program to only flip twice if the first coin came up heads, as in Listing 30.1. Describe a probability space for this code, and compute the expected value of the random variable *headcount*.

Listing 30.1: A program in which `randb` *is invoked either once or twice.*

```
1  headcount = 0
2  if (randb()): // first coin flip
3       headcount++
4       if (randb()): // second coin flip
5            headcount++
6  return headcount
```

The values taken on by a random variable X may be clustered around the mean $\bar{X}$, or widely dispersed. This dispersion is measured by the **variance,** the mean-square average of $X(s) - \bar{X}$. Formally, the variance is

$$\mathbf{Var}[X] = \mathbf{E}[(X - \bar{X})^2], \tag{30.15}$$

which can be simplified (see Exercise 30.9) to $\mathbf{E}[X^2] - \mathbf{E}[X]^2$.

The units of variance are the units of X, squared; the square root of the variance (known as the **standard deviation**) has the same units as X, and sometimes makes better intuitive sense. As a useful rule of thumb, three-fourths of the values of X lie within two standard deviations of $\bar{X}$.

Variance is *not* linear, but has the property that $\mathbf{Var}[cX + d] = c^2\mathbf{Var}[X]$ for any real numbers c and d.

The random variables X and Y are called **independent** if

$$\mathbf{Pr}\{X = x \text{ and } Y = y\} = \mathbf{Pr}\{X = x\} \cdot \mathbf{Pr}\{Y = y\} \tag{30.16}$$

for every x and y in $\mathbf{R}$. For instance, in the experiment where we flip an unbiased coin twice, if X is the number of heads that show up on the first coin flip (either 0 or 1), and Y is the number of heads that show up on the second, then our experience tells us that X and Y are independent, that is, the probability of two heads in a row is $\frac{1}{4}$. On the other hand, the variables X and X are distinctly *not* independent. We'll generally assume that any two values produced by calls to `randb` or `rand` or other such functions correspond to independent random variables.

The important properties for *independent* random variables are

- $\mathbf{E}[XY] = \mathbf{E}[X]\mathbf{E}[Y]$
- $\mathbf{Var}[X + Y] = \mathbf{Var}[X] + \mathbf{Var}[Y]$

Returning to the coin-flip experiment and the associated program, we implemented the two random variables via calls to a single random-number-generating routine, `randb`. We call these random variables samples from the distribution in which heads and tails have equal probability. The idea is that `randb` can be

thought of as producing a long list of random zeroes and ones, each equally likely, and we've simply grabbed a couple of these.

Together, these two samples are therefore *independent,* and, because they came from the same distribution, **identically distributed.** Such random variables occur often, and we refer to them as **independent identically distributed** or **iid** random variables or samples.

30.3.4 Continuum Probability

The entire discrete-probability framework can be extended by analogy to countably infinite sets, which requires some care because we have to talk about sums of infinite series, and worry about convergence. That particular case isn't of much interest in graphics, but the study of probability on *uncountably* infinite domains like the unit interval, or the unit sphere, comes up repeatedly. We'll refer to such a domain, on which you know how to compute integrals, as a **continuum,** and speak of **continuum probability** as contrasted with the discrete probability discussed earlier. Some books use the term **continuous probability** for this situation, but since we'll want to be able to discuss continuous and discontinuous functions, we prefer "continuum." In the continuum case, we'll analyze programs that contain `rand` (which returns random real numbers between 0 and 1, with every number being just as likely as every other number) rather than `randb`. There are three difficulties.

1. The procedure `rand` doesn't really produce real numbers in $[0, 1]$; it produces floating-point representations of a tiny subset of them.
2. ◈ Certain aspects of the analysis involve mathematical subtleties like measurability.
3. Our probability space will now be *infinite,* and we'll need to talk about probability *density* rather than probability mass.

We'll mostly ignore difficulties 1 and 2, on the grounds that they have little impact in the practical applications we make. Difficulty 3, however, matters quite a lot.

Let's look at another sample program as motivation. To make the code as readable as possible, we'll avoid the use of `rand` and instead write `uniform(a, b)` to indicate a procedure that produces a random real number between a and b, with each output being equally probable. This is typically implemented with `a + (b-a)*rand()`.

```
1 u = uniform(0,1); // a random real between 0 and 1
2 w = √u
3 return w
```

On the next several pages, we'll describe the sample space associated with this program, the notion of probability density, and the definitions of random variable and expected value, and will eventually compute the expected value of w.

First, for this continuum situation, a probability space is a pair (S, p) consisting of a set S, such as the real line, the unit interval, the unit square, the upper hemisphere, the whole sphere, etc., on which integration is defined, and a **probability density function,** or just **density** $p : S \to \mathbf{R}$ (see Figure 30.6), with two properties:

- Non-negativity: For all $s \in S$, $p(s) \geq 0$
- Normality: $\int_S p = 1$

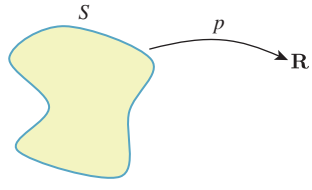


Figure 30.6: The continuum S has a density function p assigning a density to each point of S.

The second property means that when p is integrated over all of S, the result is 1, whether S is something one-dimensional like the interval $[a, b]$, in which case the normality condition would be written $\int_a^b p(s)\, ds = 1$, or something two-dimensional, like the unit square, in which case we'd write

$$\int_0^1 \int_0^1 p(x, y)\, dy\, dx = 1. \tag{30.17}$$

For the most part our probability spaces will be things like the interval or the sphere, which have the property that when we integrate the constant function 1 over them, the result is some finite value, which we'll call the size of S, and denote $size(S)$. In these cases, the associated density will usually be the constant function with value $1/size(S)$, called a **uniform density**. Note that there is no uniform density for the real line, however.

In the discrete case, the normality condition involved a sum; in the continuum case, it involves an integral. You might be tempted to think of the probability density as just like the probability mass in the discrete case, but they're quite different, as the next inline exercise shows. The proper interpretation is that the density represents probability *per unit size*. Thus, in cases where we have units, probability and probability density differ by the units of size (i.e., length, area, or volume).

Inline Exercise 30.5: Let $S = [0, 1]$ and $p(x) = 2x$. Show that (S, p) is a probability space by checking the two conditions on p. Observe that $p(1) = 2$, so a probability *density* may have values greater than 1, even though probability *masses* are never greater than 1.

To return to our program, the set of all possible executions of the program is infinite:[2] In fact, there's one execution for every real number between 0 and 1. So we can say that our probability space is $S = [0, 1]$, the unit interval. And since we regard each possible value of `uniform(0,1)` as equally likely, we associate to S the uniform density defined by $p(x) = 1$ for all $x \in [0, 1]$.

Just as in the discrete case, a **random variable** is a function X from S to $\mathbf{R}$, and an **event** is a subset of S, but we'll mostly restrict our attention to events of the form $a \leq X \leq b$, where X is some random variable.

To be honest, not every subset of S is an event; only the "measurable" ones. But it's essentially impossible to write down a non-measurable set, and certainly not possible to encounter one while performing computations on an ordinary computer, so we'll ignore this subtlety. If you like, you may consider events to be restricted to things like intervals or rectangles, or other similarly nice sets over which you know how to integrate.

The **probability of an event** E in a probability space (S, p) is the integral of p over E, just as in the discrete space the probability of an event is the sum of the probability masses of the points in the event.

2. We're pretending that our random number generator returns real numbers, rather than floating-point representations of them.

The **expected value of a random variable** X on a probability space (S, p) is defined to be

$$\mathbf{E}[X] := \int_{s \in S} X(s) p(s)\, ds. \tag{30.18}$$

As in the discrete case, expectation is linear.

Inline Exercise 30.6: (a) In the special case where S is a space with a uniform density, show that the expectation of the random variable X is just $\mathbf{E}[X] = \frac{1}{size(S)} \int_{s \in S} X(s)\, ds$.
(b) What is the expectation of a random variable Z on the interval $[a, b]$ with uniform density?

We'll now apply the notion of expectation to the example code. The variable `u` in the program corresponds to a random variable U on the interval $[0, 1]$. Similarly, the variable `w` in the program corresponds to a random variable $W = \sqrt{U}$. The expected value of W is, according to the definition,

$$\mathbf{E}[W] = \int_0^1 W(r) p(r)\, dr, \tag{30.19}$$

$$= \int_0^1 \sqrt{r}\, dr = \frac{2}{3}. \tag{30.20}$$

This should match your intuition: The variable U is uniformly distributed on $[0, 1]$, so its expected value is $\frac{1}{2}$. But for any number $0 < u < 1$, we have $u < \sqrt{u}$, so the average square root of any number should be bigger than the average number, that is, we anticipate that the expected value of W will be somewhat larger than $\frac{1}{2}$.

30.3.5 Probability Density Functions

In analogy with the probability mass function for a random variable on a discrete space described in Section 30.3.2, we'll now formulate the corresponding notion for a random variable on a continuum.

It often happens that for a random variable X, and the special class of events of the form $a \le X \le b$ (i.e., the set $\{s : a \le X(s) \le b\}$), there's a function p_X, called the **probability density function** (pdf) or **density** or **distribution** for X, with the property that

$$\mathbf{Pr}\{a \le X \le b\} = \int_a^b p_X(r)\, dr. \tag{30.21}$$

For the time being, we'll consider *only* random variables that have a pdf.

The intuition for p_X, for a random *variable* X, is that for small values of Δ, the number $p_X(a)\Delta$ is approximately the probability that X lies in an interval of size Δ, centered at a, or $[a - \Delta/2, a + \Delta/2]$, with the approximation being better and better as $\Delta \to 0$.

Inline Exercise 30.7: Explain why, if X is a random variable on the probability space S, with pdf p_X, $\int_{-\infty}^{\infty} p_X(r)\, dr = 1$.

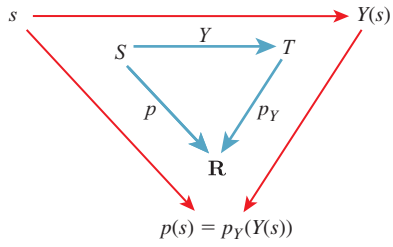


Figure 30.7: The density p_Y is defined so that starting from a point $s \in S$, you get the same result no matter which way you traverse the arrows, that is, $p(s) = p_Y(Y(s))$.

Just as in the discrete case, if S is a continuum probability space with associated density p and

$$Y : S \to T, \tag{30.22}$$

we can use Y to define a probability density on T (see Figure 30.7). We'll restrict our attention to the special case where Y is an invertible function, although we'll apply the results to cases where Y is "almost invertible," like the latitude-longitude parameterization of the sphere, where the noninvertibility is restricted to a set whose "size" is zero. In the case of the spherical parameterization, if we ignore all points of the international dateline, the parameterization is 1-1, and the dateline itself has size zero (where "size," for a surface, means "area").

Our definition closely follows the discrete case:

$$p_Y(t) = p(Y^{-1}(t)). \tag{30.23}$$

Returning to the example program, the variable U has as its pdf the function

$$p_U : [0, 1] \to \mathbf{R} : r \mapsto 1. \tag{30.24}$$

This evidently integrates to 1 on the whole interval.

> For any particular number, like $r = 0.3$, the probability that $U = r$ is $\int_r^r p_U(r)\, dr = \int_r^r 1\, dr = 0$. Intuitively, if we're picking a random number between 0 and 1, the probability of picking any *particular* number ends up being zero. Despite this, we do pick *some* number. This is a situation where the probability of a union of disjoint events (the events being "pick 0.134," "pick $\pi/13$," and infinitely many other similar events) is *not* the sum of the individual probabilities. Thus, when we shift from finite sets to infinite ones, some of our intuition about probability turns out to be mistaken.

The pdf for W is not so obvious. Evidently, values near 0 occur less often as outputs of W than do those near 1, but what's the exact pattern? The answer is obviously *not* that it's the square root of the pdf for U. We seek a function p_W with the property that

$$\mathbf{Pr}\{a \le W \le b\} = \int_a^b p_W(r)\, dr. \tag{30.25}$$

In the left-hand side of this equation is the event that $a \le W \le b$; let's rewrite this:

$$\{a \le W \le b\} = \{s \in [0, 1] : a \le W(s) \le b\}; \tag{30.26}$$

$$= \{s \in [0, 1] : a \le \sqrt{s} \le b\}; \tag{30.27}$$

$$= \{s \in [0, 1] : a^2 \le s \le b^2\}. \tag{30.28}$$

The probability of that last event is $b^2 - a^2$. (Why?) So we need to find a function p_W with the property that for any $a, b \in [0, 1]$,

$$\int_a^b p_W(r)\, dr = b^2 - a^2. \tag{30.29}$$

A little calculus shows that $p_W(r) = 2r$ (see Exercise 30.4).

The notion of a random variable can be generalized to that of a *random point*. If S is a probability space, and $Y : S \to T$ is a mapping to another space rather than the reals, then we call Y a "random point" rather than a random variable. The notion of a probability density applies here as well, but rather than just looking at an interval $[a, b] \subset \mathbf{R}$, we must now consider any set $U \subset T$; we'll say that p_Y is a pdf for the random variable Y if, for every (measurable) subset $U \subset T$, we have

$$\mathbf{Pr}\{Y \in U\} = \int_{u \in U} p_Y(u)\, du. \tag{30.30}$$

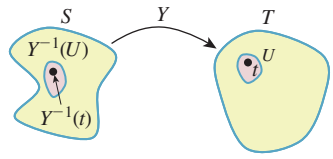


Figure 30.8: The probability of $U \subset T$ is just the size of $Y^{-1}(U) \subset S$ divided by the size of S.

In fact, it's usually fairly easy to compute p_Y if we know the mapping Y by an argument completely analogous to the one used to find p_W.

In the special case where S has a uniform probability density (see Figure 30.8), the probability of $Y \in U$ is just the probability of the set $Y^{-1}(U) \subset S$, which is the size of $Y^{-1}(U)$ divided by the size of S.

If we assume that p_Y is continuous, then we can compute p_Y directly. Consider the case of a very small neighborhood U of some point $t \in T$; then the right-hand side is given by

$$\int_{u \in U} p_Y(u)\, du \approx size(U) p_Y(t). \tag{30.31}$$

On the other hand, the left-hand side is given by a ratio of sizes as described above, so

$$\frac{size(Y^{-1}(U))}{size(S)} = size(U) p_Y(t), \text{ and therefore} \tag{30.32}$$

$$p_Y(t) \approx \frac{size(Y^{-1}(U))}{size(U)} \frac{1}{size(S)}. \tag{30.33}$$

The first factor in this expression is just an approximation of the change of area for Y^{-1}, which is given by the Jacobian of Y^{-1} at t. So in the limit, as U shrinks to a smaller and smaller neighborhood of $Y(t)$, we get

$$p_Y(t) = |(Y^{-1})'(t)| \frac{1}{size(S)}. \tag{30.34}$$

As in the discrete case, we can use p_Y as a probability density for the space T, or we can simply use it as the pdf for the random point Y.

And once again, if $\iota : S \to S : s \mapsto s$ is the identity map, then $p_\iota = p$, so the two notions of density—the probability density used in defining a continuum probability space, and the probability density of a random variable on that space—are in fact consistent.

Finally, we will again use the notation $X \sim f$ to mean "X is a random variable distributed according to f" which in the continuum case means that $p_X = f$. There is one standard distribution that we'll use repeatedly, $\mathbf{U}(a, b)$, which is the uniform distribution on $[a, b]$, or the constant function $\frac{1}{b-a}$. You'll often see this in forms like "Suppose $X, Y \sim \mathbf{U}(0, \pi)$ are two uniform random variables ..."

30.3.6 Application to the Sphere

Let's apply the ideas of the previous section to the longitude-colatitude parameterization of $\mathbf{S^2}$, namely,

$$Y : [0,1] \times [0,1] \to \mathbf{S^2} : (u,v) \mapsto (\cos(2\pi u)\sin(\pi v), \cos(\pi v), \sin(2\pi u)\sin(\pi v)). \tag{30.35}$$

We'll start with the uniform probability density p on $[0,1] \times [0,1]$, or $p(u,v) = 1$ for all u, v. For notational convenience, we'll write $(x,y,z) = Y(u,v)$.

We have the intuitive sense that if we pick points uniformly randomly in the unit square and use f to convert them to points on the unit sphere, they'll be clustered near the poles. (If you doubt this, you should write a little program to verify it.) This means that the induced probability density on T will not be uniform.

The preceding section shows that

$$p_Y(x,y,z) = \frac{1}{|\,Y'(u,v)\,|} p(u,v) \tag{30.36}$$

$$= \frac{1}{|\,Y'(u,v)\,|}. \tag{30.37}$$

But the change-of-area factor (see Figure 30.9) for Y (which is slightly messy to compute) turns out to be

$$|\,Y'(u,v)\,| = 2\pi^2\,|\sin(\pi v)\,| \tag{30.38}$$

$$= 2\pi^2\sqrt{1-\cos(\pi v)^2} \tag{30.39}$$

$$= 2\pi^2\sqrt{1-y^2}. \tag{30.40}$$

And hence,

$$p_Y(x,y,z) = \frac{1}{2\pi^2\sqrt{1-y^2}}. \tag{30.41}$$

Thus, the probability of sampling a point in a small disk of area A centered on the sphere point (x,y,z) is approximately $A/(2\pi^2\sqrt{1-y^2})$.

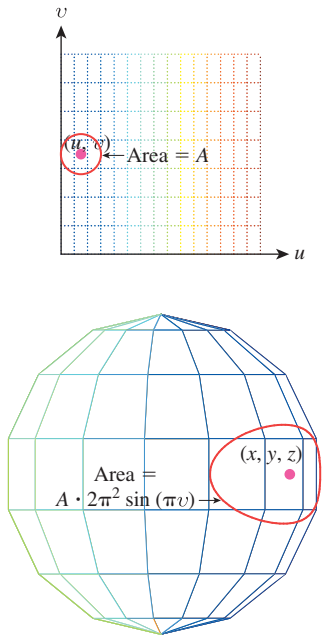


Figure 30.9: A small area A in the domain of the spherical parameterization gets multiplied by $2\pi^2 \sin(\pi v)$.

30.3.7 A Simple Example

If we start with the uniformly distributed random variable U on $[0,1] \times [0,1]$ and want a uniformly distributed variable V on, say, $[0, 2\pi] \times [0,1]$, it seems obvious to define $V(a,b) = U(\frac{a}{2\pi}, b)$. The density for U is the function

$$p_U : [0,1] \times [0,1] \to \mathbf{R} : (x,y) \mapsto 1. \tag{30.42}$$

That makes the density for V be the function

$$p_V : [0,2\pi] \times [0,1] \mapsto \mathbf{R} : (x,y) \mapsto \frac{1}{2\pi}. \tag{30.43}$$

(Quick proof: V is evidently uniformly distributed, and hence its pdf is a constant. The integral of that constant over the domain must be 1, so the constant is $\frac{1}{2\pi}$.)

In code, we might see this:

```
x = uniform(0, 1);
y = uniform(0, 1); // U = (x, y)
x′ = 2 * π * x;
y′ = y;      // V= (x′, y′)
return (x′, y′)
```

There's no mention of the pdf in the code, but it matters nonetheless when we want to analyze the code.

30.3.8 Application to Scattering

The ability to easily draw samples from a distribution (i.e., to produce many independent random variables all having the same specified pdf) will be critical in rendering. In estimating the reflectance integral, we sometimes want to pick a random direction ω in the positive hemisphere with probability density proportional to the bidirectional reflectance distribution function (BRDF), with the first argument held constant at the incoming direction, that is, the function $\omega \mapsto f_r(\omega_i, \omega)$. (Notice that this function is *not* in general a probability density on the hemisphere: Its integral is not 1.) Unfortunately, sampling in proportion to a general function on the hemisphere is not often easy.

We now examine two examples where such direct sampling is possible.

Example 1: The Lambertian BRDF. We need to choose a direction on the hemisphere uniformly at random (i.e., the probability that the direction lies in any solid angle $\Omega \subset \mathbf{S}^2_+$ is proportional to the measure of Ω.) In fact, since the total area of the hemisphere is 2π, we need the probability that the direction lies in a solid angle Ω to be exactly $\frac{m(\Omega)}{2\pi}$.

Fortunately, as we saw in Section 26.6.4, the map

$$g : (x, y, z) \mapsto (x\sqrt{1-y^2}, y, z\sqrt{1-y^2}) \tag{30.44}$$

from the unit cylinder around the y-axis to the unit sphere is area-preserving (although it's not length- or angle-preserving). By restricting our attention to the half-cylinder

$$H = \{(x, y, z) : x^2 + z^2 = 1, 0 \le y \le 1\}, \tag{30.45}$$

we get an area-preserving map from H to $\mathbf{S}^2_+$. And the map

$$f : [0,1] \times [0,1] \to H : (u, v) \mapsto (\cos(2\pi u), v, \sin(2\pi u)) \tag{30.46}$$

multiplies areas by exactly 2π, as you can check by computing its Jacobian. So the composition $Y = g \circ f$ of the two gives us a map from the unit square to $\mathbf{S}^2_+$ that multiplies areas by 2π. The density for Y is therefore the constant function $\frac{1}{2\pi}$ on $\mathbf{S}^2_+$.

In the program shown in Listing 30.2, the returned point is a random point on the hemisphere with density $\frac{1}{2\pi}$.

Listing 30.2: Producing a uniformly distributed random sample on a hemisphere.

```
Point3 randhemi()
  u = uniform(0, 1)
  v = uniform(0, 1)
  r = sqrt(1 - y*y)
  return Point3(r cos(2πu), y, r sin(2πu));
```

Example 2: Cosine-weighted sampling on a hemisphere. Now suppose that we want to sample from a hemisphere, but we want the probability of picking a point in the neighborhood of ω to be proportional to $\omega \cdot \mathbf{n}$, where $\mathbf{n} = \begin{bmatrix}0 & 1 & 0\end{bmatrix}^{\mathbf{T}}$ is the unit normal to the xz-plane that bounds the hemisphere. If we write $\omega = \begin{bmatrix}x & y & z\end{bmatrix}$, then $\omega \cdot \mathbf{n}$ is simply y.

The function $(x, y, z) \mapsto y$ is not a probability density on the hemisphere $y \geq 0$, because its integral is π rather than 1.0. Thus, $(x, y, z) \mapsto \frac{y}{\pi}$ *is* a pdf. We'd now like to sample from this pdf (i.e., we'd like to create a random-point Y whose distribution is this pdf). Because of the area-preserving property of the radial projection map, we can instead choose a random point on the half-cylinder, with density $\frac{y}{\pi}$, and then project to the hemisphere.

Because the density is independent of the angular coordinate on the cylinder, all we need to do is to generate values of $y \in [0, 1]$ whose density is proportional to y. The computation following Equation 30.26 shows that if U is uniformly distributed on $[0, 1]$, then $W = \sqrt{U}$ is linearly distributed, that is, the probability density for W at $t \in [0, 1]$ is $2t$.

Listing 30.3 shows the resultant code.

Listing 30.3: Producing a cosine-distributed random sample on a hemisphere.

```
point3 = randhemi()
  θ = uniform(0, 2 * M_PI)
  s = uniform(0, 1)
  y = sqrt(s)
  r = sqrt(1 - y^2)
  return Point3(r cos(θ), y, r sin(θ))
```

So certain distributions, like the uniform and the cosine-weighted ones, are easy to sample from. More general distributions are often not. When we design reflectance models (i.e., families of BRDFs that can be fit to observed data), one of the criteria, naturally, is goodness of fit: "Is our family rich enough to contain functions that match the observed data decently?" Another criterion is, "Once we fit our data, will we be able to sample from the resultant distributions effectively?" The two criteria are somewhat related, as described by Lawrence et al. [LRR04], who propose a factored BRDF model from which it's easy to do effective importance sampling.

30.4 Continuum Probability, Continued

We're now going to talk about ways to estimate the expected value of a continuum random variable, and using these to evaluate integrals. This is important because in rendering, we're going to want to evaluate the reflectance integral,

$$L^{\text{ref}}(P, \omega_o) = \int_{\omega_i \in \mathbf{S}^2_+} L(P, -\omega_i) f_s(P, \omega_i, \omega_o)\ \omega_i \cdot \mathbf{n}\ d\omega_i, \tag{30.47}$$

for various values of P and ω_o. Knowing how to build a random variable whose expected value is exactly this integral will be a big help in this evaluation.

We saw that code that generated a random number uniformly distributed on $[0, 1]$ and took its square root produced a value between 0 and 1 whose expected value was $\frac{2}{3}$. This also happens to be the average value of $f(x) = \sqrt{x}$ on that interval. This wasn't just a coincidence, as the following theorem shows.

Theorem: If $f : [a,b] \to \mathbf{R}$ is a real-valued function, and $X \sim \mathbf{U}(a,b)$ is a uniform random variable on the interval $[a,b]$, then $(b-a)f(X)$ is a random variable whose expected value is

$$\mathbf{E}[(b-a)f(x)] = \int_a^b f(x)\,dx. \tag{30.48}$$

This is a big result! It says that we can use a randomized algorithm to compute the value of an integral, at least if we run the code often enough and average the results. The remainder of this chapter amplifies and improves this result.

The proof of the theorem is simple. The pdf for X is $p_X(x) = \frac{1}{b-a}$, so the expected value for $(b-a)f(X)$ is

$$\mathbf{E}[(b-a)f(X)] = \int_a^b (b-a)f(x)p_X(x)\,dx; \tag{30.49}$$

$$= \int_a^b (b-a)f(x)\frac{1}{b-a}\,dx; \tag{30.50}$$

$$= \int_a^b f(x)\,dx. \tag{30.51}$$

Listing 30.4 shows the corresponding program.

Listing 30.4: Estimating the integral of the real-valued function f on an interval.

```
integrate1(double *f (double), double a, double b):
   // estimate the integral of f on [a, b]
   x = uniform(a, b) // a random real number in [a, b]
   y = (*f)(x) * (b - a)
   return y
```

The value y returned by the program is a random variable that's an estimate of the integral of f. The average quality of this estimate is measured by the variance of y, which depends on the function f. If f is constant, for example, the estimate is always exactly correct; if f has enormous variations (e.g., $f(x) = 10000\sin(\frac{2\pi x}{b-a})$), then the estimate is highly likely to be very bad. While it's true that if we run the program many times and average the results, we'll get something close to the correct value of the integral, any *individual* run of the program is likely to produce a value that's very far from the correct value.

Inline Exercise 30.8: Implement the small program above in your favorite language, and apply it to estimate the average value of a few functions—say, $f(x) = 4, f(x) = x^2$, and $f(x) = \sin(\pi x)$ on the interval $[0, 2\pi]$.

In general, when we think of a random variable as providing an estimate of some value, low variance is good and high variance is bad.

Let's now improve our estimation of the integral by taking *two* iid samples (see Listing 30.5).

Listing 30.5: Estimating the integral of the real-valued function f on an interval, version 2.

```
def integrate2(double *f (double), double a, double b):
    // estimate the integral of f on [a, b]
    x1 = uniform(a, b)  // random real number in [a, b]
    x2 = uniform(a, b)  // second random real number in [a, b]
    y = 0.5 * ((*f)(x1) + (*f)(x2)); // average the results
    return y * (b - a)      // multiply by length of interval
```

It seems intuitively obvious that the expected value here is still the integral of f on $[a,b]$. Let's verify that. In mathematical terms, we have two random variables, X_1 and X_2, each with a uniform distribution on $[a,b]$. We then compute $Y = \frac{1}{2}(f(X_1) + f(X_2))$. What is $\mathbf{E}[(b-a)Y]$? Using the linearity of expectation, we see that

$$\mathbf{E}[(b-a)Y] = \mathbf{E}[(b-a)\frac{1}{2}(f(X_1) + f(X_2))] \tag{30.52}$$

$$= \frac{1}{2}\mathbf{E}[(b-a)(f(X_1) + f(X_2))] \tag{30.53}$$

$$= \frac{1}{2}(\mathbf{E}[(b-a)f(X_1)] + \mathbf{E}[(b-a)f(X_2)]) \tag{30.54}$$

$$= \frac{1}{2}(2\mathbf{E}[(b-a)f(X_1)]) \tag{30.55}$$

The last equality follows because X_1 and X_2 have the same distribution. Thus it follows that

$$\mathbf{E}[(b-a)Y] = \mathbf{E}[(b-a)f(X_1)] \tag{30.56}$$

$$= \int_a^b f(x)\,dx. \tag{30.57}$$

What about the variance?

$$\mathbf{Var}[Y] = \mathbf{Var}[\frac{1}{2}(f(X_1) + f(X_2))] \tag{30.58}$$

$$= \frac{1}{4}\mathbf{Var}[f(X_1) + f(X_2)] \tag{30.59}$$

$$= \frac{1}{4}(\mathbf{Var}[f(X_1)] + \mathbf{Var}[f(X_2)]) \tag{30.60}$$

$$= \frac{1}{4}(2\mathbf{Var}[f(X_1)]) \tag{30.61}$$

$$= \frac{1}{2}\mathbf{Var}[f(X_1)]. \tag{30.62}$$

When we averaged two samples, the expectation remained the same, but the variance went down by a factor of two. (The standard deviation went down by a factor of $\sqrt{2}$). If we define Y_n by

$$Y_n = \frac{1}{n}\sum_{i=1}^{n} f(X_i), \tag{30.63}$$

where the X_is are all uniformly distributed on $[a, b]$, and are all independent, then the expected value of $(b-a)Y_n$, for $n = 1, 2, \ldots$, is $\int_a^b f(x)\,dx$, while the variance of Y_n is $\frac{1}{n}\mathbf{Var}[Y_1]$.

This sequence of random variables has the property that as n goes to infinity, the variance goes to zero. That (together with the fact that $\mathbf{E}[(b-a)Y_n] = \int_a^b f(x)\,dx$ for every n) makes it a useful tool in estimating the integral: We know that if we take enough samples, we'll get closer and closer to the correct value.

To bring these notions back to graphics, when we recursively ray-trace, we find a ray-surface intersection, and then, if the surface is glossy, recursively trace a few more rays from that intersection point (using the BRDF to guide our random choice of recursive rays). At the next pixel in the image, we may hit a nearby point on the glossy surface, and trace a *different* few recursive rays, again chosen randomly. We are using those few recursive ray samples to estimate the total light arriving at the glossy surface (i.e., to estimate an integral). Even if the light arriving at the two nearby points of the surface is nearly identical, our *estimates* of it may not be identical. This leads to the appearance of noise in the image. The fact that choosing more samples leads to reduced variance in the estimator means that if we increase the number of recursive rays sufficiently, the noise they cause in the image will be insignificant.

In general, we've got some quantity C (like the integral of f, above), that we'd like to evaluate, and some randomized algorithm that produces an estimate of C. (Or, on the mathematical side, we have a random variable whose expectation is [or is near] the desired value.) We call such a random variable an **estimator** for the quantity. The estimator is **unbiased** if its expected value is actually C. Generally, unbiased estimators are to be preferred over biased ones, in the absence of other factors.

Estimators, being random variables, also have **variance**. Small variance is generally preferred over large variance. Unfortunately, there tend to be tradeoffs: Bias and variance are at odds with each other.

When we have a sequence of estimators like $Y_1, Y_2, \ldots$ above, we can ask not whether Y_k is biased, but whether the bias in Y_k decreases to zero as k gets large, as does the variance. If both of these happen, then the sequence of estimators is called **consistent.** Clearly, consistency is a desirable property in an estimator: It suggests that as you do more work, you can be confident that the results you're getting are better and better, rather than getting closer and closer to a wrong answer!

These are informal descriptions of estimators, bias, and consistency; making these notions really precise requires mathematics beyond the scope of this book. See, for example, Feller [Fel68].

30.5 Importance Sampling and Integration

Let's return for one more look at the problem of computing the integral of a function f on the interval $[a, b]$, again as a proxy for computing the integral in the reflectance equation. In our previous efforts, we used uniformly distributed random variables to sample from the interval $[a, b]$. Now let's see what happens when we use a random variable with some different distribution g. Since g will favor picking numbers in some parts of $[a, b]$ over others, we can't use the samples to directly estimate the integral as before. Instead, we have to compensate for the effect of g: We do the same computation as before, but include a division by g. The result is the **importance-sampled single-sample estimate** theorem.

Theorem: If $f : [a, b] \to \mathbf{R}$ is an integrable real-valued function, and X is a random variable on the interval $[a, b]$, with distribution g, then $\frac{f(X)}{g(X)}$ is a random variable whose expected value is $\int_a^b f(x)\, dx$.

The proof is almost exactly the same as before:

$$\mathbf{E}\left[\frac{f(X)}{g(X)}\right] = \int_a^b \frac{f(x)}{g(x)} g(x)\, dx; \tag{30.64}$$

$$= \int_a^b f(x)\, dx. \tag{30.65}$$

Inline Exercise 30.9: Suppose that X is uniformly distributed on $[a, b]$. What's the pdf g for the variable X? What does this importance-sampled single-sample estimate say about integrating f using X to produce a sample? Is it consistent with the single-sample estimate theorem? What happened to the extra factor of $(b - a)$?

Just as before, if we use n samples instead of one, the variance of the estimate decreases as $\frac{1}{n}$.

The value of this nonuniform-sampling technique is that if you make the density function g be *exactly proportional* to f, then something quite interesting happens: Each sample of the random variable $\frac{f(X)}{g(X)}$ is the same (i.e., the random variable $\frac{f(X)}{g(X)}$ is a constant). This means that the variance of the estimator is *zero!*

Unfortunately, to make g exactly proportional to f (i.e., $g = Cf$ for some C) and to make it a probability distribution, we need

$$1 = \int_a^b g(x)dx \tag{30.66}$$

$$= \int_a^b Cf(x)dx \tag{30.67}$$

$$= C \int_a^b f(x)dx \tag{30.68}$$

In other words, the constant C is just the inverse of the integral we're hoping to compute. To get the ideal benefit of this approach, we'd need to know the answer to the problem we're trying to solve!

All is not lost, however. Suppose that the function g is larger where f gets larger, and smaller where f gets smaller, etc., albeit not in exact proportion. Then the variance of the weighted-sampling estimator is lower than that of the uniform-sampling approach. The use of such a function g is known as **importance sampling,** and g is sometimes called an **importance function.**

In practice in graphics, for a scene containing only reflection, we're usually trying to estimate the integral that appears in the middle of the rendering equation,

$$\int_{\omega_i \in \mathbf{S}^2_+(P)} L(R(P, \omega_i), -\omega_i) f_r(P, \omega_i, \omega_o) \omega_i \cdot \mathbf{n}(P)\, d\omega_i, \tag{30.69}$$

for a fixed ω_o and P. We know the reflectance function f_r (it's a property of the material at P), but we usually don't have much idea *a priori* about how the factor

L varies as a function of ω_i. But in the absence of other information, the *product* Lf_r is likely to be larger when f_r is large and smaller when f_r is small. We can therefore hope to reduce variance in our estimate by choosing samples in a way that's proportional to f_r, or better still, proportional to $\omega_i \mapsto f_r(P,\ \omega_i,\ \omega_o)\ \omega_i \cdot \mathbf{n}(P)$, which we call the **cosine weighted BRDF.** This may not be possible, but it may be possible to choose them in a way that's at least *related* to f_r or the cosine weighted BRDF. This can help reduce variance substantially. In fact, it's easy to see where this kind of importance sampling will help most.

- If the surface is mostly lit by a few point lights, then the variation in L will usually dominate the variation in f_r in determining where the integrand is large or small.
- If the surface is mostly lit by diffusely reflected light coming from all different places, but is itself highly specular, then the shape of f_r dominates in determining the size of the integrand, and importance sampling with respect to the cosine weighted BRDF is likely to reduce variance a lot.

Unfortunately, when we build a rendering system, we don't necessarily know what kind of scenes we'll be rendering. It would be nice to be able to combine the two strategies (use the cosine weighted BRDF as the importance function, or use some approximation of the arriving light field as the importance function) in a way that varies according to the particular situation. A technique called **multiple importance sampling** (see Section 31.18.4) allows us to do just that.

30.6 Mixed Probabilities

We've discussed discrete and continuum probabilities, but there's a third kind, which we'll call **mixed probabilities,** that comes up in rendering. They arise exactly from the impulses in bidirectional scattering distribution functions (BSDFs), or the impulses caused by point lights. These are probabilities that are defined on a continuum, like the interval $[0, 1]$, but are not defined strictly by a density. Consider the following program, for example:

```
1 if uniform(0, 1) > 0.6 :
2    return 0.3
3 else :
4    return uniform(0, 1)
```

Sixty percent of the time this program returns the value 0.3; the other 40% of the time it returns a value uniformly distributed on $[0, 1]$. The return value is a random variable that has a **probability mass** of 0.6 at 0.3, and a pdf given by $d(x) = 0.4$ at all other points. We'd *like* to be able to say that the pdf is given by

$$d(x) = \begin{cases} 0.4 & x \neq 0.3 \\ 0.6 \cdot \infty & x = 0.3 \end{cases}, \tag{30.70}$$

as shown schematically in Figure 30.10, but this makes no sense, literally. In the language of Chapter 18, we could instead say that the pdf was the sum of the constant function 0.4 and the delta function $0.6 \cdot \delta(x - 0.3)$. Or we can just say that it's a random variable that has a probability mass at the point 0.3. In any case, a **random variable with mixed probability** is one for which there is a finite

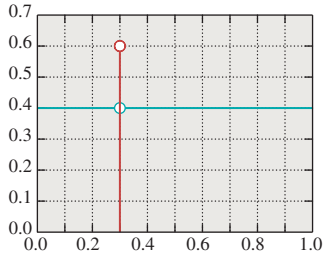


Figure 30.10: A mixed probability. The red stem (vertical line) indicates a probability mass. The blue graph (horizontal line) indicates a density.

set of points in the domain at which the pdf is undefined, but for which there are associated positive probability *masses* rather than densities. The sum of the masses and the integral of the density over the remainder of the space must be 1.0.

The only critical feature of a random variable with a mixed probability is that when we want to integrate it via importance sampling, it's important that we sample the locations of the probability masses with a nonzero probability. Since there are only finitely many probability masses, a good solution is the following.

1. Let $x_1, \ldots, x_n$ be the locations of the probability masses.
2. Let $M = \sum_i m_i \leq 1$ be the sum of the probability masses.
3. Let $u =$ `uniform(0, 1)`; if $u \leq M$, return x_i with probability m_i/M.
4. Otherwise, return some value x using uniform or other sampling methods applicable to nonmixed probabilities.

In the example above, our approach would return $x = 0.3$ 60% of the time, and return other values in the unit interval uniformly at random 40% of the time.

It's not actually essential that the probability of picking x_i be proportional to m_i/M, but it's an easy choice that makes the remaining computations in importance-weighted integration, for instance, much easier. We'll see this applied in Chapter 32.

30.7 Discussion and Further Reading

The key result of this chapter is the importance-sampled single-sample estimate theorem, with which we can estimate the integral of a function f over a region R by $\frac{f(X)}{g(X)}$, where X is a random variable on the region R with distribution g. It lies at the heart of both ray-tracing and path-tracing algorithms. But also important are the notions of consistency and bias.

The use of Monte Carlo methods for integration is described, fairly densely, in Spanier and Gelbard's classic book [SG69], where it's applied to the study of neutron transport rather than photon transport.

30.8 Exercises

The first four exercises are designed to reinforce your understanding of Monte Carlo integration both without and with importance sampling.

Exercise 30.1: A friend picks three positive numbers A, B, and C and then draws the function graph shown in Figure 30.11. You don't know the values A, B, and C. You get to ask your friend one question of the form "What is the value of $f(s)$?" for some particular s. Given these constraints, you are to estimate the integral of f over $[0, 3]$. Your approach is to flip a three-sided coin that lands on side $i = 1, 2,$ or 3. You ask your friend, "What's $f(i - \frac{1}{2})$?" You multiply his answer by three and use this as an estimate of the integral. The value produced is evidently a random variable, X.

(a) What's the expected value $\mu = \mathbf{E}[X]$?

(b) What's the variance of X? Express your answer in terms of A, B, C, and μ.

(c) Under what conditions on A, B, and C is the variance zero? Under what conditions is it large compared to μ?

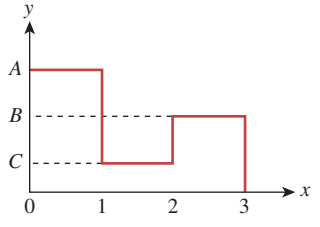


Figure 30.11: The function we'll integrate.

Exercise 30.2: Continuing the preceding problem, suppose that instead of having a three-sided coin, you have a slightly broken random number generator, `xrand`, which returns a random number between 0 and 3. Seventy percent of the time it's in $[0,1]$, uniformly distributed; 20% of the time it's in $[1,2]$, uniformly distributed; and 10% of the time it's in $[2,3]$, uniformly distributed. You're given a single sample t generated by `xrand`, and you ask your friend to tell you $y = f(t)$.
(a) Show how to use the value y to estimate the integral as before. Hint: Use importance sampling.
(b) Compute the variance of your estimate.
(c) When, qualitatively, would you expect the variance to be larger than that of the preceding problem's estimator? When would you expect it to be lower?
(d) Write a short program to confirm that the expected value and variance really are what you predicted.

Exercise 30.3: Now imagine that instead of a function with three values, you had a function that took on infinitely many, such as $h(x) = 1 + (\frac{x}{2})^2$.
(a) Compute the integral of h over $[1,3]$ using calculus.
(b) Estimate the integral of h over $[1,3]$ by writing a small program that picks a random number x uniformly in the interval, evaluates h there, and multiplies by $3-1$. Run the program 100 times and average the results, and compare with your answer to part (a).

Exercise 30.4: In Equation 30.29, we showed that the pdf p_W for the random variable W satisfied $\int_a^b p_W(r)\,dr = b^2 - a^2$ for every a and b in the interval $[0,1]$.
(a) Write $f(b) = \int_a^b p_W(r)\,dr$, and compute $f'(b)$ in terms of p_W using the Fundamental Theorem of Calculus.
(b) Since $f(b) = b^2 - a^2$ as well, compute $f'(b)$ in a different way.
(c) Conclude that $p_W(r) = 2r$ for any $r \in [0,1]$.

Exercise 30.5: Importance sampling. (a) Compute the integral of $f(x) = x^2$ on the interval $[0,1]$ using calculus.
(b) Use stochastic integration (with uniformly distributed samples) to estimate the integral, using $n = 10$, 100, 1,000, and 10,000 samples.
(c) Plot the error as a function of the number of samples. Repeat three times.
(d) Do the same computation, but for $f(x) = \cos^2(x)e^{-20x}$.
(e) Use nonuniformly distributed samples to estimate the integral again, where the probability density of generating the sample x is proportional to $s(x) = e^{-20x}$. To do this, you'll need to determine the constant of proportionality; fortunately, that's easy because s is easy to integrate on the interval $[0,1]$. You'll also need to generate samples with density proportional to s; the simplest approach is to generate a uniform sample, $u \in [0,1]$, and compute $x = -\ln(u)/20$. If x is larger than 1, ignore it and repeat the process.
(f) Does this produce better estimates of the integral? Try to give an intuitive explanation of your conclusion.

Exercise 30.6: Consistency and bias. Consider a random variable $X \sim U(0,1)$. We saw in the chapter that we can estimate its mean by averaging n samples; this estimator will be unbiased. But consider the estimator

$$Y_n = \frac{1}{n}\left(1 + \sum_{i=1}^{n} X_n\right), \tag{30.71}$$

where the X_i are iid $\sim \mathbf{U}(0,1)$.
(a) Show that each Y_n is a biased estimator of the mean $\bar{X}$.

(b) Show that the sequence $Y_1, Y_2, \ldots$ is a consistent estimator of the mean.
(c) Construct a sequence Z_n of unbiased estimators that are *not* consistent. (Hint: Consistency requires *two* things.)

Exercise 30.7: Rejection sampling. In many cases where sampling from a distribution directly is difficult, **rejection sampling** is a last-resort solution that's guaranteed to work, albeit very slowly in some cases. Figure 30.12 shows the idea: Drawing a box around the graph of a density function d, we select points uniformly randomly in the box. The x-values of these points are treated as samples, except that (x, y) is rejected (i.e., not used) if $y > d(x)$. In areas where $d(x)$ is large, a sample (x, y) is likely to be accepted; where $d(x)$ is small, it's likely to be rejected. When a sample's rejected, we continue to generate new samples until we find an acceptable one.

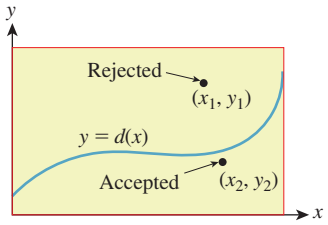


Figure 30.12: We draw a box around the graph of our probability density function d, and choose a point (x, y) uniformly randomly in the box. If (x, y) lies under the graph, we return x; if not, we try again.

(a) Write a program that uses rejection sampling to generate 10,000 samples from the distribution $d(x) = x$ on the interval $[0, 2]$ and plot your results in the form of a histogram.
(b) What fraction of your attempts at generating samples were rejected?
(c) Repeat with the distribution $d(x) = \frac{20}{1-\exp(-20)} \exp(-20x)$ to see how badly rejection sampling can work.
(d) Use the *idea* of rejection sampling (pick samples from a too-large space, and then select only the good ones) to generate points uniformly in the unit disk, and plot your results.
(e) Use the same idea to sample from the unit ball in 3-space, and the unit ball in 10-space. How well (in terms of rejection) does the last of these work?
(f) Points in the unit ball in any dimension are uniformly distributed in direction (with the exception of the origin). Use this to take your points-in-a-ball sampler and make a points-on-a-sphere sampler, being sure to reject the special case of the origin. Experimentally compare its efficiency to that of the hemisphere sampler we described. (Depending on your computer's architecture, the comparison could go either way.)
(g) When you want to rejection-sample the function $x \mapsto 1+x$ on the interval $[0, 1]$, you draw a box $[0, 1] \times [a, b]$ for some values a and b. What are the constraints on possible values for a and b? What happens if you make b quite large?

Exercise 30.8: Suppose that X is a random variable on $[0, 1]$ with density $e : [0, 1] \to \mathbf{R}$, and that $f : [a, b] \to [0, 1]$ is a bijective increasing differentiable function.
(a) Show that $t \mapsto e(f(t))f'(t)$ is a probability density on $[a, b]$.
(b) Suppose that Y is a random variable distributed according to $t \mapsto e(f(t))f'(t)$. Suppose that $t_0 \in [a, b]$ and $x_0 = f(t_0)$. How are $\mathbf{Pr}\{t_0 - \epsilon \leq Y \leq t_0 + \epsilon\}$ and $\mathbf{Pr}\{x_0 - \epsilon \leq X \leq x_0 + \epsilon\}$ related, for small values of ϵ? This whole problem is merely an exercise in chasing definitions—there should be no difficult mathematics involved.

Exercise 30.9: Use the linearity of expectation repeatedly to show that $\mathbf{Var}[X] = \mathbf{E}[X^2] - \mathbf{E}[X]^2$.

Exercise 30.10: Densities. Suppose that $([0, 1], p)$ is a probability space, that is, p is a probability density. We observed that p may take on values larger than 1. In this problem, you'll show that this cannot happen too often. Show that if $p(x) \geq M$ on the interval $a \leq x \leq b \subset [0, 1]$, then $b - a < \frac{1}{M}$. Hint: Write out the probability of the event $[a, b]$ as an integral, and then use the assumption about p to give a lower-bound estimate of this integral.

This page intentionally left blank

Chapter 31

Computing Solutions to the Rendering Equation: Theoretical Approaches

31.1 Introduction

In this chapter we discuss the theory of solving the rendering equation, concentrating on the mathematics of various approaches and on what kinds of approximations are involved in these approaches, deferring the implementation details to the next chapter. Fortunately, much of the mathematics can be understood by analogy with far simpler problems. When we render, we're trying to compute values of L, the radiance field, or expressions involving combinations (typically integrals) of many values of L. Thus, the unknown is the whole *function* L. That's in sharp contrast to the equations like

$$3x^2 + x = 13 \tag{31.1}$$

that we see in algebra class, where the unknown, x, is a single number. Nonetheless, such simple equations provide a useful model for the approximations made in the more complicated task of finding L; we discuss these first, and then go on to apply these ideas to rendering.

31.2 Approximate Solutions of Equations

There's no hope of solving the rendering equation exactly for any scene with even a moderate degree of complexity. Instead, we are forced to *approximate* solutions. There are four common forms of approximation that are routinely used in graphics:

- Approximating the equation
- Restricting the domain

- Using statistical estimators
- And bisection/Newton's method

Because the last of these is not used much in rendering, it'll get brief treatment. The statistical approach, however, which now dominates rendering, will occupy much of the rest of the chapter.

We'll discuss these in the context of a much simpler problem: Find a positive real number x for which

$$50x^{2.1} = 13. \tag{31.2}$$

The numerical solution of this equation is $x = 0.5265\ldots$, but let's pretend that we don't know that, and we're restricted to computations easily done by hand, like addition, subtraction, multiplication, division, and finding integer powers of a real number.

31.3 Method 1: Approximating the Equation

Instead of solving $50x^{2.1} = 13$, which would involve the extraction of a 2.1^{th} root, we could solve a "nearby" equation like

$$50x^2 = 12.5, \tag{31.3}$$

which simplifies to $x^2 = \frac{1}{4}$, and get the answer $x = 0.5$. Since multiplication and exponentiation are both continuous, it should be no surprise that the solution to this slightly "perturbed" equation is quite close to the solution of the original (see Figure 31.1). Solving the perturbed equation is easy.

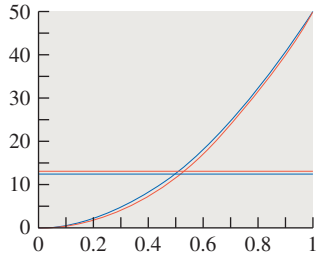


Figure 31.1: The graph of $y = 50x^2$ (blue) is very close to that of $y = 50x^{2.1}$ (just below it, in red); the x-coordinate of the intersection of the blue graph with the line $y = 12.5$ is very near that of the red graph with the line $y = 13$.

You might well complain that the word "nearby" was left undefined in the preceding paragraph. As a different example, consider solving

$$10^{-6}x = 0.1 \tag{31.4}$$

for x. The solution is $x = 10^5$. But if we alter the equation just a little, making the right-hand side 0 instead of 0.1, the solution becomes $x = 0$: A small perturbation in the equation led to a huge perturbation in the solution. Determining the sensitivity of the solution to perturbations in the equation is (for more complicated equations like the rendering equation) often extremely difficult; in practice, it's done by saying things like, "It seems pretty obvious that the moonlight coming through my closed bedroom curtains wouldn't look very different if the moon were oval rather than round." In other words, it's done by using domain expertise to decide which kinds of approximations are likely to produce only minor perturbations in the results.

An example of this in rendering is the approximation of reflection from an arbitrary surface by the Lambert reflection model, or the approximation of the "Is that light source visible from this point?" function, by the function that always says "yes." The first leads to solutions where nothing looks shiny, and the second leads to solutions where there are no shadows; each is often a better approximation than an all-black image, and a poor approximation is frequently better than no solution at all.

31.4 Method 2: Restricting the Domain

Instead of trying to find a positive real number x satisfying

$$50x^{2.1} = 13, \tag{31.5}$$

we can ask, "Is there a positive *integer* satisfying (or nearly satisfying) it?" (See Figure 31.2.) Such a **domain restriction** can simplify things enormously. In the case of this equation, we see that the left-hand side is an increasing function of x, and that when $x = 1$, its value is already 50. So any integer solution must lie between zero and one. We need only try these two possible solutions to see which one works (or, if none works, which is "best"). We quickly find that $x = 0$ gives $50x^{2.1} = 0$, which is too small, and $x = 1$ gives 50, which is too large.

We then have two choices: We can report the "best" solution in the restricted domain ($x = 0$), or we can perhaps say, "The ideal solution lies somewhere between 0 and 1, much closer to 0 than to 1; linear interpolation gives $x = 0.26$ as a best-guess answer." (See Figure 31.3.)

Our use of linear interpolation incorrectly *assumes* that the values of the left-hand side $F(x) = 50x^{2.1}$ vary almost linearly as a function of x between $x = 0$ and $x = 1$, which is why the estimated answer isn't very close to the true one. More generally, if the domain of some variable is D, and we restrict to a subset $D' \subset D$, then estimating a solution in D from approximate solutions in D' requires that D' is "large enough" that any point d of D lies near enough to points of D' that $F(d)$ can be well inferred from values of F at nearby points of D'.

We'll see an example of domain restriction in rendering when we discuss radiosity. Note that methods 1 and 2 both violate the Approximate the Solution principle: they approximate the problem rather than the solution.

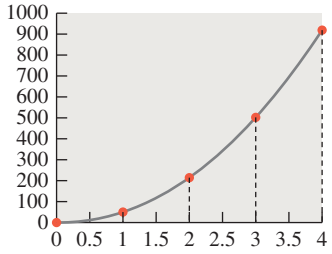


Figure 31.2: The graph of $y = 50x^{2.1}$, restricted to $x = 0, 1, 2, 3, 4$, shown as a stem plot with small red circles, atop the graph on the whole real line (shown in gray).

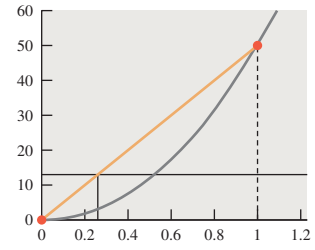


Figure 31.3: Because the value at $x = 0$ is too small, and at $x = 1$ it's too large, we estimate the solution x by intersecting the connect-the-dots plot (orange) with the line $y = 13$ to get $x = 0.26$.

31.5 Method 3: Using Statistical Estimators

A third approach is to "estimate" the solution statistically, that is, find a way to produce a sequence of values $x_1, x_2, \ldots$ such that each x_i is a possible solution, and such that the average a_n of $x_1, x_2, \ldots, x_n$ gets closer and closer to a solution as n gets large.

In this case, we're trying to solve

$$50x^{2.1} = 13, \tag{31.6}$$

whose solution is

$$x = \left(\frac{13}{50}\right)^{\frac{1}{2.1}}. \tag{31.7}$$

This can be easily evaluated on a computer, but we're assuming we lack the ability to compute anything more complicated than an integer power of a real number. (When we look at the rendering equation, the corresponding statement will be, "Suppose we lack the ability to compute anything except an integral number of bounces of a ray of light," which is very reasonable: It's hard to imagine what it might mean to compute 2.1 bounces of a light ray!) We can still find a solution using the binomial theorem, which says that

$$(1+t)^{\alpha} = \sum_{k=0}^{\infty} \binom{\alpha}{k} t^k, \tag{31.8}$$

where

$$\binom{\alpha}{k} = \frac{\alpha \cdot (\alpha - 1) \cdot \ldots \cdot (\alpha - k + 1)}{k!} \tag{31.9}$$

is defined for any *real* number α and for $k = 0, 1, \ldots$. We'll be applying this to the case $\alpha = \frac{1}{2.1}$ and $t = -\frac{37}{50}$, so that $1 - t = \frac{13}{50}$, so that evaluating Equation 31.8 will give us the value of the solution in Equation 31.7.

To do so requires summing an infinite series, however. The great insight is the realization that the sum of an infinite series can be *estimated* by looking at individual elements of the series.

31.5.1 Summing a Series by Sampling and Estimation

We now lay the foundations for all the Monte Carlo approaches to rendering, starting with a few simple applications of probability theory.

31.5.1.1 Finite Series

Suppose that we have a *finite* series

$$A = a_1 + a_2 + a_3 + \ldots + a_{20}, \tag{31.10}$$

and we want to estimate the sum, A. We can do the following: Pick a random integer i between 1 and 20 (with probability $1/20$ of picking each possible number), and let $X = 20a_i$. Then X is a random variable. Its expected value is the weighted average of its values, with weights being the probabilities, that is,

$$E[X] = (1/20)(20a_1) + (1/20)(20a_2) + \ldots + (1/20)(20a_{20}) \tag{31.11}$$

$$= a_1 + a_2 + \ldots + a_{20} \tag{31.12}$$

$$= A. \tag{31.13}$$

We've got a random variable whose expected value is the sum we're seeking! By actually taking samples of this random variable and averaging them, we can approximate the sum.

Inline Exercise 31.1: Suppose that all 20 numbers $a_1, a_2, \ldots$ are equal. What's the variance of the random variable X? How many samples of X do you need to take to get a good estimate of A in this case?

In general, the variance of X is related to how much the terms in the sequence vary: If all the terms are identical, then X has no variance, for instance. It's also related to the way we *chose* the terms, which happens to have been uniform, but we'll use nonuniform samples in other examples later. When we apply these ideas to rendering, we will end up sampling among various paths along which light can travel; the value being computed will be the light transport along the path. Since some paths carry a lot of light (e.g., a direct path from a light source to your eye) and some carry very little, a large variance is present; to make estimates accurate

will require lots of samples, or some other approach to reducing variance. For a basic ray tracer, this means you may need to trace many rays per pixel to get a good estimate of the radiance arriving at a single image pixel.

31.5.1.2 Infinite Series

It's tempting to generalize to infinite series $A = a_1 + a_2 + \ldots$ in the obvious way: Pick a non-negative integer i, and let $X = a_i$; make all choices of i equally probable, and then the expected value of X should be A. There are two problems with this, however. First, there's the missing factor of 20. In the finite example, we multiplied each a_i by 20 because the probability of picking it was $1/20$. This means that in the infinite case, we'd need to multiply each a_i by infinity, because the probability of picking it is infinitesimal. This doesn't make any sense at all. Second, the idea of picking a positive integer uniformly at random *sounds* good, but it's mathematically not possible. We need a slightly different approach, motivated by Equation 31.11, in which each term of the series is multiplied by the probability of picking that term ($1/20$) and by the inverse of that probability (20). All we need to do is abandon the idea of a *uniform* distribution.

To sum the series

$$A = a_1 + a_2 + \ldots, \tag{31.14}$$

we can pick a non-negative integer j with probability $1/2^j$ so that the probability of picking $j = 1$ is $1/2$ and the probability of picking $j = 10$ is $1/2^{10} = 1/1024$. (This particular choice of probabilities was made because it's easy to work with, and it's obvious that the probabilities sum to 1, but any other collection of positive numbers that sum to 1 would work equally well.)

We then let

$$X = 2^j a_j. \tag{31.15}$$

Just as before, the expected value of X is

$$E[X] = \sum_{j=1}^{\infty} \frac{1}{2^j}(2^j a_j) \tag{31.16}$$

$$= \sum_{j=1}^{\infty} a_j \tag{31.17}$$

$$= A. \tag{31.18}$$

And just as before, the variance in the estimate is related to the terms of the series. If a_j happens to be 2^{-j}, then the variance is zero and the estimator is great. If $a_j = 1/j^2$, then the variance is considerably larger, and we'll need to average lots of samples to get a low-variance estimate of the result.

As we said, the particular *choice* we made in picking j—the choice to select j with probability 2^{-j}—was simple, but we could have used some other probability distribution on the positive integers; depending on which distribution we choose, the estimator may have lower or higher variance.

When it comes to applying this approach to rendering, the choice of j will become the choice of "how many bounces the light takes." If we have a scene in which the albedo of every surface is about 50%, then we expect only about half as much light to travel along paths of length $k + 1$ as did along paths of length k.

In this case, assigning half the probability to each successive path length makes some sense. In general, picking the right sampling distribution is at the heart of making such Monte Carlo approaches work well.

31.5.1.3 Solving $50x^{2.1} = 13$ Stochastically

Applying these methods to our particular equation, we know that

$$x = \left(\frac{13}{50}\right)^{\frac{1}{2.1}}, \tag{31.19}$$

and that in general we can transform the right-hand side of the equation using the binomial theorem

$$(1+t)^{\alpha} = \sum_{k=0}^{\infty} \binom{\alpha}{k} t^k. \tag{31.20}$$

Doing so, with $t = \frac{13}{50} - 1 = -\frac{37}{50}$ and $\alpha = \frac{1}{2.1}$, we get

$$x = \left(\frac{13}{50}\right)^{\frac{1}{2.1}} \tag{31.21}$$

$$= \binom{\alpha}{0} + \binom{\alpha}{1}\left(\frac{-37}{50}\right)^{1} + \binom{\alpha}{2}\left(\frac{-37}{50}\right)^{2} + \ldots \tag{31.22}$$

$$= 1 + \frac{\alpha}{1}\left(\frac{-37}{50}\right)^{1} + \frac{\alpha(\alpha-1)}{2!}\left(\frac{-37}{50}\right)^{2} + \ldots. \tag{31.23}$$

Now, to estimate a solution, we pick a positive integer j with probability 2^{-j}, and evaluate the jth term. As we wrote this chapter, we flipped coins and counted the number of flips until heads, generating the sequence 3, 3, 1; our three estimates of x are thus the third, third, and first terms of the series, multiplied by 8, 8, and 2, respectively:

$$x_1 = 8\frac{\alpha(\alpha-1)}{2!}\left(\frac{-37}{50}\right)^{2} \approx -0.5464, \tag{31.24}$$

$$x_2 = x_1 = -0.5464\ldots \tag{31.25}$$

$$x_3 = 2. \tag{31.26}$$

Recall that the correct solution to the problem is 0.5265. The average of our three samples is $x = .3024$, which admittedly is not a very good estimate of the solution. When we used 10,000 terms, the estimate was 0.5217, which is considerably closer (see Figure 31.4).

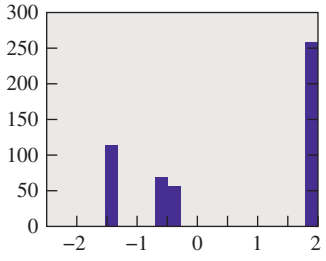


Figure 31.4: A histogram of 500 estimates of the root of $50x^{2.1} = 13$; their average is quite near 0.5625.

You may be concerned that we've assumed we can write a power series for x, but that when we get to the rendering equation, such a rewrite may not be so easy. Fortunately, in the case of the rendering equation, the rewrite as an infinite series is actually *quite* easy, although estimating the sum of the resultant series still involves the same randomized approaches.

31.6 Method 4: Bisection

Our final approach to solving the equation is to find a value of x for which $50x^{2.1}$ is less than 13 (such as $x = 0$) and one for which it's greater than 13 (such as $x = 1$).

Since $f(x) = 50x^{2.1}$ is a continuous function of x, it must take on the value 13 somewhere between $x = 0$ and $x = 1$. We can evaluate $f(\frac{1}{2})$ and find that it's less than 13; we now know that the solution's between $\frac{1}{2}$ and 1. Repeatedly evaluating f at the midpoint of an interval that we know contains the answer, and then discarding half the interval on the basis of the result, rapidly converges on a very small interval that must contain the solution.

This can be seen as a kind of binary search on the real line. There are also "higher order" methods, like Newton's method, in which we start at some proposed solution x_0 and say, "If f were linear, then we could write it as $y = f(x_0) + f'(x_0)(x - x_0)$." But *that* function is zero at $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$. So let's evaluate f at x_1 and see whether it's any smaller there, and iterate. If x_0 happens to be near a root of f, this tends to converge to a root quite fast. If it's not (or if $f'(x_0) = 0$), then it doesn't work so well.

Despite the appeal of these approaches, there's no easy analog in the case of functional equations (ones where the answer is a *function* rather than a number) like the rendering equation. There's no simple way to generalize the notion of one number being between two others to the more general category of functions.

Nonetheless, bisection gets used a lot in graphics, and these four approaches to solving equations serve as archetypes for solving equations throughout the field.

31.7 Other Approaches

There are other approaches to equations that cannot be easily illustrated with our $50x^{2.1} = 13$ example. For instance, you might say, "I can solve systems of two linear equations in two unknowns . . . but only if the coefficients are integers rather than arbitrary real numbers." In doing so, you're not really solving the general problem ("two linear equations in two unknowns"), but it may be that the subclass of problems you *can* solve is interesting enough to merit attention. In graphics, for instance, early rendering algorithms could only work with a few point lights rather than arbitrary illumination; some later algorithms could only work on scenes where all surfaces were Lambertian reflectors, etc.

As a second example, you may arrive at a method of solution that's too complex, and choose to approximate the *method* of solution rather than the original equation. For instance, the Monte Carlo approach used to sum an infinite series above might seem overly complicated, and you might choose to just sum the first four terms. This *sounds* laughable, but in practice it can often work quite well. Most basic ray tracers, for instance, trace secondary rays only to a predetermined depth, which amounts to truncating a series solution after a fixed number of terms.

31.8 The Rendering Equation, Revisited

Recall that a radiance field is a function on the set $\mathcal{M}$ of all surface points in our scene, and that it takes a point, $P \in \mathcal{M}$, and a direction, ω, and returns a real number indicating the radiance along a ray leaving P in direction ω. Our model of the radiance field is a function $L : \mathcal{M} \times \mathbf{S^2} \to \mathbf{R}$.

There is a subtlety here that we discussed in Section 29.4: There are some "surface points" that are part of two surfaces. For instance, if we have a solid glass sphere (see Figure 31.5), the point at the north pole of the sphere is really best

thought of as *two* points: one on the "outside" and the other on the "inside." Light traveling northward at the outer point is either reflected or transmitted, while light traveling northward at the inner point is *arriving* there and is about to be transmitted or reflected by the glass-air interface. As we suggested, we can enhance the notion of the light field to take three arguments—a point, a direction, and a normal vector that defines the "outside" for this point—but in the remainder of this chapter, we're going to instead discuss *only* reflection (except at a few carefully indicated points), since (a) the two-points-in-one-place idea complicates the notation, which is complex enough already, and (b) the actual changes in the programs that we'll see in Chapter 32 to account for transmission are relatively minor and straightforward. As for the matter of keeping two separate copies of the north pole, in practice, as we'll discuss in Chapter 32, we'll only keep a single copy of the geometry, and there will be no explicit representation of the light field; on the other hand, the meaning of an arriving light ray, and how it is treated, will depend on the dot product of its direction with the unit normal $\mathbf{n}$, resulting in several `if-else` clauses in our programs.

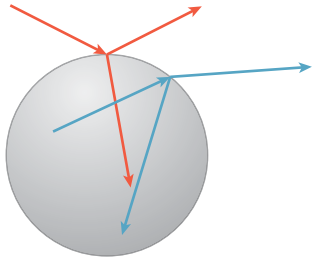


Figure 31.5: Light from above the sphere both reflects and refracts, as does light in the inside of the sphere.

We'll continue to write f_s for the scattering function, however, but you'll need to remember that in the case of transmission, some $\omega \cdot \mathbf{n}$ terms may need absolute-value signs on them.

> Notation: In some papers, L_{in} is $L(P, \omega)$ and L_{out} is $L(P, -\omega)$. Jensen uses L_r for reflected radiance, L_i for incoming radiance, and L_t for transmitted radiance. RTR does the same thing. RTR uses L_i and L_o, with the direction changing based on the subscript. Shirley uses k_i and k_o for our w_i and w_o; Arvo calls L_i and L_o "field" and "surface" radiance. By the way, what we call the radiance field is also called the "plenoptic function" and the "light field."

The rendering equation characterizes the radiance field $(P, \omega) \rightarrow L(P, \omega)$ in a scene by saying that the radiance at some surface point, in some direction, is a sum of (a) the radiance *emitted* at that point in that direction, and (b) all the incoming light at that point that is scattered in that direction. This equation has the form

$$L = E + T(L). \tag{31.27}$$

Recall the meaning of the terms:

- E is the *emitted* radiance field, with $E(P, \omega) = 0$ unless P is a point of some luminaire, and ω is a direction in which that luminaire emits light from P.
- $T(L)$ is the scattered radiance field due to L; $T(L)(P, \omega_o)$ is the light scattered from the point P in the direction ω_o when the radiance field for the whole scene is L.[1]

To be specific, T is defined by

$$T(L)(P, \omega_o) = \int_{\omega_i \in \mathbf{S}^2_+(P)} L(P, -\omega_i) f_s(P, \omega_i, \omega_o)(\omega_i \cdot \mathbf{n}(P))\, d\omega_i. \tag{31.28}$$

1. We use the letter T rather than S (for "scattering") because S will be used later in describing various light paths.

The critical feature of this expression, for our current discussion, is that L appears in the integral.

Note that if we solve Equation 31.27 for the unknown L, we don't yet have a *picture!* We have a function, L, which can be evaluated at a bunch of places to build a picture. In particular, we might evaluate $L(P, \omega)$ for each pixel-center P and the corresponding vector ω from the pinhole of a pinhole camera through P (assuming a physical camera, in which the film plane is *behind* the pinhole).

Inline Exercise 31.2: What are the domain and codomain of T? In other words, what sort of object does T operate on, and what sort of result does it produce? The answer to the latter question is not "It produces real numbers."

 The function T is a higher-order function: It takes in functions and produces new functions. You've seen other such higher-order functions, like the derivative, in calculus class, and perhaps have encountered programming languages like ML, Lisp, and Scheme, in which such higher-order functions are commonplace.

 Let's consider this integral from the computer science point of view. We have a well-defined problem we want to solve ("find L"), and we can examine how difficult a problem this is. First, for even fairly trivial scenes, it's provable that there's no simple closed-form solution. Second, observe that the domain of L is not discrete, like most of the things we see in computer science, but instead is a rather large continuum—there are three spatial coordinates and two direction coordinates in the arguments to L, so it's a function of five real variables. (Note: In graphics, it's common to call this a "five-dimensional function," but it's more accurate to say that it's a function whose domain is five-dimensional.) In computer science terminology, we'd call a classic problem like a traveling salesman problem or 3-SAT "difficult," because the only known way to solve such a problem is no simpler, in big-O terms, than enumerating all potential solutions. By comparison, because of the continuous domain, the rendering equation is even harder, because it's infeasible even to *enumerate* all potential solutions. Your next thought may be to develop a nondeterministic approach to *approximate* the solution. That's a good intuition, and it's what most rendering algorithms do. But unlike many of the nondeterministic algorithms you've studied, while we can characterize the runtime of these randomized graphics algorithms, that in itself isn't meaningful, because the errors in the approximation are unbounded in the general case: Because the domain is continuous, and we can only work with finitely many samples, it's always possible to construct a scene in which all the light is carried by a few sparse paths that our samples miss.

 One strategy for generating approximate solutions is to discretize the domain in some way so that we can bound the error. That's also a good idea, because we might then be able to enumerate some sizable portion of the solution space. I can't look at light transport for every point on a curved surface, but I *can* look at it for every vertex of a triangle-mesh approximation of that surface. Graphics isn't unique in this. The moment you take computer science out of pure theory and start applying it to physics, you'll find that problems are often of more than exponential complexity, and you often need to find good approximations that work well on the general case, even if you can't bound the error in all cases.

Recall from Chapter 29 the division of light in a scene into two categories (see Figure 31.6): At a point of a surface, light may be arriving from various points in the distance, a condition called **field radiance.** This light hits the surface and is scattered; the resultant outgoing radiance is called **surface radiance.**

It's also helpful to divide radiance even further: The surface radiance at a point P can be divided into the emitted radiance there (nonzero only at luminaires) and the reflected radiance. These correspond to the two terms on the right-hand side of the rendering equation. Dually, the field radiance at P can be divided into **direct lighting**, $L^{\mathrm{d}}(P, \omega)$ at P (i.e., radiance emitted by luminaires and traveling through empty space to P), and **indirect lighting,** $L^{\mathrm{i}}(P, \omega)$ at P (i.e., radiance from a point Q to a point P along the ray $P - Q$, but that was *not* emitted at Q). We'll return to these terms in the next chapter.

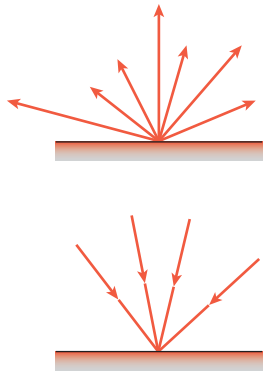


Figure 31.6: Top: The surface radiance consists of all the light leaving a point of a surface. Bottom: The field radiance consists of all the incoming light.

Inline Exercise 31.3: Suppose that P and Q are mutually visible. How are the emitted and reflected radiance at Q, in the direction $P - Q$, related to the direct and indirect light at P, in the direction $P - Q$? Express these in terms of $L^{\mathrm{d}}, L^{\mathrm{i}}, L^{\mathrm{r}}$, and L^{e}, being careful about signs. Use $\omega = \mathcal{S}(P - Q)$ in expressing your answer.

Writing the rendering equation in the form of Equation 31.27 makes it clear that the scattering operator transforms one radiance field (L) into another ($T(L)$). Not only does it do so, but it does so *linearly:* If we compute $T(L_1 + L_2)$, we get $T(L_1) + T(L_2)$, and $T(rL) = rT(L)$ for any real number r, as you can see from Equation 31.28. This linearity doesn't arise from some cleverness in the formulation of the rendering equation. It's a physically observable property, commonly called the principle of superposition in physics, and it's extremely fortunate, for those hoping to solve the rendering equation, that it holds. Later, in Chapter 35, we'll see this principle of superposition applying to forces and velocities and other things that arise in physically based animations, and once again it will simplify our work considerably.

We can rewrite the rendering equation in the form

$$L - T(L) = L^{\mathrm{e}} \tag{31.29}$$

or even

$$(I - T)L = L^{\mathrm{e}}, \tag{31.30}$$

where I denotes the identity operator: It transforms L into L, and we've used TL to denote the application of the operator T to the radiance field L.

Much of the remainder of this chapter describes approaches to solving this equation. Remember as we examine such approaches that L^{e}, the light emitted by each light source, is given as an input, as is the bidirectional reflectance distribution function (BRDF) at each surface point, so that the operator T can be computed. The unknown is the radiance field, L.

The similarity of this formulation to the way eigenvalue problems are described in linear algebra is no coincidence. We'll use many of the same techniques that you saw in studying eigenvalues as we look at solving the rendering equation.

31.8.1 A Note on Notation

We summarize in Table 31.1 the notation we'll use repeatedly throughout this chapter. Figure 31.7 gives the geometric situation to which items in this table refer.

Notice that the subscript "i" in ω_i is set in Roman font rather than italics; that's to indicate that it's a "naming-style" subscript (like V_{IN} to denote input voltage) rather than an "indexing-style" subscript (like b_i, denoting the ith term in a sequence).

When we aggregate over λ, it's important to decide once and for all whether this aggregate denotes a *sum* (perhaps an integral from $\lambda = 400$ nm to 750 nm), or an *average*; you can do either one in your code, but you must do it consistently.

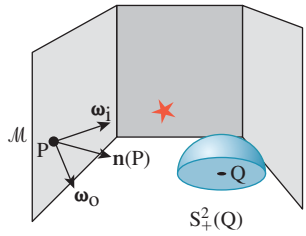


Figure 31.7: Some standard notation. The vector ω_i points toward the light source (indicated by a star).

Table 31.1: Notation used in this chapter.

Symbol	Meaning
$\mathcal{M}$	The set of all surfaces in the scene.
P, Q, R	Points of $\mathcal{M}$.
$\mathbf{n}$	The function that takes a surface point P and returns the normal at P. When the point P is easily understood, we sometimes write $\mathbf{n}$ instead of $\mathbf{n}(P)$.
ω	A generic unit vector in some direction, also thought of as a point in $\mathbf{S^2}$ (the endpoint of ω when it's based at $\mathbf{0}$).
ω_i	A vector from some point P toward incoming light, so the light is propagating in direction $-\omega_i$. In general, $\omega_i \cdot \mathbf{n}(P) > 0$.
ω_o	A direction in which reflected light travels from a point P; in general, $\omega_o \cdot \mathbf{n}(P) > 0$.
$\mathbf{S^2_+}(P)$	The set of all directions ω with $\omega \cdot \mathbf{n}(P) > 0$, that is, the outgoing directions at a point $P \in \mathcal{M}$. Defined only for points on surfaces in the scene.
$\mathbf{S^2_+}$	Shorthand for $\mathbf{S^2_+}(P)$ in the case where the point P is self-evident.
$L(P, \omega)$	The radiance in a scene at point P in direction ω.
$L^e(P, \omega)$	The emitted radiance from a point P on some luminaire in direction ω. $L^e(P, \omega)$ is zero at any point not on a luminaire.
$L^d(P, \omega)$	The light arriving at P along direction ω that was emitted from an emitter toward P along an unobstructed ray between them. This is called the **direct light.**
$L^i(P, \omega)$	The light arriving at P along direction ω that is not a direct light. This is called the **indirect light.**
$L^r(P, \omega)$	The light leaving P in direction ω as a result of scattering rather than emission.
$f_s(P, \lambda, \omega_i, \omega_o)$	The scattering function (BSDF) at the point P, at wavelength λ.
$f_s(P, \omega_i, \omega_o)$	The nonspectral scattering function (BSDF) at the point P (i.e., an aggregate over λ).
$f_s(\omega_i, \omega_o)$	The nonspectral scattering function (BSDF) at a point that's clear from context.
ρ	The *reflectance* of a Lambertian material.

Occasionally we will have several incoming vectors at a point P, and we'll need to index them with names like $\omega_1, \omega_2, \ldots$. When we want to refer to a generic vector in this list, we'll use ω_j, avoiding the subscript i to prevent confusion with the previous use. You will have to infer, from context, that these vectors are all being used to describe incoming light directions, that is, serving in the role of ω_i.

As we discussed in Chapter 26, many terms, and associated units, are used to describe light. In an attempt to avoid problems, we'll use just a few: power (in watts), flux (in watts per square meter), radiance (in watts per square meter steradian), and occasionally spectral radiance (in watts per square meter steradian nanometer).

31.9 What Do We Need to Compute?

Much of the work in rendering falls into a few categories:

- Developing data structures to make the ray-casting operation fast, which we discuss in Chapter 36
- Choosing representations for the function f_s that are general enough to capture the sorts of reflectivity exhibited by a wide class of surfaces, yet simple enough to allow clever optimizations in rendering, which we've already seen in Chapter 27
- Determining methods to approximate the solution of the rendering equation

It is this last topic that concerns us in this chapter.

The rendering equation characterizes the function L that describes the radiance in a scene. Do we really need to know everything about L? Presumably radiance that's scattered off into outer space (or toward some completely absorbing surface) is of no concern to us—it cannot possibly affect the picture we're making. In fact, if we're trying to make a picture seen from a pinhole camera whose pinhole is at some point C, the only values we really care about computing are of the form $L(C, \omega)$. To compute these we may need to compute other values $L(P, \eta)$ in order to better estimate the values we care about.

Suppose, however, that we want to simulate an actual camera, with a lens and with a sensor array like the CCD array in many digital cameras. To compute the sensor response at a pixel P, we need to consider all rays that convey light to P—rays from any point of the lens to any point of the sensor cell corresponding to P (see Figure 31.8).

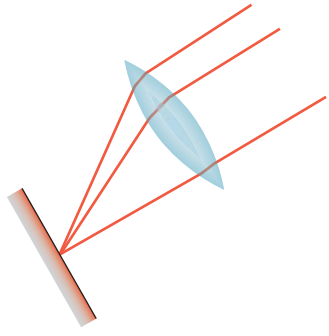


Figure 31.8: Light along any ray from the lens to the sensor cell contributes to the measured value at that cell.

As we said in Chapter 29, light arriving along different rays may have different effects: Light arriving orthogonal to the film plane may provoke a greater response than light arriving at an angle, and light arriving near the center of a cell may matter more than light arriving near an edge—it all depends on the structure of the sensor. The measurement equation, Equation 29.15, says that

$$m_{ij} = \int_{U \times \mathbf{S}^2} M_{ij}(P, -\omega) L^{\text{in}}(P, -\omega) |\omega \cdot \mathbf{n}_P| \, dP \, d\omega, \tag{31.31}$$

where M_{ij} is a sensor-response function that tells us the response of pixel (i,j) to radiance along the ray through P in direction $-\omega$.

One perfectly reasonable idealization is that the pixel area is a tiny square, and that M_{ij} is 1.0 for any ray through the lens that meets this square, and 0 otherwise.

Even with this idealization, however, the pixel value that we're hoping to compute is an *integral* over the pixel area and the set of directions through the lens. Even if we assume a lens so tiny that the latter integral can be accurately estimated by a single ray (the pinhole approximation), there's still an area integral to estimate.

One very bad way to estimate this integral is with a single sample, taken at the center of the pixel region (i.e., the simplest ray-tracing model, where we shoot a ray through the pixel center). What makes this approach particularly bad are *aliasing* artifacts: If we're making a picture of a picket fence, and the spacing of the pickets is slightly different from the spacing of the pixels, the result will be large blocks of constant color, which the eye detects as bad approximations of what *should* be in each pixel (see Figure 31.9).

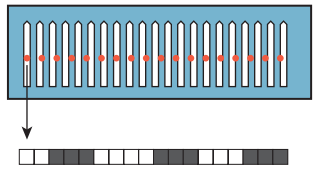


Figure 31.9: Pixel-center samples of a picket-fence scene lead to large blocks of black-and-white pixels.

If we instead take a *random* point in each pixel, then this aliasing is substantially reduced (see Figure 31.10). Instead, we see salt-and-pepper noise in the image.

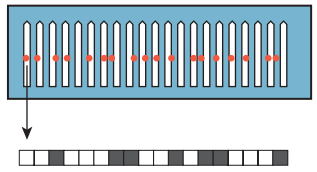


Figure 31.10: Random ray selection within each pixel reduces aliasing artifacts, but replaces them with noise.

Because our visual system does not tend to see "edges" in such noise, but is very likely to see incorrect edges in the aliased image, the tradeoff of aliases for noise is a definite improvement (see Figure 31.11).

This notion of taking many (randomized) samples over some domain of integration and averaging them applies in far more generality. We can integrate over wavelength bands (rather than doing the simpler RGB computations that are so

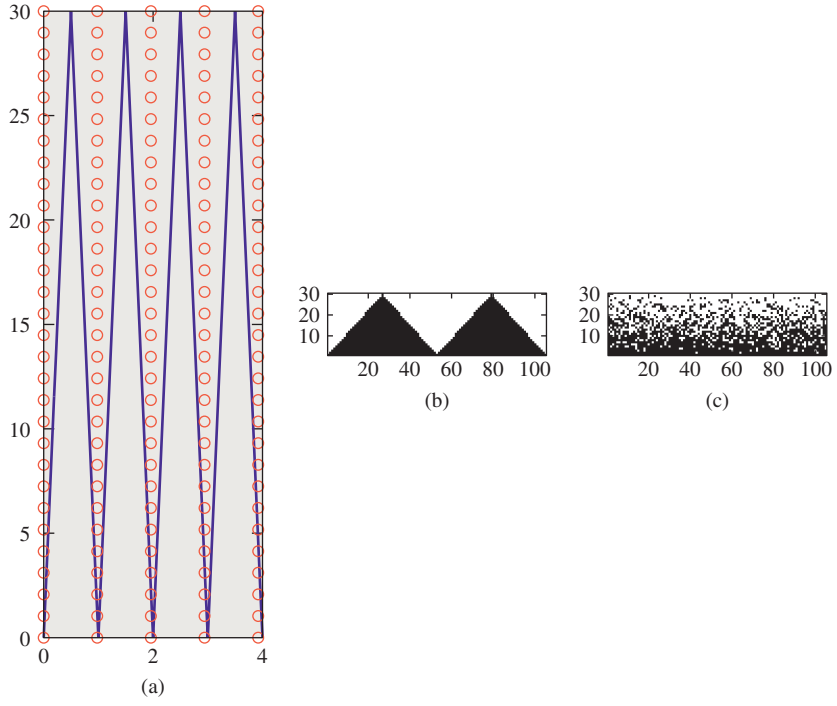


Figure 31.11: (a) A close-up view of a portion of a sawtooth-shaped geometry (note that each sawtooth occupies a little more than one unit on the x-axis) and the locations of pixel samples (small circles). (b) The resultant image. Even though there are 102 teeth in this 104-pixel-wide image, aliasing causes us to see just two. (c) When we take "jittered samples" (each sample is moved up to a half pixel both vertically and horizontally), the resultant image is noisy, but exhibits no aliasing.

common, which amount to a fixed-sample strategy). We can integrate over the lens area to get depth-of-field effects and chromatic aberration. For a scene that's moving, we can simulate the effect of a shutter that's open for some period of time by integrating over a fixed "time window." All of these ideas were described in a classic paper by Cook et al. [CPC84], which called the process distributed ray tracing. Because of possible confusion with notions of distributed processing, we prefer the term **distribution ray tracing** which Cook now uses to describe the algorithm [Coo10]. We'll discuss the particular sampling strategies used in distribution ray tracing in Chapter 32.

In short: To render a realistic image in the general case, we need to average, in some way, many values, each of which is $L(P, \omega)$ for some point P of the image plane and some direction $\omega \in \mathbf{S^2}$.

31.10 The Discretization Approach: Radiosity

We'll now briefly discuss radiosity—an approach that produces renderings for certain scenes very effectively—and then return to the more general scenes that require sampling methods and discuss how to effectively estimate the value $L(P, \omega)$ in the algorithms that work on those scenes.

The radiosity method for rendering differs from the methods we've seen in Chapter 15; in those methods, we started with the imaging rectangle and said, "We need to compute the light that arrives here, so let's cast rays into the scene and see where they hit, and compute the light arriving at the hit point by various methods." Whether we did this one pixel at a time or one light at a time or one polygon at a time was a matter of implementation efficiency. The key thing is that we said, "Start from the imaging rectangle, and use *that* to determine which parts of the light transport to compute." A radically different approach is to simulate the physics directly: Start with light emitted from light sources, see where it ends up, and for the part that ends up falling on the imaging rectangle, record it. This approach was taken by Appel [App68], who cast light rays into the scene and then, at the image plane location of the intersection point (if it was visible), drew a small mark (a "+" sign). In areas of high illumination there were many marks; in areas of low illumination, almost none. By taking a black-and-white photograph of the result (which was drawn with a pen on plotter paper) and then examining the *negative* for the photograph, he produced a rendering of the incident light.

Radiosity takes a similar approach, concentrating first on the light in the scene, and only later on the image produced. Because the surfaces in the scene are assumed Lambertian, the transformation from a representation of the surface radiance at all points of the scene to a final rendering is relatively easy.

The radiosity approach has two important characteristics.

- It's a solution to a *subproblem,* in the sense that it only applies to Lambertian reflectors, and is generally applied to scenes with only Lambertian emitters.
- It's a "discretization" approach: The problem of computing $L(P, \omega_o)$ for every $P \in \mathcal{M}$ and $\omega_o \in \mathbf{S}^2_+(P)$ is reduced to computing a finite set of values. The scene is partitioned into small patches, and we compute a radiosity value for each of these finitely many patches.

The division into patches means that radiosity is a **finite element** method, in which a function is represented as a sum of finitely many simpler functions, each typically nonzero on just a small region. (The word "finite" here is in contrast to "infinitesimal": Rather than finding radiance at every single point of the surface, each point being "infinitesimal," we compute a related value on "finite" patches.)

Radiosity was the first method to produce images exhibiting color bleeding (in which a red wall meeting a white ceiling could cause the ceiling to be pink near the edge where they meet), and not requiring an "ambient term" in its description of reflection—a term included in scattering models (see Chapter 27) to account for all the light in a scene that wasn't "direct illumination," which had presented problems for years previously. Figure 31.12 shows an example.

Figure 31.12: A radiosity rendering of a simple scene. Note the color-bleeding effects. (Courtesy of Greg Coombe, "Radiosity on graphics hardware" by Coombe, Harris and Lastra, Proceedings of Graphics Interface 2004.)

The first step in radiosity is to partition all surfaces in the scene into small (typically rectangular) patches. The patches should be small enough that the illumination arriving at a patch is roughly constant over the patch so that the light leaving the patch will be too, and hence can be represented by a single value. This "meshing" step has a large impact on the final results, which we'll discuss presently. For now, let's just assume the scene surfaces are partitioned into many small patches. We'll use the letters j and k to index the patches, and use A_j to indicate the area of patch j, B_j to denote a value proportional to the radiance leaving any point of patch j, in any outgoing direction[2] ω_o, and $\mathbf{n}_j$ to indicate the normal vector at any point of patch j.

Each patch j is assumed to be a Lambertian reflector, so its BRDF is a constant function,

$$f_s(P, \omega_i, \omega_o) = \rho_j/\pi, \tag{31.32}$$

where ρ_j is the reflectivity and P is any point of the patch. Furthermore, each luminaire is assumed to be a "Lambertian" emitter of constant radiance, that is, $L^e(P, \omega_o)$ is a constant for P in patch j and ω_o an outgoing vector at P.

This simple form for scattering and the assumption about constant emission together mean that the rendering equation can be substantially simplified.

For the moment, let's make four more assumptions. The first is that the scene is made up of closed 2-manifolds, and no 2-manifold meets the interior of any other (e.g., two cubes may meet along an edge or face, but they may not interpenetrate). This also means that we don't allow two-sided surfaces (i.e., a single polygon that reflects from both sides)—these must be modeled as thin, solid panels instead.

For the other three, we let P and P' be points of patch i and Q and Q' be points of patch j, and $\mathbf{n}_i$ and $\mathbf{n}_j$ be the patch normal vectors. Then we assume the following.

- The distance between P and Q is well approximated by the distance from the center C_j of patch j to the center C_k of patch k.
- $\mathbf{n}_j \cdot (P - Q) \approx \mathbf{n}_j(P' - Q')$, that is, any two lines between the patches are almost parallel.
- If $\mathbf{n}_j \cdot \mathbf{n}_k < 0$, then every point of patch j is visible from patch k, and vice versa. So, if two patches face each other, then they are completely mutually visible.

2. By "outgoing direction," we mean that $\omega_o \cdot \mathbf{n}_j > 0$; the radiance is independent of direction because the surfaces are assumed Lambertian.

We need one more definition: We let Ω_{jk} denote the solid angle of directions from patch j to patch k (see Figure 31.13), assuming that they are mutually visible; if they're not, then Ω_{jk} is defined to be the empty set.

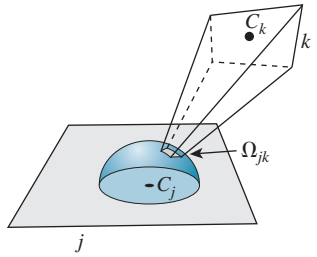


Figure 31.13: Patch k is visible from patch j; when it's projected onto the hemisphere around C_j, we get a solid angle called Ω_{jk}.

Now let's use these assumptions to simplify the rendering equation. Let's start with a point P in some patch j. The rendering equation says that for a direction ω_o with $\omega_o \cdot \mathbf{n}_j > 0$, that is, an outgoing direction from patch j,

$$L(P,\omega_o) = L^e(P,\omega_o) + \int_{\omega_i \in \mathbf{S}^2_+(\mathbf{n}_j)} f_r(P,\omega_i,\omega_o) L(P,-\omega_i)(\omega_i \cdot \mathbf{n}_j)\, d\omega_i. \quad (31.33)$$

We now introduce some factors of π to simplify the equation a bit. We let $B_j = L(P,\omega_o)/\pi$. Since $L(P,\omega_o)$ is assumed independent of the outgoing direction ω_o, the number B_j does not have ω_o as a parameter. Similarly, we define $E_j = L^e(P,\omega_o)/\pi$. And substituting $f_r(P,\omega_i,\omega_o) = \rho_j/\pi$, we get

$$\pi B_j = \pi E_j + \frac{\rho_j}{\pi} \int_{\omega_i \in \mathbf{S}^2_+(\mathbf{n}_j)} L(P,-\omega_i)(\omega_i \cdot \mathbf{n}_j)\, d\omega_i. \quad (31.34)$$

The inner integral, over all directions in the positive hemisphere, can be broken into a sum over directions in each Ω_{jk}, since light arriving at patch j must arrive from some patch k. The equation thus becomes

$$\pi B_j = \pi E_j + \frac{\rho_j}{\pi} \sum_k \int_{\omega_i \in \Omega_{jk}} L(P,-\omega_i)(\omega_i \cdot \mathbf{n}_j)\, d\omega_i. \quad (31.35)$$

The radiance in the integral is radiance leaving patch k, and is therefore just πB_k. Substituting, and rearranging the constant factors of π a little, we get

$$\pi B_j = \pi E_j + \frac{\rho_j}{\pi} \sum_k \int_{\omega_i \in \Omega_{jk}} \pi B_k (\omega_i \cdot \mathbf{n}_j)\, d\omega_i \quad (31.36)$$

$$= \pi E_j + \frac{\rho_j}{\pi} \pi \sum_k \left(\int_{\omega_i \in \Omega_{jk}} (\omega_i \cdot \mathbf{n}_j)\, d\omega_i \right) B_k \quad (31.37)$$

$$= \pi E_j + \rho_j \pi \sum_k \left(\frac{1}{\pi} \int_{\omega_i \in \Omega_{jk}} (\omega_i \cdot \mathbf{n}_j)\, d\omega_i \right) B_k. \quad (31.38)$$

Dividing through by π, we get

$$B_j = E_j + \rho_j \sum_k \left(\frac{1}{\pi} \int_{\omega_i \in \Omega_{jk}} (\omega_i \cdot \mathbf{n}_j)\, d\omega_i \right) B_k. \quad (31.39)$$

The coefficient of B_k inside the summation is called the **form factor** f_{jk} for patches j and k. So the equation becomes

$$B_j = E_j + \rho_j \sum_k f_{jk} B_k, \quad (31.40)$$

which is called the **radiosity equation.** Before we try to solve it, let's look at the form factor more carefully. For patches j and k, it is

$$f_{jk} = \frac{1}{\pi} \int_{\omega_i \in \Omega_{jk}} (\omega_i \cdot \mathbf{n}_j)\, d\omega_i. \quad (31.41)$$

Using the second assumption (that all rays from patch k to patch j are essentially the same) we see that the vector ω_i can be replaced by $\mathbf{u}_{jk} = \mathcal{S}(C_k - C_j)$, the unit vector pointing from the center of patch j to the center of patch k. Since $\mathbf{u}_{jk} \cdot \mathbf{n}_j$ is a constant, it can be factored out of the integral.

The form factor can then be written:

$$\frac{1}{\pi} \int_{\Omega_{jk}} \omega_i \cdot \mathbf{n}_j \, d\omega_i = \frac{1}{\pi} \int_{\Omega_{jk}} \mathbf{u}_{jk} \cdot \mathbf{n}_j \, d\omega_i; \tag{31.42}$$

$$= \frac{1}{\pi} \left(\int_{\Omega_{jk}} 1 \, d\omega_i \right) \mathbf{u}_{jk} \cdot \mathbf{n}_j. \tag{31.43}$$

The remaining integral is just the measure of the solid angle Ω_{jk}, which is the area A_k of patch k, divided by the square of the distance between the patches (i.e., by $\|C_j - C_k\|^2$), using the third assumption and scaled down by the cosine of the angle between $\mathbf{n}_k$ and $\mathbf{u}_{jk}$ (by the Tilting principle). Thus, the form factor becomes

$$f_{jk} = \frac{1}{\pi} \frac{A_k}{\|C_j - C_k\|^2} |\mathbf{u}_{jk} \cdot \mathbf{n}_j| \cdot |\mathbf{u}_{jk} \cdot \mathbf{n}_k|. \tag{31.44}$$

Inline Exercise 31.4: (a) The form of Equation 31.44 makes it evident that $f_{jk}/A_k = f_{kj}/A_j$. Explain why, if j and k are mutually visible, exactly one of the two dot products is negative.
(b) Suppose that patch k is enormous and occupies essentially all of the hemisphere of visible directions from patch j. What will the value of f_{jk} be, approximately?

If we compute all the numbers f_{jk} and assemble them into a matrix, which we multiply by a diagonal matrix $\mathbf{D}(\rho)$ whose jth diagonal entry is ρ_j, and we assemble the radiosity values B_j and emission values E_j into vectors $\mathbf{b}$ and $\mathbf{e}$, then the radiosity equation, under the assumptions listed above, becomes

$$\mathbf{b} = \mathbf{e} + \mathbf{D}(\rho)\mathbf{F}\mathbf{b}. \tag{31.45}$$

This can be simplified (just like the integral form of the rendering equation) to

$$(\mathbf{I} - \mathbf{D}(\rho)\mathbf{F})\mathbf{b} = \mathbf{e}, \tag{31.46}$$

which is just a simple system of linear equations (albeit possibly with many unknowns).

Standard techniques from linear algebra can be used to solve this equation (see Exercise 31.2). ◈ The existence of a solution depends on the matrix $\mathbf{D}(\rho)\mathbf{F}$ being "small" compared to the identity, that is, having all eigenvalues less than one. This is a consequence of our assumption that all the reflectivities were less than one (and your computation of the largest possible form factor). Note that we do not suggest solving the equation by inverting the matrix; in general that's $O(n^3)$, while approximation techniques like Gauss-Seidel work extremely well (and much faster) in practice.

Computing f_{jk} is a once-per-scene operation. Once the matrix **F** is known, we can vary the lighting conditions (the vector **e**) and then recompute the emitted radiance at each patch center (the vector **b**) quite quickly.

Once we know the vector **b**, how do we create a final image, given a camera specification? We can create a scene consisting of rectangular patches, where patch j has value B_j, and then rasterize the scene from the point of view of the camera. Instead of computing the lighting at each pixel, we use the value stored for the surface shown at that pixel: if that pixel shows patch j, we store the value πB_j at that pixel. The resultant radiance image is a radiosity rendering of the scene.

This, however, is rarely done as described; such a radiosity rendering looks very "blocky," while we know from experience that totally Lambertian environments tend to have very smoothly varying radiance. Instead of rendering the computed radiance values directly, we usually *interpolate* them between patch centers, using some technique like bilinear interpolation, or even some higher-order interpolation. This is closely analogous to the approach discussed in Section 31.4, in which we solved an equation on the integers and then interpolated to guess a solution on the whole real line. In this case, we've found a piecewise-constant function (represented by the vector **b**) that satisfies our discretized approximation of the rendering equation, but we're displaying a *different* function, one that's *not* piecewise constant.

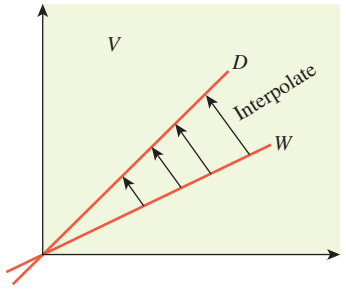


Figure 31.14: Schematically, the space V of all surface-radiance fields contains a subspace W of piecewise constant fields, and another subspace D of piecewise linear fields. There's a map from W to D defined by linear interpolation.

What we've done is to take the space V of *all* possible surface radiance fields, and consider only a subset W of it, consisting of those that are piecewise constant on our patches. We've approximated the equation and found a solution to this in W; we've then transformed this solution (by linear interpolation) into a *different* subspace D consisting of all piecewise-linear radiance fields. If D and W are "similar enough," then this is somewhat justified (see Figure 31.14).

One way to address this apparent contradiction is to not assume that the radiance is piecewise constant, and instead assume it's piecewise linear, or piecewise quadratic, and do the corresponding computations. Cohen and Wallace [CWH93] describe this in detail.

The computation of form factors is the messiest part of the radiosity algorithm. One approach is to render the entire scene, with a rasterizing renderer, five times, projecting onto the five faces of a **hemicube,** (the top half of a cube as shown in Figure 31.15). Rather than storing a radiance value at each pixel, you store the *index* k of the face visible at that pixel. You can precompute the projected solid angle for each "pixel" of a hemicube face once and for all; to compute the projected solid angle subtended by face k, you simply sum these pixel contributions over all pixels storing index k. For this to be effective, the hemicube images must have high enough resolution that a typical patch projects to hundreds of hemicube "pixels"; as scene complexity grows (or as we reduce the patch size to make the "constant radiance on each patch" assumption more correct), this requires hemicube images with increasingly higher resolution.

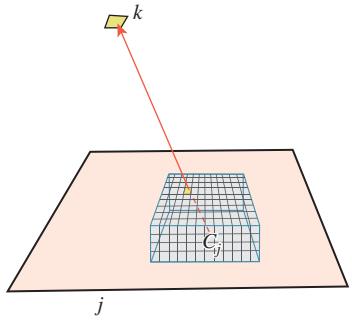


Figure 31.15: We project the scene onto a hemicube around P; since patch k is visible from P through the pixel shown, the hemicube image at that pixel stores the value k.

In solving the radiosity equation, Equation 31.46, some approximation techniques do not use every entry of **F**; it therefore makes sense to compute entries of **F** on the fly sometimes, perhaps caching them as you do so. Various approaches to this are described in great detail by Cohen and Wallace [CWH93].

Before we leave the topic of radiosity, we should mention four more things.

First, in our development, we assumed that patches were completely mutually visible; the hemicube approach to computing form factors removes this requirement. On the other hand, the hemicube approach *does* assume that the solid angle

subtended by patch k from the *center* of patch j is a good representation of the solid angle subtended at any other point of patch j. That's fine when j and k are distant, but when they're nearby (e.g., one is a piece of floor and another is a piece of wall, and they share an edge) the assumption is no longer valid. The form-factor computation must then be written out as an integral over all points of the two patches, and even for simple geometries it has, in general, no simple expression. Schroeder [SH93] expresses this form factor in terms of (fairly) standard functions, but the expression is too complex for practical use.

Second, meshing has a large impact on the quality of a radiosity solution; in particular, if there are shadow edges in the scene, the final quality is far better if the mesh edges are aligned with those shadow edges, or if the patches near those edges are very small (so that they can effectively represent the rapid transition in brightness near the edge). Lischinski et al. [LTG92] describe approaches to precomputing meshes that are well adapted to representing the rapid transitions that will appear during a radiosity computation.

A different approach to the meshing problem is to examine, for each patch, the assumption of constant irradiance across the patch. We do this by evaluating the irradiance at the corners of the patch and comparing them. If the difference is great enough, we *split* the patch into two smaller patches and repeat, thus engaging in **progressive refinement.** This approach is not guaranteed to work: It's possible that, for some patch, the irradiance varies wildly across the patch but *happens* to be the same at all corners; in this case, we *should* subdivide, but we will not. One thing that's fortunate about this approach is that when we subdivide, there's relatively little work to do: We need to compute the form factors for the newly generated subpatches and remove the form factors for the patch that was split. We also need to take the current surface-radiance estimate for the split patch and use it to assign new values to the subpatches; it suffices to simply copy the old value to the subpatches, although cleverer approaches may speed convergence.

Third, although radiosity, as we have described it, treats only pure-Lambertian surfaces and emitters, one can generalize it in many directions: Instead of assuming that outgoing radiance is independent of direction, one can build meshes in both position *and* direction (i.e., subdivide the outgoing sphere at point C_i into small patches, on each of which the radiance is assumed constant); this allows for more general reflectance functions, but it increases the size of the computation enormously. Alternatively, one can represent the hemispherical variation of the emitted light in some *other* basis, such as **spherical harmonics**; an expansion in spherical harmonics is the higher-dimensional analogue of writing a periodic function using a Fourier series. Ramamoorthi et al. [RH01] have used this approach in studying light transport. In each case, specular reflections are difficult to handle: We either need very tiny patches on the sphere's surface, or we need very high-degree spherical harmonics, both of which lead to an enormous increase in computation. One approach to this problem is to separate out "the specular part" of light transport into a separate pass. Hybrid radiosity/ray-tracing approaches [WCG87] attempt to combine the two methods, but this approach to rendering has largely given way to stochastic approaches in recent years.

Fourth, we've been a little unfair to radiosity. The simplification of the rendering equation under the assumption that all emittance and reflectance is Lambertian is the true "radiosity equation." It's quite separate from the division of the scene into patches, and the resultant matrix equation. Nonetheless, the two are often discussed together, and the matrix form is often called the radiosity equation. Cohen

and Wallace's first chapter discusses radiosity in full generality, and treats the discretization approach we've described as just one of many ways to approximate solutions to the equation.

31.11 Separation of Transport Paths

The distinction between diffuse and specular reflections is so great that it generally makes sense to handle them separately in your program. For instance, if a surface reflects half its light in the mirror direction, absorbs 10%, and scatters the remaining 40% via Lambert's law, your code for computing an outgoing scattering direction from an incoming direction will look something like that in Listing 31.1. This is the algorithmic version of the discussion in Section 29.6, and it involves the ideas of mixed probabilities discussed in Chapter 30.

Listing 31.1: Scattering from a partially mirrorlike surface.

```
Input: an incoming direction wi, and the surface normal n
Output: an outgoing direction wo, or false if the light is absorbed

function scatter(...):
   r = uniform(0, 1)

   if (r > 0.5): // this is mirror scattering
     wo = -wi + 2 * dot(wi, n) * n;
   else if (r > 0.1): // diffuse scattering
     wo = sample from cosine-weighted upper hemisphere
   else: // absorbed
     return false;
```

31.12 Series Solution of the Rendering Equation

The rendering equation, written in the form

$$(I - T)L = L^{\mathrm{e}}, \tag{31.47}$$

is an equation of the form

$$\mathbf{Mx} = \mathbf{b} \tag{31.48}$$

that's familiar from linear algebra, except that in place of a vector $\mathbf{x}$ of three or four elements, we have an unknown *function,* L; instead of a target vector $\mathbf{b}$, we have a target function, the emitted radiance field L^{e}; and instead of the linear transformation being defined by multiplication by a matrix $\mathbf{M}$, it's defined by applying some linear operator.  Roughly speaking, the difference is between the finite-dimensional problems familiar from linear algebra, and the infinite-dimensional problems that arise when working with spaces of real-valued functions. (Indeed, the *radiosity* approximation amounts to a finite-dimensional approximation of the infinite-dimensional problem, so the radiosity equation ends up being an actual matrix equation.)

For the moment, let's pretend that the problem we care about really *is* finite dimensional: that L is a vector of n elements, for some large n, and so is L^e, while $I - T$ is an $n \times n$ matrix. The solution to the equation is then

$$L = (I - T)^{-1}(L^e). \tag{31.49}$$

In general, computing the inverse of an $n \times n$ matrix is quite expensive and prone to numerical error, particularly for large n. But there's a useful trick. We observe that

$$(I - T)(I + T + T^2 + \ldots + T^k) = I - T^{k+1}. \tag{31.50}$$

Inline Exercise 31.5: Verify Equation 31.50 by multiplying everything out. Remember that matrix multiplication isn't generally commutative. Why is it OK to swap the order of multiplications in this case?

Suppose that as k gets large, T^{k+1} gets very small (i.e., all entries of T^{k+1} approach zero). Then the sum of all powers of T ends up being the inverse of $(I - T)$, that is, in this special case we can in fact write

$$(I - T)^{-1} = I + T + T^2 + \ldots. \tag{31.51}$$

Multiplying both sides by L^e, we get

$$L = (I - T)^{-1}L^e; \tag{31.52}$$

$$= IL^e + TL^e + T^2L^e + \ldots. \tag{31.53}$$

In words, this says that the light in the scene consists of that emitted from the luminaires (IL^e), plus the light emitted from luminaires and scattered once (TL^e), plus that emitted from luminaires and scattered twice (T^2L^e), etc.

Our fanciful reasoning, in which we assumed everything was finite-dimensional, has led us to a very plausible conclusion. In fact, the reasoning *is* valid even for transformations on infinite-dimensional spaces. The only restriction is that T^2 must be interpreted as "apply the operator T twice" rather than "square the matrix T."

We *did* have to assume that $T^k \to 0$ as k gets large, however. When T is a matrix, this simply means that all entries of T^k go toward zero as k gets large. For a linear operator on an infinite dimensional space, the corresponding statement is that T^kH goes to zero as k gets large, where H is an arbitrary element of the domain of T. (In our case, this means that for any initial emission values, if we trace the light through enough bounces, it gets dimmer and dimmer.)

We'll assume, from now on, that the scattering operator T has the property that $T^k \to 0$ as $k \to \infty$ so that the series solution of the rendering equation will produce valid results.

Of course, the series solution has infinitely many terms to sum up, each of them expensive to compute, so it's not, as written, a practical method for rendering a scene. On the other hand, as we've already seen with radiosity, there are practical approximations to be made based on this series solution.

When do high powers of a linear operator approach the zero operator? We can answer this by looking at eigenvalues: If all eigenvalues are strictly less than one, then $T^k L^e \to 0$ as k goes to ∞. In rendering, this more or less corresponds to there being no perfect reflectors in a scene; indeed, one can imagine a scene consisting of two enormous planar mirrors that face each other, and a point light source between them. Equal amounts of light moving left and right constitute an eigenvector of the light-scattering operator T: After reflection, we once again have equal amounts of light moving left and right. So in this situation, T has an eigenvalue of 1, and iterative computation is not guaranteed to converge. Indeed, if the light source puts out some light, a moment later that light will be reflected by the mirrors and will be added to *new* light sent out by the source, etc., so that the transported light goes to infinity. The unrealistic assumption of perfect mirrors leads to the unrealistic prediction of infinite light transport (and the nonconvergence of the iterative method for solving the equation).

In practice, most surfaces we encounter have relatively low reflectance, and an iterative computation not only converges, it converges fairly quickly. Unfortunately, the convergence isn't necessarily the kind we want: Our estimate of the radiance field L, after a few iterations, may be very close to the true radiance field L_0, but the scene's *appearance* to a human observer might be very different. For instance, if the scene consists of a room lit by a tiny pinhole, behind which there's a light source, the true light in the room is very small ... and therefore very similar to no light in the room; similar, that is, when we compare using the standard mathematical measure of similarity. When we compare using a perceptual metric, the difference is clear: A tiny bit of light when you awaken at night lets you avoid stubbing your toe, while no light at all does not!

31.13 Alternative Formulations of Light Transport

We've described light transport in term of the radiance field, L, which is defined on $\mathbf{R^3} \times \mathbf{S^2}$ or $\mathcal{M} \times \mathbf{S^2}$, where $\mathcal{M}$ is the set of all surface points in a scene. (Since radiance is constant along rays in empty space, knowing L at points of $\mathcal{M}$ determines its values on all of $\mathbf{R^3}$.) And we've used the scattering operator, which transforms an incoming radiance field to an outgoing one in writing the rendering equation. But there are alternative formulations.

Arvo [Arv95] describes light transport in terms of two separate operators. The first operator, G, takes the *surface* radiance on $\mathcal{M}$ and converts it to the *field* radiance, essentially by ray casting: Surface radiance leaving a point P in a direction ω becomes field radiance at the point Q where the ray first hits $\mathcal{M}$. The second operator, K, takes field radiance at a point P and combines it with the BRDF at P to produce surface radiance (i.e., it describes single-bounce scattering locally). Thus, the transport operator T can be expressed as $T = K \circ G$.

Kajiya [Kaj86] takes a different approach in which light directly transported from any point $P \in M$ to any point $Q \in M$ is represented by a value $I(P, Q)$; if P and Q are not mutually visible, then $I(P, Q)$ is zero. Kajiya calls the quantity I the **unoccluded two-point transport intensity.** (The letters I, ρ, ϵ, M, and g used

in this and the following section will not be used again; we are merely explaining the correspondence between his notation and ours.) Kajiya's version of the BRDF is not expressed in terms of a point and two directions, but rather in terms of three points; he writes $\rho(P, Q, R)$ for the amount of light *from* R to Q that's scattered *toward* the point P. His "emitted light" function also has points as parameters rather than point-direction parameters: $\epsilon(P, Q)$ is the amount of light emitted from Q in the direction of P. Kajiya's quantities exclude various cosines that appear in our formulation of the rendering equation, including them instead as part of the integration (his integrals are over the set $\mathcal{M}$ of all surfaces in the scene, while ours are usually over hemispheres around a point; the change-of-variables formula introduces the necessary cosines, as described in Section 26.6.4). Kajiya's formulation of the rendering equation is therefore

$$I(P, Q) = g(P, Q) \left[\epsilon(P, Q) + \int_{R \in M} \rho(P, Q, R) I(Q, R) dR \right], \tag{31.54}$$

where $g(P, Q)$ is a "geometry" term that in part determines the mutual visibility of P and Q: It's zero if P is occluded from Q. Expressing this in terms of operators, he writes

$$I = g\epsilon + gMI, \tag{31.55}$$

where M is the operator that combines I with ρ in the integral. The series solution then becomes

$$I = g\epsilon + gMg\epsilon + gMgMg\epsilon + g(Mg)^3\epsilon + \ldots \tag{31.56}$$

This formulation has the advantage that the computation of visibility is explicit: Every occurrence of g represents a visibility (or ray-casting) operation.

31.14 Approximations of the Series Solution

As we mentioned, summing an infinite series to solve the rendering equation is not really practical. But several approximate approaches have worked well in practice. We follow Kajiya's discussion closely.

The earliest widely used approximate solution consisted (roughly) of the following:

- Limiting the emission function to point lights
- Computing only one-bounce scattering (i.e., paths of the form *LDE*)

That is to say, the approximation to Equation 31.56 used was

$$I = g\epsilon + gM\epsilon_0, \tag{31.57}$$

where ϵ_0 denotes the use of only point lights. (The first term has ϵ, because it was possible to render directly visible area lights.) Note that the second term *should* have been $gMg\epsilon_0$, that is, it should have accounted for whether the illumination could be seen by the surface (i.e., was the surface illuminated?). But such visibility computations were too expensive for the hardware, with the result that these early pictures lacked shadows.

Note that since ϵ_0 consisted of a finite collection of point lights, the integral that defines M became a simple sum.

As an approach to solving the rendering equation, this involves many of the methods described in Section 31.2: The restriction to a few point lights amounts to solving a subproblem. The truncation of the series amounts to approximating the solution method rather than the equation. The transport operator, M, used in the early days was also restricted: All surfaces were Lambertian, although this was soon extended to include specular reflections as well.

Improving the algorithm to use $g\epsilon_0$ instead of ϵ_0 (i.e., including shadows) was a subject of considerable research effort, with two main approaches: exact visibility computations, and inexact ones. Exact visibility computations are discussed in Chapter 36.

A typical inexact approach consists of rendering a scene from the point of view of the light source to produce a **shadow map:** Each pixel of the shadow map stores the distance to the surface point closest to the light along a ray from the light to the surface. Later, when we want to check whether a point P is illuminated by the light, we project P onto the shadow map from the light source, and check whether it is farther from the light source than the distance value stored in the map. If so, it's occluded by the nearer surface and hence not illuminated. This approach has many drawbacks, the main one being that a single sample at the center of a shadow map pixel is used to determine the shadow status of all points that project to that pixel; when the view direction and lighting direction are approximately opposite, and the surface normal is nearly perpendicular to both, this can lead to bad aliasing artifacts (see Figure 31.16).

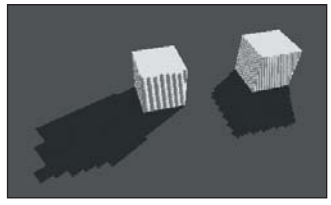


Figure 31.16: Aliasing produced by a low-resolution shadow map. The aliasing on the shadows is the problem; the stripes on the cubes themselves arise from a different problem. (Courtesy of Fabien Sanglard.)

By the way, the approaches used in the early days of graphics were not, at the time, seen as approximate solutions to the rendering equation. They were practical "hacks," sometimes in the form of applications of specific observations (e.g., Lambert's law for reflection from a diffuse surface) to more general situations than appropriate, and sometimes were approximations to the phenomena that were observed, without any particular reference to the underlying physics. When you read older papers, you'll seldom see units like watts or meters; you'll also on rare occasions notice an extra cosine or a missing one. Be prepared to read carefully and think hard, and trust your own understanding.

31.15 Approximating Scattering: Spherical Harmonics

We've discussed patch-based radiosity, in which the field radiance is approximated by a piecewise constant function; one can also think of this as an attempt to write the field radiance in a particular *basis* for a subspace of all possible field-radiance functions, in this case the basis consisting of functions that are identically one on some patch j, and zero everywhere else. Linear combinations of these functions are the piecewise constant functions used in radiosity.

A similar approach is to represent the surface radiance at a point (which is a function on the hemisphere of incoming directions) in some basis for the space of functions on the sphere. Assuming we limit ourselves to continuous functions, such a basis is provided by spherical harmonics, $h_1, h_2, \ldots$, which are the analog, for $\mathbf{S^2}$, of the Fourier basis functions $\sin(2\pi nx)$ $(n = 1, 2, \ldots)$ and $\cos(2\pi nx)$ $(n = 0, 1, 2, \ldots)$ on the unit circle. The first few spherical harmonics,

in xyz-coordinates, are proportional to $1, x, y, z, xy, yz, zx$, and $x^2 - y^2$, with the constant of proportionality chosen so that each integrates to one on the sphere. In spherical polar coordinates, they can be written $1, \cos\theta$, $\sin\theta, \sin\phi$, $\sin 2\theta, \sin\theta\sin\phi$, $\cos\theta\cos\phi$, and $\cos 2\theta$. Like the Fourier basis functions on the circle, they are pairwise orthogonal: The integral of the product of any two distinct harmonics over the sphere is zero. Figure 31.17 shows the first few harmonics, plotted radially. The plot of h_1, which is the constant function 1, yields the unit sphere.

To be clear: If you have a continuous function $f : \mathbf{S^2} \to \mathbf{R}$, you can write f as a sum[3] of spherical harmonics:

$$f(P) = \sum_{j=1}^{\infty} c_j h_j(P). \tag{31.58}$$

The coefficients c_j depend on f, of course, just as when we wrote a function on the unit circle as a sum of sines and cosines, the coefficients of the sines and cosines depended on the function. In fact, they're determined the same way: by computing integrals.

The cosine-weighted BRDF at a fixed point P is a function of two directions ω_i and ω_o, that is, the expression

$$\bar{f}(\omega_i, \omega_o) = f_s(P, \omega_i, \omega_o)\omega_i \cdot \mathbf{n}(P) \tag{31.59}$$

defines a map $\bar{f} : \mathbf{S^2} \times \mathbf{S^2} \to \mathbf{R}$. So the preceding statement about representing functions on $\mathbf{S^2}$ via harmonics does not directly apply. But we *can* approximate the cosine-weighted BRDF $\bar{f}$ at P with spherical harmonics in a two-step process. To simplify notation, we'll omit the argument P for the remainder of this discussion.

First, we fix ω_i and consider the function $\omega_o \mapsto \bar{f}(\omega_i, \omega_o)$; this function on $\mathbf{S^2}$—let's call it F_{ω_i}—can be expressed in spherical harmonics:

$$F_{\omega_i}(\omega_o) = \sum_{j=1}^{\infty} c_j h_j(\omega_o). \tag{31.60}$$

If we chose a different ω_i, we could repeat the process; this would get us a different collection of coefficients $\{c_j\}$. We thus see that the coefficients c_j depend on ω_i; we can think of these as functions of ω_i and write

$$\bar{f}(\omega_i, \omega_o) = \sum_j c_j(\omega_i) h_j(\omega_o). \tag{31.61}$$

Now each function $\omega_i \mapsto c_j(\omega_i)$ is itself a function on the sphere, and can be written as a sum of spherical harmonics. We write

$$c_j(\omega_i) = \sum_k w_{jk} h_k(\omega_i). \tag{31.62}$$

Substituting this expression into Equation 31.61, we get

$$\bar{f}(\omega_i, \omega_o) = \sum_j c_j(\omega_i) h_j(\omega_o); \tag{31.63}$$

$$= \sum_j \sum_k w_{jk} h_k(\omega_i) h_j(\omega_o). \tag{31.64}$$

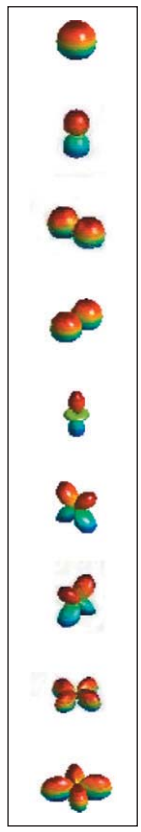


Figure 31.17: The first few spherical harmonics. For each point on the unit sphere $(1, \theta, \phi)$ in spherical polar coordinates, we plot a point (r, θ, ϕ), where $r = |h_j(\theta, \phi)|$. The absolute value avoids problems where negative values get hidden, but is slightly misleading.

3. ◈ Limiting to a finite sum gives an approximation to the function; if f is discontinuous, then the sum converges to f only in regions of continuity.

The advantage of this form of the expression is that when we evaluate the integral at the center of the rendering equation, namely,

$$\int_{\omega_i \in \mathbf{S}^2_+(P)} L(P, -\omega_i) f_s(P, \omega_i, \omega_o) \omega_i \cdot \mathbf{n}(P) d\omega_i, \tag{31.65}$$

both L and f_s are expressed in the spherical harmonic basis. This will soon let us evaluate the integral very efficiently. Note, however, that in expressing the BRDF as a sum of harmonics, we were assuming that the BRDF was continuous; this either rules out any impulses (like mirror reflection), or requires that we replace all equalities above by approximate equalities.

Unfortunately, while L may be expressed with respect to the global coordinate system, f_s is usually expressed in a coordinate system whose x- and z-directions lie in the surface, and whose y-direction is the normal vector. Transforming L's spherical-harmonic expansion into this local coordinate system requires some computation; it's fairly straightforward for low-degree harmonics, but it gets progressively more expensive for higher degrees. If we write the field radiance L at point P in spherical harmonics in this local coordinate system (absorbing a minus sign as we do so),

$$L(P, -\omega_i) = \sum u_m h_m(\omega_i), \tag{31.66}$$

then the central integral takes the form

$$S(\omega_o) = \int_{\omega_i \in \mathbf{S}^2_+(P)} \sum_m u_m h_m(\omega_i) \sum_{jk} w_{jk} h_j(\omega_i) h_k(\omega_o) d\omega_i; \tag{31.67}$$

$$= \sum_k h_k(\omega_o) \int_{\omega_i \in \mathbf{S}^2_+(P)} \sum_m u_m h_m(\omega_i) \sum_j w_{jk} h_j(\omega_i) d\omega_i; \tag{31.68}$$

$$= \sum_k h_k(\omega_o) \sum_{j,m} w_{jk} u_m \int_{\omega_i \in \mathbf{S}^2_+(P)} h_m(\omega_i) h_j(\omega_i) d\omega_i. \tag{31.69}$$

The integral in this expression is 0 if j and m differ, and 1 if they're the same. So the entire expression simplifies to express the surface radiance in direction ω_o as

$$S(\omega_o) = \sum_k h_k(\omega_o) \sum_j w_{jk} u_j. \tag{31.70}$$

The inner sum can be seen as a matrix product between the row vector $\mathbf{u}$ of the coefficients of the field radiance and the matrix of coefficients for the cosine-weighted BRDF. The product vector provides the coefficients for the surface radiance in terms of spherical harmonics in the local coordinate system.

At the cost of expressing the BRDF and field radiance in terms of spherical harmonics, we've converted the central integral into a matrix multiplication. This is another instance of the Basis principle: *Things are often simpler if you choose a good basis*. In particular, if you're likely to be integrating products, a basis like the spherical harmonics, in which the basis functions are pairwise orthogonal, is especially useful.

If the field radiance can be assumed to be independent of position (e.g., if most light comes from a partly overcast sky), then the major cost in this approach is transforming the spherical harmonic expression for field radiance in global coordinates to local coordinates. If not, there's the further problem of converting surface radiance at one point, expressed in terms of spherical harmonics there, into

field radiance at other points, expressed in terms of spherical harmonics at those points.

This approach, which we've only sketched here, has been developed thoroughly by Ramamoorthi [RH01]. There are several important challenges.

- There is a conversion of harmonic decompositions in different coordinate systems, which we've already described.
- We replaced the work of integration with the work of matrix multiplication; that's only a good idea if the matrix size is not too large. Unfortunately, the more "spiky" a function is, the more terms you need in its spherical-harmonic series to approximate it well (just as we need high-frequency sines and cosines to approximate rapidly changing functions in one dimension). Since many BRDFs are indeed spiky (mirror reflectance is a particularly difficult case), a good approximation may require a great many terms, making the matrix multiplication expensive, especially if the matrix is not sparse.
- We've ignored a critical property of irradiance at surfaces: It's nonzero only on the upper hemisphere. That makes the equator generally a line of discontinuity, and fitting spherical harmonics to discontinuities is difficult: A good fit requires more terms.

Ramamoorthi makes a strong argument for much of field radiance being well represented by just the first few spherical harmonics. Sloan [Slo08] has written a useful summary of properties of spherical harmonics, and with Kautz and Snyder [KSS02] has shown how to use them very efficiently in rendering scenes with either fixed view and moving lights, or vice versa.

31.16 Introduction to Monte Carlo Approaches

We now move on to Monte Carlo techniques for solving the rendering equation. The basic idea is to estimate the integral in the rendering equation (or some other integral) by probabilistic methods, which we discussed in Chapter 29. Broadly speaking, we collect samples of the integrand and average them, multiplying the size of the domain of integration.

We begin with a broad and informal view of the various techniques we'll be examining.

Classic ray tracing, which you already saw in Chapter 15, consists of repeatedly casting a ray from the eye and determining the color of the point where it first intersects a surface in the scene. This color is the sum of the illumination from each visible light source (we use ray casting to check visibility), plus illumination from elsewhere in the scene, which we only compute if the intersection point has some specular component, in which case we recursively trace the reflected or refracted ray.

In Chapter 15, we only computed the direct illumination, but modifying a ray tracer to include the recursive part is fairly straightforward. Listing 31.2 gives the pseudocode for point light sources.

The use of "color" here is a shorthand for "spectral radiance distribution," but the main idea is that there's *some* quantity we can compute by evaluating direct illumination at the intersection point, and a remaining quantity that we compute by a recursion. We must generally cast rays from the eye through every pixel in

Listing 31.2: Recursive ray tracing.

```
foreach pixel (x, y):
  R = ray from eyepoint E through pixel
  image[x, y] = raytrace(R)

raytrace(R):
  P = raycast(R) // first scene intersection
  return lightFrom(P,R) // light leaving P in direction opposite R

lightFrom(P, R):
  color = emitted light from P in direction opposite R.
  foreach light source S:
    if S is visible from P:
      contribution = light from S scattered at
                     P in direction opposite R
      color += contribution
  if scattering at P is specular:
      R_new = reflected or refracted ray at P
      color += raytrace(R_new)
  return color
```

the image we're computing, and we often use multiple samples per pixel, with some kind of averaging. But the broad idea remains: Cast rays from the eye; compute direct illumination; add in recursive rays. Ray tracing computes the contributions of paths of the form $LD^?S^*E$.

The essential features of a ray tracer (and all the subsequent algorithms) are a ray-casting function (something that lets us shoot a ray into a scene and find the first surface it encounters), and a BRDF or bidirectional scattering distribution function (BSDF), which takes a surface point P and two rays ω_i and ω_o and returns $f_s(P, \omega_i, \omega_o)$. (In the pseudocode, that BRDF is hidden in the "light from S scattered at P in direction opposite R," which has to be computed by multiplying the radiance from the light by a cosine and the BRDF to get the scattered radiance.) A slightly more sophisticated ray tracer removes the "if scattering is specular" condition, and if the scattering is *non*specular, the scattered radiance is estimated by casting multiple recursive rays in many directions, weighting the returned results by the BSDF according to the reflectance equation. It's this form of ray tracing that we'll refer to henceforth.

Later, in addition to the BRDF or BSDF, we'll also want a function that takes a surface point P and a ray ω_i and returns a random ray ω_o with probability density approximately proportional to the BRDF or BSDF, or cosine-weighted versions of these.

Following Kajiya, we can draw a highly schematic representation of ray tracing (see Figure 31.18). Rays are traced from the eye to the scene; at an intersection, we compute direct lighting and accumulate it as part of the radiance from the light to the eye along the leftmost segment of the path. We also cast recursive rays. In basic ray tracing, we only do so in the specular-bounce direction (if the surface is partly specular); in more sophisticated approaches, we may do so in many directions. When a recursive ray meets a different surface point, the direct lighting there is propagated back to the first intersection, and thence to the eye. This propagation involves *two* scattering operations. Further depths involve further scattering. In the end, we have a branching tree representing the gathering of light into a single pixel of the final image. The branching factor of the tree depends on the number of recursively cast rays at each scattering event.

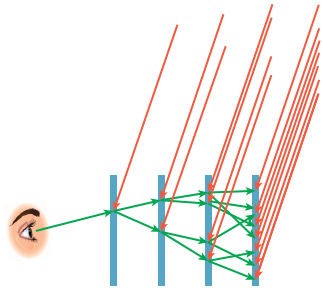


Figure 31.18: Each blue vertical line represents all of $\mathcal{M}$, the set of points of the scene; a green path from the eye (at left) meets $\mathcal{M}$ at some point, and then recursively, multiple rays are traced; these meet the second copy of $\mathcal{M}$, etc. Each branching at such an intersection is a scattering event. The red rays from the upper right to the ray-surface intersections represent light arriving from light sources, which in this schematic representation are placed infinitely far away so that all direct illuminations can be drawn parallel.

In **path tracing,** rays are again cast from the eye, and direct illumination is again computed, but a single recursive ray is also cast, not necessarily in the specular direction: The direction for the recursive ray is determined by some probability distribution on possible directions. The result is to effectively compute the contributions of all paths of the form `L(S|D)*E`. Because of the probabilistic nature of the algorithm, we get noise in the resultant image. To reduce the noise, a great many rays must be cast. The schematic representation (see Figure 31.19) of path tracing is similar to that for ray tracing, except that the branching factor for each tree is *one*. There are, however, many more trees associated to each pixel (although we only draw one).

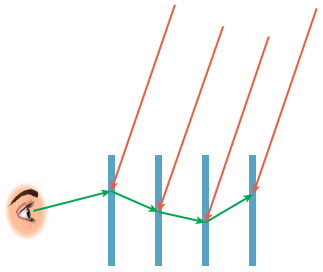


Figure 31.19: In path tracing, each path from the eye either terminates with some probability or traces a single recursive ray. In the version of the algorithm schematically depicted here, direct-light contributions to light transport along the path are computed at every scattering or absorption point, and then transported back along the path.

In **bidirectional path tracing,** one path is traced from the eye and another path is traced from a light source. By splicing together these paths (say, the third point on the light path gets joined to the second point on the eye path), we create paths that may carry light all the way from the source to the eye. If the splice segment meets an occluder, we get no light transport at all, however. Each spliced path gives information about light going to the eye (or camera), and we take the information from multiple paths and splicings to estimate the color of each pixel. Bidirectional path tracing drastically improves the handling of caustics (bright regions arising from paths of the form $\texttt{LS}^+\texttt{DE}$, i.e., light focused in various ways on a diffuse surface). When caustics are seen only in reflection (e.g., with paths of the form LS^+DSE), or when they arise not directly from a light source, but from its reflection in a small diffuse object (LDS^+DE), they are once again difficult to compute, however.

The schematic representation of bidirectional path tracing (see Figure 31.20) consists of *two* trees, one starting from a light and one starting from the eye, with splices between all pairs of interior nodes. Again, we have to understand that there are many paths for each image pixel and many paths emanating from each light source, so a more accurate schematic would consist of two *forests* with many possible splices.

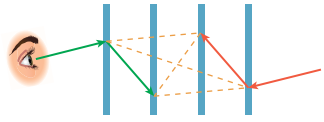


Figure 31.20: In bidirectional path tracing, we compute many eye paths (green) and many light paths (red), and then consider all possible "splices" (orange) between the two sets.

In **photon mapping,** the forest of paths starting at lights is treated differently: Rather than joining an eye path, the nodes of this forest—each node represents light that arrives at some point of the scene—are used to *estimate* the light arriving at *any* point. The light conducted by an eye path can then be evaluated by summing up the light arriving at each of its nodes, that arriving light being estimated from the forest of light paths. One very simple way to estimate the incident light at a point is to look for the nearest point for which the incident light *is* known and use that. This "nearest neighbor interpolation" is not quite what the photon mapping algorithm uses, but it's related. The schematic representation (see Figure 31.21) contains a cloud of estimated incident light into which eye paths reach.

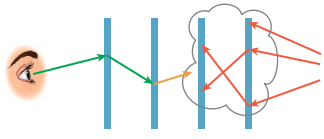


Figure 31.21: In photon mapping, the forest of light paths is used to estimate the incident light at every *surface point by a kind of local interpolation. Eye paths then get incoming radiance values from this estimate, which we draw as a cloud around the leaves of the light-path trees.*

One last algorithm we'll discuss—Metropolis light transport—doesn't directly fit into this schematization. It *does* involve bidirectional path tracing, but the paths are chosen and used in a rather different way.

As described informally above, we'll be recursively tracing lots of rays, forming paths in a scene. In Monte Carlo methods, these paths are generated through a randomized process, usually by sampling from some distribution related to the BRDF.

The two main forms of sampling we tend to do are very similar.

- In ray tracing, we take a ray from the eye to a point P on some surface, and ask, "Which rays arriving at P contribute light that will reach the eye?" For

a mirror surface, the answer is "the mirror direction"; for a diffuse surface, it's "light coming from any direction could produce light in the eye-ray direction." For others, the answer lies somewhere in between. We need, given the direction ω_o, to be able to draw samples from $\mathbf{S}^2_+$ in a way that's proportional to $\omega_i \mapsto f_s(\omega_i, \omega_o)$ (perhaps multiplied by a cosine factor).

- In methods like photon mapping, where we "push" light out from sources rather than "gathering it toward the eye," we instead need to address the problem: "Given that some light arrived at this surface point P from direction ω_i, it will be scattered in many directions. Randomly provide me an outgoing direction ω_o where the probability of selecting ω_o is proportional to $f_s(\omega_i, \omega_o)$."

Clearly these two problems are closely related.

The collecting of samples is done with a **sampling strategy.** What we've outlined above—"Give me a sample that's proportional to something"—comes up both in the Metropolis algorithm and in importance sampling. Other sampling strategies can be used in other approaches to integration. Sometimes it's important to be certain that you've got samples over a whole domain, rather than accidentally having them all cluster in one area; in such cases, stratified sampling, Poisson disk sampling, and other strategies can generate good results without introducing harmful artifacts.

Regardless of the sampling approach that we use, the values we compute are always random variables, and their properties, as estimators of the corresponding integrals, are how we can measure the performance of the various algorithms.

Before we proceed, here's a brief review of what we learned about Monte Carlo integration (following Kellerman and Szirmay-Kalos [KSKAC02]).

To compute the integral of a function f over some domain H, we represent it as an expected value of a random variable:

$$\int_H f(x)\,dx = \int_H \frac{f(x)}{p(x)} p(x)\,dx \tag{31.71}$$

$$= \mathbf{E}\left[\frac{f(x)}{p(x)}\right], \tag{31.72}$$

where p is a probability density on the domain H. To estimate the expected value, we draw N mutually uncorrelated samples according to p and average them, resulting in

$$\int_H f(x)\,dx = \mathbf{E}\left[\frac{f(x)}{p(x)}\right] \tag{31.73}$$

$$\approx \frac{1}{N}\sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}, \tag{31.74}$$

where the standard deviation of this approximation is $\frac{\sigma}{\sqrt{N}}$, σ^2 being the variance of the random variable $f(X)/p(X)$. By choosing p to be large where f is large and small where f is small, we reduce this variance. This is called **importance sampling,** with p being the importance function. (If the samples are correlated, the variance reduction is not nearly as rapid; we'll return to this later.)

With this in mind, let's begin with a general approach to approximating the series solution to the rendering equation.

31.17 Tracing Paths

In solving $L = L^e + TL$ for the unknown L, we found that the solution could be written

$$L = L^e + TL^e + T^2L^e + \dots \tag{31.75}$$

where L^e represented the emitted radiance from luminaires, T represented the transport operator, and L was the total radiance in the scene.

Let's look at ray tracing in this framework: We take a single ray that enters the eye and find the surface point it hits; we then compute direct lighting at that point, and recursively trace more rays, which hit more surface points, where we again compute direct lighting, etc. We are, at the level of computation, building a branching structure: If we trace, on average, n recursive rays at each scattering, then after k steps of scattering, we are tracing n^k rays. If the average reflectance is $\rho < 1$, then the direct lighting at the start of each such ray sequence suffers about ρ^k attenuation by the time it contributes to the light reaching the eye. For large n, this attenuation is substantial, which means that we're doing lots of work ($O(n^k)$) to gather up illumination that amounts to a tiny (ρ^k) fraction of the final result.

Kajiya observed that in many scenes most of the light reaching the eye undergoes at most a few scattering events, and proposed a modified approach, called **path tracing,** shown schematically in Figure 31.19 as described earlier.

In path tracing, the indirect illumination at a point P is estimated not with n samples, but with a single sample! In this approach, the work done grows linearly with k, the depth of recursive ray tracing. Of course, it's possible that the single sample misses something important; to resolve this, one can repeatedly trace the same ray from the eye: Each time it arrives at the first scattering event, the recursive sampler will probably choose a different recursive ray. (Indeed, in practice it makes more sense to gather many samples over the area associated to a single pixel in the image [CPC84], so the recursive rays will almost certainly be different.) With the drastically reduced number of recursively traced rays compared to direct ray tracing, we can afford to instead spend this time tracing multiple rays from the eye for each pixel. On average, only a small fraction of the computation time spent on paths of length 1 (directional lighting) is spent on paths of length 2, etc.; this means that only a small amount of effort is expended in estimating the relatively small contribution of indirect illumination.

So far we've said nothing about how the path lengths are chosen. In the usual ray-tracing model—pick some recursive depth k and apply recursive ray tracing through k bounces—we will never get contributions from any paths of length more than k, and we will always underestimate the radiance. Hence ray tracing is *biased* and *inconsistent:* Even if we trace many rays per pixel, the more-than-k-bounces radiance will never get counted. Path tracing removes the restriction on path length, and thus the corresponding source of bias and inconsistency.

There are two algorithmic drawbacks to path tracing, however. First, the algorithm trades increased accuracy of the mean against increased variance. Having an estimator that can get you the right answer is a great thing, but if you have a small computational budget, then producing an image that's wrong, but more aesthetically pleasing, may be preferable to producing one that is, on average, correct, but looks very noisy. Second, to reduce bias and move toward a consistent estimator of the final image, the algorithm suffers an **unbounded** worst-case runtime, although in practice this is rarely a significant problem. The linear-versus-exponential

arguments we made earlier don't tell the whole story. The key idea that Kajiya (and Cook et al.) put forth is that while the time to trace any given ray is about the same, the amount of *information* that you get from some rays is greater than for others, and you'd like to favor those. If you measure the difference from the true mean of a pixel in a path-traced image that used r rays (total, not just primary rays) and an image produced using r rays in the exponential fan-out pattern, you'll find that the path-traced image is usually closer to the true mean, that is, the bias is smaller. Kajiya actually leverages the variable importance of rays in several ways; here we'll just focus on the strategy of sending one (terminal) direct illumination ray and one (recursive) indirect illumination ray at each surface.

31.18 Path Tracing and Markov Chains

We indicated earlier that computing the series solution

$$L = (1 + T + T^2 + \ldots)L^{\mathrm{e}} \tag{31.76}$$

$$= L^{\mathrm{e}} + \sum_{k=1}^{\infty} T^k L^{\mathrm{e}} \tag{31.77}$$

to the rendering equation

$$L(P, \omega_{\mathrm{o}}) = L^{\mathrm{e}}(P, \omega_{\mathrm{o}}) + \int_{\mathbf{S}^2} L(P, -\omega_{\mathrm{i}}) f_{\mathrm{s}}(P, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}) |\omega_{\mathrm{i}} \cdot \mathbf{n}| \, d\omega_{\mathrm{i}}, \tag{31.78}$$

where L^{e} is the emitted radiance, L is the radiance, and T is the transport operator, was closely related to computing

$$\mathbf{x} = \mathbf{b} + \sum_{k=1}^{\infty} \mathbf{M}^k \mathbf{b}, \tag{31.79}$$

as the solution to the equation

$$\mathbf{x} = \mathbf{b} + \mathbf{M}\mathbf{x}, \tag{31.80}$$

where $\mathbf{b}$ and $\mathbf{x}$ are n-vectors and $\mathbf{M}$ is an $n \times n$ matrix. The analogy is that T is a linear operator on the space of all possible radiance functions, while multiplication by $\mathbf{M}$ is a linear operation on the vector space $\mathbf{R}^n$; in the case where we approximated the space of possible radiance functions by those that are piecewise constant on a fixed mesh, and where the transport operation involved only Lambertian reflection (the radiosity approximation), the analogy was exact: Instead of solving an integral equation, we had to solve a finite-dimensional matrix equation.

We'll consider the case where $\mathbf{M}$ is a 2×2 matrix, chosen so that its eigenvalues are both less than one in magnitude, which makes the series converge, and so that its entries are all non-negative, because they're meant to correspond, roughly, to values of the BSDF in the rendering equation, and these values are never negative.

In the next several pages we'll estimate the value of x_1, the first entry of the solution. We can, of course, solve Equation 31.80 directly, given any particular 2×2 matrix $\mathbf{M}$, but you should imagine that $\mathbf{M}$ could have a thousand or a trillion rows rather than just two; it's in that case that the methods described here work best.

We're going to describe two approaches to finding x_1: one nonrecursive and the other recursive. The first is somewhat more complex, and you can skip it if you'd like. The reasons for including it are as follows.

- It provides the justification for the second method.
- The particular formulation is one that you will see often in modern rendering research papers.

31.18.1 The Markov Chain Approach

If we attempt to find the value of x_1 using Equation 31.79, we'll need to sum up infinitely many terms. The first few are

$$b_1, \tag{31.81}$$

$$m_{11}b_1 + m_{12}b_2, \text{ and} \tag{31.82}$$

$$(m_{11}m_{11} + m_{12}m_{21})b_1 + (m_{11}m_{12} + m_{12}m_{22})b_2. \tag{31.83}$$

The last term, expanded out, is

$$m_{11}m_{11}b_1 + m_{12}m_{21}b_1 + m_{11}m_{12}b_2 + m_{12}m_{22}b_2. \tag{31.84}$$

From Section 31.5.1, we have a method for estimating such an infinite sum: Select a random term a_i in the series (according to some probability distribution p on the positive integers); then $a_i/p(i)$ is an estimator for the sum. In this application, we'll treat each *individual* summand in Equations 31.81–31.83 as a term, so the first term is b_1, the second is $m_{11}b_1$, and so on.

If you look at the sequence of subscripts in any term—say, $m_{12}m_{21}b_1$—you'll notice the following.

- The subscripts start at 1, because we're computing the *first* entry of the answer, x_1.
- Each subsequent subscript is repeated twice (in this case, two more twos, then two more ones). This repetition is a consequence of the definition of matrix multiplication.
- *Every* such sequence occurs.

Thus, to pick a term at random, we need only to pick a "random" finite sequence of subscripts starting at 1. We'll do so by a slightly indirect method.

Figure 31.22 shows a probabilistic finite-state automaton (FSA). The edge from node i to node j is labeled with a probability p_{ij}, which you can read as the probability that if we're at node i at some moment, we'll next move to node j. (In a probabilistic FSA of this sort, the path you take through the FSA is not determined by an input string, but instead by random choices at each node.)

By starting at node 0 and making random transitions to other states, using the probabilities p_{ij} that label the edges of the graph, we'll eventually arrive at state 3, where we'll stop. The sequence we generate will start at 0 and end at 3. In between will be a sequence of 1s and 2s that we'll use as our index sequence.

An FSA like this is a visual representation of a **Markov chain**—a sequence of states with transition probabilities, with the constraint that the probability of being at some state i at step $k + 1$ depends *only* on where you were at step k, and not on any history of your prior steps. Such a Markov chain can be completely described by the matrix $\mathbf{P}$ of transition probabilities p_{ij}, which has the **Markov property:**

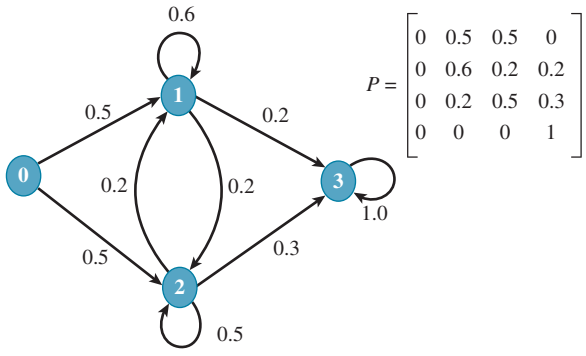


Figure 31.22: A probabilistic FSA with four states, 0, 1, 2, *and* 3*;* 0 *is the start state;* 3 *is an absorbing state, and for the remaining states, we have all possible transitions, labeled with probabilities. A graph like this describes a Markov chain. The transition matrix is shown as well.*

Every row sums to 1 (because when you're in state i, you have to transition to *some* state j).

Inline Exercise 31.6: If $\mathbf{P}$ is an $n \times n$ matrix with the Markov property just described, show that the column vector $\begin{bmatrix}1 & \dots & 1\end{bmatrix}^{\mathbf{T}}$ is a right eigenvector of $\mathbf{P}$. What's its eigenvalue?

A typical path from state 0 to state 3 in the FSA looks like 01223; the numbers between the first and last states constitute a subscript sequence that corresponds to a term in our summation. In this example, the term is

$$m_{12}m_{22}b_2. \tag{31.85}$$

Furthermore, such a path has a *probability* of arising from a random walk in the FSA: We write the product of the probabilities for the edges we traveled. In our example, this is

$$p_{01}p_{12}p_{22}p_{23}. \tag{31.86}$$

We can now describe a general algorithm for computing the sum in Equation 31.79 (see Listing 31.3).

Listing 31.3: Estimating matrix entries with a Markov chain.

```
1  Input: A 2 × 2 matrix M and a vector b, both with
2     indices 1 and 2, and the number N of samples to use.
3  Output: an estimate of the solution x to x = Mx + b.
4
5  P = 4 × 4 array of transition probabilities, with indices
6           0,...,3, as in Figure 31.22.
7
8  S_1 = S_2 = 0 // sums of samples for the two entries of x.
9  repeat N times:
10   s = a path in the FSA from state 0 to state 3, so s(0) = 0.
```

```
    k = length(s) - 2.
    p = probability for s // product of edge probabilities
    T = term associated to subscript sequence s(1), s(2), ..., s(k)
    S_{s(1)} += T/p; // increment the entry of S named by s(1)

return (S_1/N, S_2/N)
```

Under very weak hypotheses on the matrix $\mathbf{P}$, this algorithm provides a consistent estimator of $\mathbf{x}$. That is to say, as $N \to \infty$, the values of S_1/N and S_2/N will converge to the values of x_1 and x_2. How fast will they converge? That depends on the matrix $\mathbf{P}$: For some choices of $\mathbf{P}$, the estimator will have low variance, even for small N; for others, it will have large variance. You, as the user of this estimator, get to choose $\mathbf{P}$.

The following exercises give you a chance to think about "designing" $\mathbf{P}$ effectively. While they may seem far removed from rendering, the ideas you get from these exercises will help you understand the approaches taken in path tracing, which we'll discuss next.

Inline Exercise 31.7: (a) Suppose you know that $m_{12} = 0$, and you're choosing the matrix $\mathbf{P}$ in hopes of getting rapid convergence (i.e., low variance in the estimator) of the answer. Is there a reason for picking $p_{12} = 0$? Is there a reason for picking $p_{12} \neq 0$? Experiment with $\mathbf{M} = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{4} \end{bmatrix}$.
(b) The transitions represented by p_{01} and p_{02} tell us how often (i.e., what fraction of the time) we're getting a new estimate of x_1 and how often we're getting a new estimate of x_2. What are reasonable values for these, and why? How would you choose them if you wanted to only estimate x_1?

Inline Exercise 31.8: Write a program to compute the vector $\mathbf{x}$ as in Inline Exercise 31.7, and experiment with different transition matrices $\mathbf{P}$ to see how they affect convergence. Hint: Experiment with the case where $\mathbf{M}$ is a diagonal matrix to see whether you can notice any patterns, and be sure to use only matrices $\mathbf{M}$ whose entries are non-negative and whose eigenvalues have magnitudes less than one. (For complex eigenvalues $a + b\mathbf{i}$, this means $a^2 + b^2 < 1$.)

We advise that you spend the time doing the inline exercise above, preferably in some language like Matlab or Octave, where such programs are easy to write. Merely debugging your program will give you insight into the method we've described.

The "weak conditions" mentioned above are these: First, if $m_{ij} > 0$, then $p_{ij} > 0$. Second, if $b_i \neq 0$, then $p_{i3} > 0$. Together these ensure that any *nonzero* term in our infinite sum ends up being selected with nonzero probability. The dual of these conditions is also relevant: If $b_i = 0$, then p_{i3} should be set to zero as well so that we never end up working with a term whose last factor is zero. Similarly, if $m_{ij} = 0$, we save ourselves some work by picking $p_{ij} = 0$.

The corresponding ideas, when we apply what we've learned to study light transport, are that (1) we must never ignore any light source (the condition on b_i and p_{i3}) and (2) we never want to select a ray direction for which the BSDF is zero, if we can avoid it (the condition on p_{ij} being zero).

31.18.1.1 An Alternative Markov Chain Estimator

When we generated the sequence 01223, we used it to compute a single term of the series, and the associated probability. But there's a slightly different approach, due to Wasow, in which we can use a k-term sequence to generate k different estimates for the sum all at once. It's described in detail, and a rigorous proof of its correctness is given, in Spanier and Gelbard [SG69]. Here we'll give an informal derivation of the method.

The idea is this: When we had generated the partial sequence 012, there were several possibilities. We could have generated a 3 next, ending the sequence, or we could have gone on to generate another 1 or 2. If you imagine walking through the FSA thousands of times, some fraction of those times your initial sequence will be 012. If there are 100 such initial sequences, then about 30 of them will be 0123, because $p_{23} = 0.3$ in our particular FSA. About 20 of them will have the form 0121 . . . , and about 50 will have the form 0122 . . . . Under the basic scheme, the 30 terminating sequences (0123) would each have associated probability $0.5 \cdot 0.2 \cdot 0.3 = (0.5 \cdot 0.20) \cdot 0.3$, where we've put parentheses around everything except the last factor. The associated term in the sum we're estimating is $m_{12}b_2$, and for each of those 30 sequences, our estimate of the sum is

$$\frac{m_{12}b_2}{(0.5 \cdot 0.2) \cdot 0.3}. \tag{31.87}$$

So, among all sequences starting with 012, 30% contribute $\frac{m_{12}b_2}{(0.5 \cdot 0.2) \cdot 0.3}$ as their estimate of the sum. Suppose that instead we made 100% of the sequences contribute the *smaller* value $\frac{m_{12}b_2}{(0.5 \cdot 0.2)}$ (the division by 0.3 has been removed) as *part* of their estimate of the sum. The expected value would be the same, because we'd be averaging either 30 copies of the larger value or 100 copies of the smaller value! Of course, we apply the same logic to every single initial sequence, and we arrive at a new version of the algorithm. Before we do so, however, let's look at the value $\frac{m_{12}b_2}{(0.5 \cdot 0.2)}$ above. It arises as

$$\frac{m_{12}b_2}{(0.5 \cdot 0.2)} = \frac{1}{p_{01}} \frac{m_{12}}{p_{12}} b_2. \tag{31.88}$$

More generally, in a longer index sequence, we'll have a first factor of the form $\frac{1}{p_{0i}}$ for $i = 1$ or 2, followed by several terms of the form $\frac{m_{ij}}{p_{ij}}$, and finally a b_k. The revised algorithm now looks like the code in Listing 31.4.

Listing 31.4: Estimating matrix entries with a Markov chain and the Wasow estimator.

```
1  Input: A 2 × 2 matrix M and a vector b, both with
2     indices 1 and 2, and the number N of samples to use.
3  Output: an estimate of the solution x to x = Mx + b.
4
5  P = 4 × 4 array of transition probabilities, with indices
6           0,...,3, as in Figure 31.22.
7
8  S1 = S2 = 0 // sums of samples for the two entries of x.
```

```
repeat N times:
  s = a path in the FSA from state 0 to state 3, so s(0) = 0.
  (i, value) = estimate(s)
  S_i += value;

return (S_1/N, S_2/N)

// from an index sequence s(0) = 0, s(1) = ..., s(k+1) = 3,
// compute one sample of x_{s(1)}; return sample and s(1).
define estimate(s):
  u = s(1); // which entry of x we're estimating
  T = 1/p_{1u} // accumulated probability
  value = T · b_u
  for i = 1 to k − 1:
    j = s_i
    k = s_{i+1}
    T *= m_{jk}/p_{jk}
    value += T · b_k
  return (u, value)
```

Pause briefly to look at the big picture of this approach to computing $(1+\mathbf{M}+\mathbf{M}^2+\ldots)\mathbf{b}$.

- Depending on the entries p_{0i}, we may dedicate more or less of our time to computing one entry of $\mathbf{x}$ than another.
- Depending on the entries p_{i3}, the average length of a path may be short or long. If it's short, but the powers of the matrix $\mathbf{M}$ don't decrease very fast, then it may take lots of samples to get good estimates of $\mathbf{x}$.
- There's a whole infinity of algorithms encoded in this single algorithm, in the sense that the transition probabilities p_{ij} can be chosen at will, provided the rows of $\mathbf{P}$ sum to one, that $p_{ij} \neq 0$ whenever $m_{ij} \neq 0$, and that $p_{i3} \neq 0$ whenever $b_i \neq 0$.

31.18.2 The Recursive Approach

We'll now build up a recursive approach to estimating the solution to

$$\mathbf{x} = \mathbf{b} + \mathbf{M}\mathbf{x} \tag{31.89}$$

in several steps, all based on the ideas in Chapter 30.

As a warm-up, suppose that you wanted to estimate the sum of two numbers A and B using a Monte Carlo method. You could write

```
define estimate():
  u = uniform(0, 1)   // random number in [0,1] with
                      // uniform distribution
  if (u < 0.5):
    return A / 0.5
  else:
    return B / 0.5
```

This is just an importance-sampled estimate of the sum, with importance values 0.5 and 0.5.

Inline Exercise 31.9: (a) Show that the expected value of the output of this small program is exactly $A + B$.
(b) Modify the program by writing `if (u < 0.3)...`, and adjusting the two 0.5s as needed to get a different estimator for the same value.
(c) Suppose that $A = 8$ and $B = 12$. What fraction p would you use to replace 0.5 in line 4 to ensure that the estimates produced by multiple runs of the program had the smallest possible variance?
(d) What if $A = 0$ and $B = 12$?

Now let's suppose that B is in fact a sum of n terms: $B = B_1 + \ldots B_n$. We can modify the program to handle this:

```
1  define estimate():
2    u = uniform(0, 1)
3    if (u < 0.5):
4      return A/0.5
5    else:
6      i = randint(1, n) // random integer from 1 to n.
7      return B_i / (0.5 * (1/n))
```

We're just using a uniform single-sample estimate of B now as well as the earlier random choice to estimate $A + B$.

Let's apply these ideas to estimating the solution $\mathbf{x}$ to $(\mathbf{I} - \mathbf{M})\mathbf{x} = \mathbf{b}$, which is

$$\mathbf{x} = \mathbf{b} + \mathbf{M}\mathbf{b} + \mathbf{M}^2\mathbf{b} + \ldots \tag{31.90}$$

$$= \mathbf{b} + \mathbf{M}(\mathbf{b} + \mathbf{M}\mathbf{b} + \ldots). \tag{31.91}$$

To keep things simple in what follows, and in analogy with the light-transport application we'll soon examine, we'll assume that all entries of $\mathbf{M}$ are non-negative, the row-sum $r_i = \sum_j m_{ij}$ is less than 1 for all i, and the eigenvalues of $\mathbf{M}$ have magnitude less than 1, so that the series solution converges. We'll also assume that $\mathbf{M}$ is an $n \times n$ matrix rather than just a 2×2 matrix.

Let's use the ideas of the short programs above to estimate x_1, the first entry of $\mathbf{x}$ in Equation 31.90. According to the equation, the value x_1 is the sum of two numbers, b_1 and $(\mathbf{M}\mathbf{x})_1$. That makes it easy to find x_1 using our sum-of-numbers estimator, but only if we already know $\mathbf{x}$! But $\mathbf{M}\mathbf{x}_1$ is a sum: $m_{11}x_1 + \ldots + m_{1n}x_n$. We can estimate this by selecting among these terms at random, as usual. In each term, we know the m_{ij} factor, and we need to *estimate* the x_j factor. So we can estimate x_1 only if we can estimate x_j for any j. This begs for recursion. And as is typical in recursion, the recursion gets easier if we broaden the problem. So we'll write a procedure, `estimate(i)`, that estimates the ith entry of $\mathbf{x}$, and to find x_1 we'll invoke it for $i = 1$.

```
1  define estimate(i):
2    u = uniform(0, 1)
3    if (u > 0.5):
4      return b_i / 0.5
5    else:
6      k = randint(1, n)
7      return m_ik· estimate(k) / (0.5 * (1/n))
```

The book's website has actual code that implements this approach, and compares it with the answer obtained by actually solving the system of equations.

You'll find, if you run it, that it works surprisingly well, at least when the eigenvalues are small, so that the series solution converges quickly.

For those who read the section on Markov chains, this program corresponds (when $n = 2$) to the Markov chain solution with all edges in the graph labeled with probability 0.5: When we're in some state, half the time we terminate and half the time we make a recursive call, putting another factor of 0.5 in the denominator. Because the Markov chain version of the algorithm is guaranteed to produce a consistent estimator, this recursive version must do so as well. For the rest of this section, you can imagine how the recursive forms we write correspond to various Markov chain forms.

Note that although this code is recursive, you can't prove its correctness the same way you would prove that merge sort is correct, using induction: The recursive invocation of `estimate` is not "simpler" than the calling invocation, and there's no base case. The *structure* of the code is recursive, but the *analysis* of it relies on the graph theory and Markov chains. In fact, the program's *not* correct, at least not in the way that merge sort is correct. There's an execution path (one where the random number u always turns out less than 0.5) in which the program executes forever, for instance! Nonetheless, with probability one the program produces an unbiased estimate of x_i, which is all we can hope for with a Monte Carlo algorithm. (Recall from Chapter 30 that "probability one" does not mean certainty: There may be cases that fail, but they are as unlikely as picking, say, e/π when asked to choose a random number in the unit interval.)

Let's now improve the algorithm in two ways. First, the values returned by `estimate(1)` tend to vary a lot: Half of them are $2b_1$ and the other half arise from a recursive call. We can get the same *average* result if we simply return b_1 in all of them:

```
1  define estimate(i):
2    result = b_1
3    u = uniform(0, 1)
4    if (u < 0.5):
5      k = randint(1, n)
6      result += m_ik * estimate(k) / (0.5 * (1/n))
7    return result
```

Now if m_{ij} is small for all j, then the recursive part of the code is likely to produce small results, because of the factor of m_{ik}. It would be nice, in this case, to bias our program toward skipping the recursive part.

Recall that we assumed that $r_i = \sum_j m_{ij}$ was less than one. That means we can change our code to

```
1  define estimate(i):
2    result = b_1
3    u = uniform(0, 1)
4    if (u < r_i):
5      k = randint(1, n)
6      result += m_ik * estimate(k) / (r_i * (1/n))
7    return result
```

That's the final version we'll write for this particular problem. The use of r_i corresponds, in the Markov chain model, to choosing greater or lesser values for p_{i3}; the choice of k as a random integer between 1 and n *could* be improved a bit: We might, for instance, want to choose k with a probability proportional

to m_{ik}. That would reduce variance, but it might also take longer. When we apply this technique to solving the rendering equation, we actually *will* do some clever things in that recursive part of the estimator. It just happens that in the matrix model of the computation, there's no good analog for these subtleties.

Inline Exercise 31.10: Make certain you really understand every piece of the code in this section. Ask yourself things like, "Why is that $1/n$ in the denominator?" and "Why do we multiply by m_{ik} rather than m_{ki}?" Only proceed to the next section when you are confident of your understanding.

31.18.3 Building a Path Tracer

We'll now describe how to build a path tracer in analogy with the recursive version of the linear equation solver, transforming a simple path tracer into one that's increasingly tuned to use its sampling efficiently. The wrapper of the path tracer is quite simple:

```
for each pixel (x,y) on the image plane:
   ω = ray from the eyepoint E to (x, y)
   result = 0
   repeat N times:
      result += estimate of L(E, −ω)
   pixel[x][y] = result/N;
```

There are variations on the wrapper. We could estimate the radiance through various points associated with pixel (x, y), and then weight the results by the measurement equation to get a pixel value. We could, for a dynamic scene, estimate the radiance at (x, y) for various different *times* and average them, generating motion-blur effects, etc. But all of these have at their core the problem of estimating $L(E, -\omega)$; we'll concentrate on this from now on, writing a procedure called `estimateL` to perform this task.

The first version of the estimation code is completely analogous with the matrix equation solver: Because L is a sum of L^e and an integral, we'll use a Monte Carlo estimator to average the two terms. We'll use the generic point C as the one where we're estimating the radiance, but you should think of C as being the eyepoint E, at least until the recursive call.

Figure 31.23 shows the relevant terms. The red arrow at the bottom is $L^e(P, \omega)$ (which for this scene happens to be zero, because the surface containing P is not an emitter).

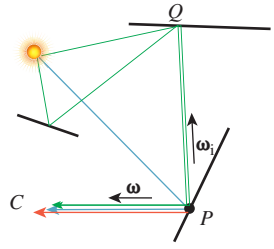


Figure 31.23: Names for some points and paths in our path tracer.

```
// Single-sample estimate of radiance through a point C
// in direction ω.
define estimateL(C, ω):
  P = raycast(C, −ω) // find the surface this light came from
  u = uniform(0, 1)
  if (u < 0.5):
    return L^e(P, ω)/0.5
  else:
    ω_i = randsphere() // unit vector chosen uniformly
    integrand = estimateL(P, −ω_i) · f_s(P, ω_i, ω)|ω_i · n_P|
    density = 1/(4π)
    return integrand / (0.5 * density)
```

The choice of ω_i uniformly on the sphere is analogous to choosing an index $k = 1 \ldots n$, except for one thing: When we chose k, it was possible that $m_{ik} = 0$, that is, the particular path we were following in the Markov chain would contribute nothing to the sum. But when we choose ω_i, we're going to estimate the arriving radiance at P in direction $-\omega_i$ with a *ray cast*. We'll then *know* that P is visible from whatever point happens to be sending light in that direction toward P. This was one of the great insights of the original path tracing paper: that using ray casting amounted to a kind of importance sampling for a certain integral over all surfaces in the scene. (Kajiya performed surface integrals rather than hemispherical or spherical ones.)

Inline Exercise 31.11: If the surface at P is purely reflective rather than transmissive, then half of our recursive samples will be wasted. Assume that `transmissive(P)` returns `true` only if the BSDF at P has some transmissive component. Rewrite the pseudocode above to only sample in the positive hemisphere if the scattering at P is nontransmissive.

Once again, we can replace the occasional inclusion of L^e with an always inclusion. The code is then

```
1  define estimateL(C, ω):
2    P = raycast(C, −ω)
3    u = uniform(0, 1)
4    resultSum = L^e(P, ω)
5    if (u < 0.5):
6      ω_i = randsphere()
7      integrand = estimate(P, −ω_i) * f_s(P, ω_i, ω)|ω_i · n_P|
8      density = 1/(4π)
9      resultSum += integrand / (0.5 * density)
10   return resultSum
```

Now if we suppose that our BSDF can not only be evaluated on a pair of direction vectors, but also can tell us the scattering fraction (i.e., if the surface is illuminated by a uniform light bath, the fraction of the arriving power that is scattered), we can adjust the frequency with which we cast recursive rays:

```
1  define estimateL(C, ω):
2    P = raycast(C, −ω)
3    u = uniform(0, 1)
4    resultSum = L^e(P, ω)
5    ρ = scatterFraction(P)
6    if (u < ρ):
7      ω_i = randsphere()
8      Q = raycast(P, ω_i)
9      integrand = estimate(Q, −ω_i) * f_s(P, ω_i, ω) |ω_i · n|
10     density = 1/(4π)
11
12     resultSum += integrand / (ρ * density)
13   return resultSum
```

The second insight we'll take from Kajiya's original paper is that we can write the arriving radiance from Q as a sum of two parts: radiance emitted at Q (which we call **direct light** L^d at P), and radiance arriving from Q having been scattered after arriving at Q from some other point, which we call **indirect light** L^i. Figure 31.24 shows these. There's no direct light from Q to P in the figure, since Q

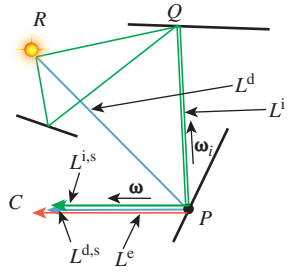


Figure 31.24: The light arriving at P can be broken into direct and indirect light.

is not an emitter, but there is direct light from R to P, shown in cyan. Notice that this division of light arriving at P from Q into different parts is a mathematical convenience. There's no way, when a photon arrives at P from Q, to tell whether it was directly emitted or was scattered. But dividing the arriving light in this way allows us to structure our program to get better results.

The figure also shows the results of scattering these at P, which we'll call $L^{\mathrm{d,s}}$ and $L^{\mathrm{i,s}}$. The scattered radiance L^{r} is just $L^{\mathrm{d,s}} + L^{\mathrm{i,s}}$. Thus,

$$L^{\mathrm{i,s}}(P,\omega) = \int L^{\mathrm{r}}(Q,\omega_{\mathrm{i}}) f_{\mathrm{s}}(Q,\omega,\omega_{\mathrm{i}}) |\omega_{\mathrm{i}} \cdot \mathbf{n}_Q| \, d\omega_{\mathrm{i}}, \text{ and} \tag{31.92}$$

$$L^{\mathrm{d,s}}(P,\omega) = \int L^{\mathrm{e}}(Q,\omega_{\mathrm{i}}) f_{\mathrm{s}}(Q,\omega,\omega_{\mathrm{i}}) |\omega_{\mathrm{i}} \cdot \mathbf{n}_Q| \, d\omega_{\mathrm{i}}, \tag{31.93}$$

where $Q = raycast(P,\omega)$.

With these definitions, we restructure the code slightly (see Listing 31.5).

Listing 31.5: Kajiya-style path tracer, part 1.

```
1  define estimateL(C, ω):
2    P = raycast(C, −ω)
3    resultSum = L^e(P,ω) + estimateLr(P, ω)
4    return resultSum
5
6  define estimateLr(P, ω):
7    return estimateLds(P, ω) + estimateLis(P, ω)
8
9  define estimateLis(P, ω): // single sample estimate
10   u = uniform(0, 1)
11   ρ = scatterFraction(P)
12   if (u < ρ):
13     ω_i = randsphere()
14     Q = raycast(P, ω_i)
15     integrand = estimateLr (Q, −ω_i) * f_s(P,ω_i,ω) |ω_i · n|
16     density = 1/(4π)
17
18     return integrand / (ρ * density)
19   return 0
20
21 define estimateLds(...
```

Notice that in estimating the scattered indirect light, we didn't scatter *all* the radiance arriving at P from Q, but only L^{r}. The L^{e} portion of the radiance is the *direct* light, which we're deliberately not including.

To estimate $L^{\mathrm{d,s}}$ (the scattering of the direct light), we are trying to evaluate the integral

$$L^{\mathrm{d,s}}(P,\omega) = \int L^{\mathrm{e}}(Q,-\omega_{\mathrm{i}}) f_{\mathrm{s}}(P,\omega,\omega_{\mathrm{i}}) \, |\omega_{\mathrm{i}} \cdot \mathbf{n}_P| \, d\omega_{\mathrm{i}}, \tag{31.94}$$

where Q denotes the result of a ray cast from P in direction ω_{i}.

We're going to shift from a spherical integral to a surface integral (which involves the usual change of variable factor), and integrate over all light sources. (You should now think of Q as being a point on the light source in Figure 31.24.) To keep things simple, we'll assume that we have only *area* light sources (no point lights!), and that there are K of them, with areas $A_1, \ldots, A_K$, and total area $A = A_1 + A_2 + \ldots + A_K$.

The integral we want to compute is

$$L^{\mathrm{d,s}}(P,\omega) = \int_{Q \in lights} L^{\mathrm{e}}(Q, -\omega_{PQ}) f_{\mathrm{s}}(P, \omega, \omega_{PQ}) V(P,Q) \frac{|\omega_{PQ} \cdot \mathbf{n}_P||\omega_{PQ} \cdot \mathbf{n}_Q|}{\|Q-P\|^2}\, dQ, \tag{31.95}$$

where $\omega_{PQ} = \mathcal{S}(Q-P)$ is the unit vector pointing from P to Q, and $V(P,Q)$ is the visibility function, which is 1 if P and Q are mutually visible and 0 otherwise. (The extra dot product, and the squared length in the denominator, come from the change of variables.) Notice that in the integrand, we have L^{e} and not L: We only want to estimate scattering of *direct* light.

We can estimate this integral by picking a point Q uniformly at random with respect to area (i.e., with probability A_i/A it'll be a point of light i, and within light i, it'll be uniformly randomly distributed), and using Q to perform a single-sample estimate of the integral:

$$L^{\mathrm{d,s}}(P,\omega) \approx A \cdot L^{\mathrm{e}}(Q, -\omega_{PQ}) f_{\mathrm{s}}(P, \omega, \omega_{PQ})\ V(P,Q) \frac{|\omega_{PQ} \cdot \mathbf{n}_P||\omega_{PQ} \cdot \mathbf{n}_Q|}{\|Q-P\|^2}. \tag{31.96}$$

This makes the code for `estimateLds` fairly straightforward (see Listing 31.6).

Listing 31.6: Kajiya-style path tracer, part 2.

```
1 define estimateLds(P, ω):
2   Q = random point on an area light
3   if Q not visible from P:
4      return 0
5   else:
6      ω_PQ = S(Q − P)
7      geom = |ω_PQ·n_P||ω_PQ·n_Q| / ‖Q−P‖²
8      return A · L^e(Q, −ω_PQ) f_s(P, ω, ω_PQ) · geom
```

Everything we've done here depends on the BSDF being "nice," that is, having no impulses. In the next chapter, we'll adjust the code somewhat to address that.

To summarize, path tracing works by estimating the integrand using a Markov chain Monte Carlo approach, including the reuse of initial segments of the chain for efficiency. It avoids the plethora of recursive rays generated by conventional ray tracing; the time saved is allocated to collecting multiple samples for each pixel. While path tracing is, in the abstract, an excellent algorithm, it does require that you choose an acceptance probability (we've used the scattering fraction) and a sampling strategy for outgoing rays; if these are chosen badly, the variance will be high, requiring lots of samples to reduce noise in the final image. Furthermore, the sampling strategy must be general enough to find all paths in your scene that actually transport important amounts of light. If you allow purely specular surfaces and point lights, then paths of the form LS^+DE are problematic as mentioned above: The path starting from the eye must choose a direction, after the first bounce, that happens to be reflected one or more times to reach the point light. The probability of choosing this direction is unfortunately zero. If you allow nearly point lights and nearly specular reflections, you can still get the same effect: The probability of sampling a good path can be made arbitrarily small. To address this limitation, we have to consider other ways of sampling from the space of all

possible light paths, or else take a great many samples to get a good estimate (i.e., a low-noise image).

What's the difference between a path tracer and a conventional ray tracer? Well, the path-tracing result tends to be noisy, as we mentioned earlier: We're making Monte Carlo estimates of the radiance at each point, and there's variance in these estimates. A basic ray tracer that only computes recursive rays when there's a mirror reflection uses a very *low*-variance estimate of the diffusely scattered indirect light: It estimates it as zero! That makes the basic ray-traced image darker than it should be, but uncluttered with noise. On the other hand, by taking more samples in the path tracer, we can reduce the noise a lot. For a small ray-casting budget (i.e., you can only afford to cast a certain number of rays in your scene), the simple ray-tracer result is wrong but nice looking; the path-traced result is generally "right on average," but noisy. As the ray-casting budget increases, the ray-traced result does not really improve (except for deeper levels of reflection), while the path-traced result gets less and less noise, and correctly includes diffusely scattered indirect light.

Finally, we've treated "measurement" as part of the wrapper for a path tracer, but we could instead include it in the thing being computed, so rather than estimating L, we could estimate L multiplied by the measurement function M. When we do so, the thing we're integrating, expanded out recursively, is a product of some number of scattering functions and cosines, an L^e at the end, and M at the beginning. (If we're integrating with respect to area rather than solid angle, there will also be some change-of-variables factors.) There's a symmetry in this formulation: We could swap the roles of M and L^e, and imagine rendering a scene in which the eye was emitting light according to M, and it was being measured at the light sources using L^e as the measurement function. The integral we'd write down for estimating light transport in this "swapped" scene would be exactly the same as in the original. This provides some theoretical justification for the "trace rays from the eye" approach rather than tracing photons from the lights, the way nature does it: The integrals we're estimating are the same.

31.18.4 Multiple Importance Sampling

When we consider alternative ways to sample from the space of light paths, we may find one sampling strategy that is effective for one class of paths and another that works well for a different class. It's difficult to know in advance which will be useful in any particular scene. Veach developed **multiple importance sampling** as a way to use many different sampling strategies in evaluating a single integral, by weighting samples from the different strategies differently. He describes a motivating example: a single glossy surface reflecting an area light source (e.g., think of a slightly rippled ocean reflecting moonlight). In this case, there are only two interesting kinds of paths: *LE* and *LDE*; if the lights are outside the visible part of the scene, then there are only *LDE* paths. There are no paths of length greater than two in the scene. This makes the analysis particularly easy. We'll look at only the *LDE* paths.

Suppose that we try to estimate the light arriving at some pixel, P. One approach to sampling paths is that we trace a ray through P, meeting the glossy surface at a point x. Then we sample from the BRDF at x and trace a second ray that may hit the area light source, in which case the sample contributes some radiance, or that may miss the light altogether, in which case the sample contributes

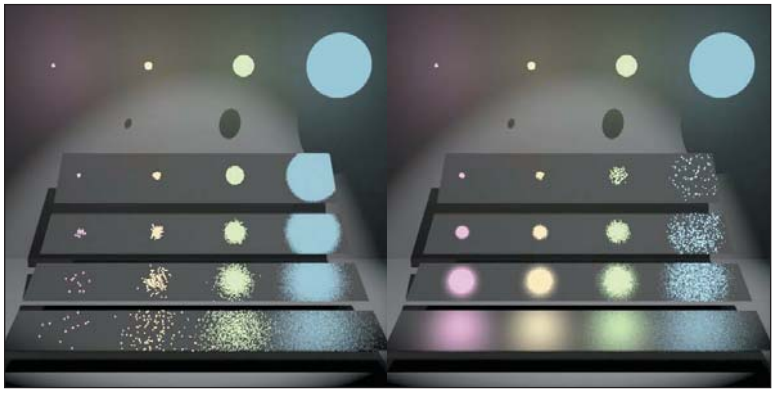


Figure 31.25: The image on the left was made by sampling the BRDF of the rough surface; the nearer slabs are rougher than the distant ones. Four light sources of varying sizes produce different glossy reflections. The image on the right used sampling on the light sources. The upper right in the first picture is preferable to that in the second; the lower left in the second picture is preferable to that in the first. No one sampling strategy is best. (Note that the scene is also lit by a weak light above the camera, and that the slabs all have a small diffuse component, letting us see their general shapes.) (Courtesy of Eric Veach.)

zero. An alternative approach is to trace the ray through P to x, but then sample a point x' uniformly at random on the light source, and connect x to x'. Now the path *certainly* carries some radiance.

The second approach initially seems far better than the first approach, since the paths will always conduct some light. But what if the surface is very rough? Then the BRDF in the xx' direction may be nearly zero, and so the contribution is again very small. Figure 31.25 shows the results in practice.

Clearly we want to use one sampling strategy in estimating the integral in some cases, the other in other cases, and a mix of the two in still other cases. This is where multiple importance sampling comes in. Before we discuss that, there's one point worth noting: If you have two estimators, one with large variance and one with small variance, and you average them, you're in trouble: It's really hard to get rid of the variance except by taking lots of samples. Informally speaking, the central idea of multiple importance sampling is that it provides a way to work with a kind of average of two estimators without letting the larger variance of one of them creep into the later computations.

To describe multiple importance sampling, we return to the abstract setting: We're trying to integrate a function f on some domain D, and we have two different sampling methods that produce samples $X_{1,j}, j = 1, 2, \ldots$ and $X_{2,j}$, $j = 1, 2, \ldots$ with density functions p_1 and p_2, respectively.

To estimate the integral using samples from the two different distributions, we need only produce two weighting functions, w_1 and w_2, from D to $\mathbf{R}$, with two properties:

- $w_1(x) + w_2(x) = 1$ for any x with $f(x) \neq 0$.
- $w_1(x) = 0$ whenever $p_1(x) = 0$, and similarly for w_2 and p_2.

It generally makes sense for both weighting functions to be non-negative, in which case both vary between zero and one.

As a trivial example, at least in the case where each p_i is nonzero everywhere on the domain, we could pick $w_1(x) = 0.25$ for every x, and $w_2(x) = 0.75$ for every x. But more interesting cases arise when the weights are allowed to vary as a function of the samples. We'll return to this in a moment.

Once we have chosen weighting functions, we take n_1 samples of $X_{1,j}$ from the first distribution, and n_2 samples from the second, and combine them in the **multisample estimator** given by

$$F = \frac{1}{n_1}\sum_{j=1}^{n_1} w_1(X_{1,j})\frac{f(X_{1,j})}{p(X_{1,j})} + \frac{1}{n_2}\sum_{j=1}^{n_2} w_2(X_{2,j})\frac{f(X_{2,j})}{p(X_{2,j})}. \tag{31.97}$$

Veach shows that the multisample estimator F is in fact unbiased, and that with suitably chosen weights, it has good variance properties. (An exactly analogous formula works for three, four, or more samplers.)

What are good weight choices? The naive constant-weight approach is one; another is closely related: If we partition the domain D into two subsets D_1 and D_2 with $D_1 \cup D_2 = D$ and $D_1 \cap D_2 = \emptyset$, we can define $w_i(x) = 1$ when $x \in D_i$, and 0 otherwise. This effectively says, "Use one kind of sampling on each part of the domain." One application of this is when we partition the space of paths into, for instance, those with zero, one, two, ... specular bounces, and use a different sampler for each. Another is when we're sampling from a Phong-style BRDF and have to choose between a specular, a glossy, and a diffuse reflection.

31.18.5 Bidirectional Path Tracing

Path tracing makes its choice about extending paths based on the *current* point (i.e., `selectRay` is a function of only x_k). But the actual lighting in a scene may matter as well: If a bright light shines on a dark surface, considerable light may still be reflected. And lots of dim rays reflected even from a low-reflectance surface may converge to form caustics which are perceptually significant. In recognition of this, Lafortune and Willems [Laf96] and Veach [VG94] independently proposed **bidirectional path tracing,** in which paths are traced both from light sources and from the eye. At a naive level, this sounds implausible: There's essentially no chance that two such paths will ever land on the same point so that light is carried all the way from the luminaire to the eye! But a simple trick (see Figure 31.26) addresses this difficulty: We join the two paths with a segment! In fact, we can join any point on the first path to any point on the second path with such a segment, and compute the total transport along the resultant path.

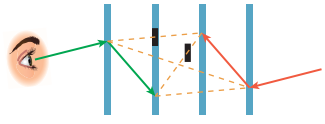


Figure 31.26: A path is traced from the light and the eye; each point of the first path is connected to each point of the second to create paths from the luminaire to the eye. These connecting segments may, however, meet occluders (shown as black segments), creating no net transport.

While the light-path and eye-path segments are all generated by tracing rays, and hence are guaranteed to transport light, the joining segments may meet occluders that make them transport no light at all. This potential occlusion can waste a great deal of path-tracing time and increase the variance of the pixel estimates. When the joining segment is *not* occluded, the computation of the contribution of the resultant path to the total transport is also very complex.

In practice, however, bidirectional path tracing tends to produce quite good results, good enough that they can (with enough samples) be used as a reference standard for evaluating other rendering methods.

To use the traced paths as samples for estimating the integral in the rendering equation, we need to know not only the transport along the path, but also the probability of having generated the path. Computing this probability requires careful

analysis of the program used to generate the paths: How likely are we to have generated the two spliced-together subpaths? How likely are we to have spliced the paths together along this particular edge? These probabilities depend on the sampling approach. For instance, to generate a path, we might do one of the following.

- Repeatedly cast a ray in a uniform random direction (i.e., the probability that the direction lies in some solid angle is proportional to the measure of that solid angle) in the outgoing hemisphere from the current path point.
- Repeatedly cast a ray uniformly with respect to the *projected* solid angle.
- Repeatedly choose a point (toward which we'll cast a ray) at random with respect to the area on the union of all surfaces in the scene.

Each of these will sample some paths more often than others; switching from one to another is similar to changing variables in an ordinary integration problem: An extra factor is introduced in the integrand. And while the geometry term for the "choosing surface points" version has a $\frac{1}{r^2}$ factor, which leads to very large values when two chosen points are close to each other (see Figure 31.27), in the "choosing directions" version this change of variables exactly cancels out the bad factor in the geometry term, *except* for the factor associated with the edge joining the light path to the eye path.

Veach describes how multiple importance sampling can be used to ameliorate this problem.

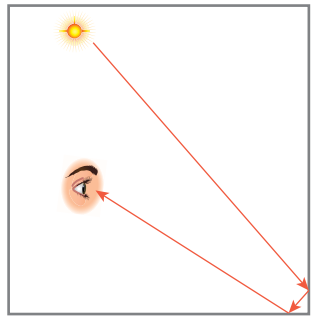


Figure 31.27: If this light path is chosen by selecting points uniformly on surfaces, it will contribute a large value to the integral because of the short segment in the corner and the $\frac{1}{r^2}$ term in the integrand.

31.18.6 Metropolis Light Transport

In 1997, Veach and Guibas [VG97] described the **Metropolis light transport,** or MLT, algorithm. This was yet another Markov chain Monte Carlo approach, but the Markov chain no longer formed a random walk in the set $\mathcal{M}$ of all surface points in the scene, as it did in the path-tracing algorithms; instead, the algorithm takes a random walk in the space of all *paths* in the scene. That is to say, the algorithm may first examine a path of length 1, then a path of length 20, then a path of length 3, etc. Explicitly describing this space of paths, and how to randomly sample from it, is difficult. There's also the unfortunate fact that the Metropolis-Hastings algorithm, on which MLT is based, only provides a result that's guaranteed to be *proportional* to the result you're seeking. Fortunately in MLT, we're seeking a whole array of results (the output pixel values), and the constant of proportionality is the same for all of them. In other words, we get an image that's some constant multiple of the desired image. But once again, there's bad news: The constant of proportionality may be zero, that is, we may get an all-black image. Finally, while MLT's result is guaranteed to be unbiased, it may take a great many paths to ensure that the variance is low (just as with other Monte Carlo methods); the rapidity of variance reduction depends, in part, on how cleverly one chooses a **mutation strategy,** which determines how new paths are generated from the current one. (This is closely analogous to what we saw in computing sums of matrix powers: Picking the transition probabilities p_{ij} dramatically affected the rate of convergence.) Thus, once you develop an MLT renderer, you can improve upon it by adding more and better mutation strategies; such mutation strategies are ways to express your knowledge of the structure of light transport. That is to say, you might argue that if lots of light is being carried along *this* path, then if you *only* move the eye ray a tiny bit, which moves the first interior point of the path, but you leave everything else the same, maybe the new path will also

carry lots of light. The nature of the algorithm makes it easy to express this kind of high-level understanding.

Furthermore, there are some mutation strategies that are relatively easy to implement, but which end up more frequently sampling a "bright" area of the path space that's previously been rarely sampled. Such strategies represent a big improvement in the algorithm.

The details of the algorithm are, unfortunately, rather complex, and involve substantial mathematics. The reader interested in reproducing the results should read Veach's dissertation [Vea97]; indeed, anyone really interested in rendering should do so.

31.19 Photon Mapping

Recall that we described photon mapping as being like bidirectional path tracing, except that rather than connecting an eye path to a light path, we took the final point, P, of the eye path and used the collection of all the light paths to *estimate* the light arriving at P. The problem, of course, is that in any finite set of light paths, there aren't likely to be *any* that end exactly at P. Instead, we have to estimate the arriving light by looking at the arriving light at nearby points and interpolating somehow.

Doing so entails representing all the arriving light in such a way that searching for "nearby" points is easy. In photon mapping, this incoming light is stored in a **photon map,** a relatively compact structure that's not directly related to the scene geometry. Information in the map is stored at points; we'll describe exactly the data stored at each point presently.

Thus, photon mapping has two phases: the construction of the photon map (via **photon tracing**), and the estimation of the outgoing radiance (at many different points) using the photon map.

The process of estimating the radiance *leaving* a surface from the knowledge of the radiance *arriving* at several nearby points involves scattering the arriving light (via the BSDF). One advantage of storing the arriving light (or at least a *sample* of the arriving light) is that it compactly represents a great many outgoing light rays (via the BSDF). A sample of the arriving light is stored in a record that's unfortunately called a **photon,** despite being quite distinct from the elementary particle of the same name. A photon-mapping photon (which is the only kind we'll discuss in the remainder of this section) consists of a location in space, a direction vector ω_i that points *toward* the light source or the last bounce the light took before arriving at this point (i.e., ω_i points opposite the direction of propagation), and an incident power. It's helpful to assign units: The coordinates of the location should be specified in meters; the direction vector is unitless, but has length 1; and the power is specified in watts. We'll denote these by P, ω_i, and Φ_i, respectively. A photon-mapping photon therefore represents an *aggregate* flow of many physical photons per second.

The scene represented by the photon map consists of surfaces and luminaires (which may themselves be surfaces). For each light L we let Φ_L be the emissive power of the light. For example, a 40 W incandescent lamp has an emissive power of about 10 W in the visible spectrum, so for such a bulb, $\Phi_L = 10$ W. And we denote the total power of all light sources by

$$\Phi = \sum_L \Phi_L. \tag{31.98}$$

Surfaces in the scene have a BSDF $f_s(P, \omega_i, \omega_o)$ at each point P of the surface. Because the BSDF may vary with wavelength, we include a fourth parameter: $f_s(P, \lambda, \omega_i, \omega_o)$, with the understanding that λ may represent an actual wavelength, or it may (as in most implementations) represent a band of wavelengths, where the bands are typically denoted by the symbols R, G, and B. The same applies to Φ_L and Φ: Each depends on wavelength. For all three, the omission of the wavelength parameter indicates summation. For instance,

$$f_s(P, \omega_i, \omega_o) := \sum_\lambda f_s(P, \lambda, \omega_i, \omega_o). \tag{31.99}$$

Photon mapping has, in addition to the scene description, two main parameters: N, the total number of photons to be "shot" from the light sources, and K, the number of photons to be used in estimating the outgoing radiance at any point. The value N is used only in the construction of the photon map; the value K only in the radiance estimation. A third parameter, *maxBounce*, limits the number of bounces that a photon can take during the photon-tracing phase of the algorithm.

The photon map itself is a k-d tree storing photons, using the position of the photon as the key. Listing 31.7 shows how the photon map is constructed.

Listing 31.7: Constructing a photon map via photon tracing.

```
1  Input: N, the number of photons to emit,
       maxBounce, how many times a photon may be reflected
       a scene consisting of surfaces and lights.
   Output: a k-d tree containing many photons.
2
3  define buildPhotonMap(scene, N, maxBounce):
4    map = new empty photon map
5    repeat N times:
6      ph = emitPhoton(scene, N)
7      insertPhoton(ph, scene, map, maxBounce)
8    return map
9
10 define emitPhoton(scene, N):
11   from all luminaires in the scene, pick L with probability
12     p = ∑_λ Φ_L(λ) / ∑_λ Φ(λ).
13
14   ph = a photon whose initial position P is chosen uniformly
15        randomly from the surface of L, whose direction ω_i
16        is chosen proportional to the cosine-weighted radiance at P in
17        direction ω_i, and with power Φ_i = Φ_L/(Np).
18
19 define insertPhoton(ph = (P, ω_i, Φ_i), scene, map, maxBounce)
20   repeat at most maxBounce times:
21     ray trace from ph.P in direction ph.ω_i to find point Q.
22     ph.P = Q
23     store ph in map
24     foreach wavelength band λ:
25       p_λ = ∫ f_s(Q, λ, ω_i, ω_o) ω_o · n dω_o, probability of scattering.
26     p̄ = average of p_λ over all wavelength bands λ.
27     if uniform(0, 1) > p̄
28        // photon is absorbed
29        exit loop
30     else
31        ω_o = sample of outgoing direction in proportion to f_s(Q, ω_i, ·)
32        ph.ω_i = −ω_o
33        foreach wavelength band λ:
34           ph.Φ_i(λ) *= p_λ/p̄
```

Most of this is a straightforward simulation of the process of light bouncing around in a scene. If, for a moment, we ignore wavelength dependence, then the absorption step can be explained, just as we saw in path tracing, as follows: When a photon hits a surface that scatters 30% of the arriving light, we could produce a scattered photon with its power multiplied by a factor of 0.3, or we could produce a scattered photon will full power, but only 30% of the time, an approach called **Russian roulette.** Over the long term, as many photons arrive at this point and get scattered, the total outgoing power is the same, but there's an important difference between the two strategies: In the second, at least for a scene that is not dependent on wavelength, the power of a photon never changes. This means that all samples stored in the photon map have the same power. This makes the radiance-estimation step work better in general, although the statistical reasons for that are beyond the scope of this book. The code, in saying "reflect a full-strength photon with a probability determined by the scattering probability," is applying Russian roulette.

Because the scattering is wavelength-dependent, the final update to $\Phi_\mathrm{i}(\lambda)$ scales the power in each band in proportion to that band's scattering probability. Notice that if the surface is white (i.e., reflectance is the same across all bands), then $\Phi(\lambda)$ is unchanged. By contrast, if we're using RGB and the surface is pure red, then the average scattering probability is $1/3$; the red component of the photon power is multiplied by $\frac{1}{1/3} = 3$, while the green and blue components are set to zero.

Inline Exercise 31.12: What happens to the power of a photon if the surface is a uniform 30% gray, so it reflects 30% of the light at each wavelength?

The actual implementation of light emission and of scattering (particularly for reflectance models that have a diffuse, a glossy, and a specular part, for instance) requires some care; we discuss these further in Chapter 32.

The second part of photon mapping is radiance estimation at points visible from the eye, determined, for instance, by tracing rays from the eye E into the scene. Before performing any radiance estimation, however, we balance the k-d tree. Then for each visible point P, with normal $\mathbf{n}$, we let $\omega_\mathrm{o} = \mathcal{S}(E - P)$, and compute the radiance.

1. Set $L = 0\ \mathrm{W/m^2sr}$. L represents the radiance scattered toward the eye.
2. Find the K photons nearest to P in the photon map, by searching for a radius r within which there are K photons.
3. For each photon $ph = (Q, \omega_\mathrm{i}, \Phi_\mathrm{i})$, update L using

$$L \leftarrow L + f_s(P, \omega_\mathrm{i}, \omega_\mathrm{o})\Phi_\mathrm{i}\kappa(Q - P), \tag{31.100}$$

   where $\kappa(Q - P) = \frac{1}{\pi r^2}$ is called the estimator **kernel.** This assignment is wavelength-dependent (i.e., if we use three different wavelength bands, Equation 31.100 represents three assignments, one for each of R, G, and B).

It's easy to see that this computation is an approximation of the integral $\int f_s(P, \omega_\mathrm{i}, \omega_\mathrm{o})\omega_\mathrm{i} \cdot \mathbf{n}\, d\omega_\mathrm{i}$ over the positive hemisphere at P: The arriving radiance at nearby points is used as a proxy for the arriving radiance at P in the integral. Of course, light arriving at some point Q that's *near* P from direction ω_i may have originated at a fairly nearby light source. If so, then the arriving direction at Q will be different from that at P (see Figure 31.28).

Furthermore, it's possible that some light is visible from point P but shadowed at point Q, in which case the use of the incoming light at Q in estimating the incoming light at P is inappropriate. It is for this reason that photon mapping is *biased:* Without infinitely many photons, some points in dark areas will get their radiance estimates in part from photons in lighter areas, biasing them toward brighter estimates.

On the other hand, at least at points of diffuse surfaces, photon mapping is *consistent:* As the number of photons, N, goes to infinity, the K samples used to estimate the light arriving at P are closer and closer to P, causing the incoming directions to be increasingly better approximations of the incoming direction at P, and the radiances to be increasingly better approximations of the radiances at P.

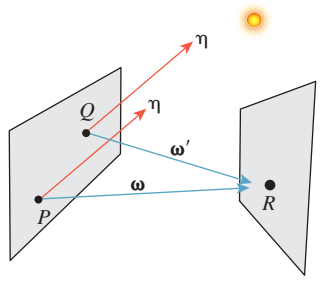


Figure 31.28: Light from the sun arrives at P from some direction $\boldsymbol{\eta}$*; sunlight will also arrive at Q from almost exactly that direction. But if light from a nearby point R arrives at P in direction* $\boldsymbol{\omega}$*, that same source will provide light at Q from direction* $\boldsymbol{\omega}'$ *which may not be close to* $\boldsymbol{\omega}$*.*

Inline Exercise 31.13: The preceding analysis of consistency assumed that K was held constant while N goes to infinity. Is the analysis still valid if K is set to a constant multiple of N, say, $K = 10^{-5}N$? Why or why not?

Estimating arriving radiance from nearby samples works best when the arriving radiance varies smoothly as a function of both position and angle. When there are point lights and sharp edges in the scene, we get hard shadows, which makes the arriving radiance discontinuous. On the other hand, this nonsmoothness in arriving radiance is primarily a consequence of *direct lighting,* that is, light paths of the form *LDE*. We can therefore divide the domain of the integral in the rendering equation into two parts: those paths of the form *LDE*, and all other paths. We can estimate the integral as a sum of the integrals over each part. The first part is relatively easy: Single-bounce ray tracing suffices to estimate the direct lighting at every point of the scene. What about the second part? We can estimate that using photon mapping! But to do so, we need to eliminate any estimate of transport of the form *LDE* from the photon map. We do so by slightly modifying the construction of the photon map: For each of the N photons, we record in the photon map only the second and subsequent bounces.

Photon mapping has other limitations. Because points that are nearby in space may not be nearby in surface normal (see Figure 31.29), using all nearby photons can give erroneous estimates. Several heuristics have been applied to mitigate this problem [Jen01, ML09].

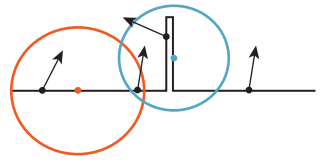


Figure 31.29: Estimating arriving radiance with nearby photons works badly at corners and near thin walls.

When we use the constant kernel $\kappa(\mathbf{v}) = 1/(\pi r^2)$, we have the problem that as the point at which we're estimating radiance moves, we typically lose one photon at one side of the moving disk and gain another somewhere on the other side; if these two photons have different **power vectors** (power in each wavelength band), then the radiance estimate can have discontinuities, which appear as noticeable artifacts in the final results. Alternative kernels can mitigate this somewhat, as Jensen describes.

What we've described is the basic form of photon mapping presented in Jensen's book, but there are many implicit parameters in the description. For instance, the k-d tree used for storing photons can be replaced by other data structures like spatial hashing [MM02], the kernel used for density estimation can be varied, and even the density estimation technique itself can be altered.

31.19.0.1 Final Gathering

One particularly effective enhancement is use of a "final gathering" step during density estimation. When we examine a point P in the scene, we can estimate the

field radiance at P either by looking at its nearest neighbors, as we've described, or by shooting lots of rays from P to hit *other* points, $Q_i (i = 1, 2, \ldots)$, and then using nearest-neighbor techniques to estimate the field radiance at each Q_i, and the light reflected back toward P from each Q_i. The collection of these gathered lights is also a valid estimate of the field radiance at P, but is much less likely to exhibit the discontinuities described above, as any discontinuity is typically averaged with a great many other continuous functions.

31.19.1 Image-Space Photon Mapping

McGuire and Luebke [ML09] have rethought photon mapping for a special case—point lights and pinhole cameras—by recognizing that in this case, some of the most expensive operations could be substantially optimized. One of these operations—the transfer of information from photons in the photon map to pixels in the image—is highly memory-incoherent in the original photon-mapping algorithm: One must seek through the k-d tree to find nearby photons, and depending on the memory layout of that tree, this may involve parts of memory distant from other parts. On the other hand, if every photon, once computed, could make its contribution to all the relevant pixels (which are naturally close together in memory), there would be a large improvement. The resultant algorithm is called **image-space photon mapping.** This approach harkens back to Appel's notion of drawing tiny "+" signs on a plotter: These marks were spatially localized, and hence easy to draw with a plotter. It's also closely related to progressive photon mapping [HOJ08], another approach that works primarily in image space.

The key insight is that when we ray-cast into the scene to gather light from photons, adjacent pixels are likely to gather light from the same photons; we could instead project the photons onto the film plane and add light to all the pixels within a small neighborhood. There are quite a few subtleties (How large a neighborhood? What about occlusion?), but the algorithm, implemented as a CPU/GPU hybrid, is much faster than ordinary photon mapping. While the algorithm only works with point lights and pinhole cameras, the added speed may be sufficient to justify this limitation in some applications such as video games.

31.20 Discussion and Further Reading

Many of the ideas in this chapter have been implemented in the open-source Mitsuba renderer [Jak12]. Seeing such an implementation may help you make these ideas concrete (indeed, we strongly recommend that you look at that renderer), but we also recommend that you first follow the development of the next chapter, in which some of the practical little secrets of rendering, which clutter up many renderers, are revealed. This will make looking at Mitsuba far easier.

While much of this chapter has been about simulation of light transport, there are a few large-scale observations about light in scenes that have crept into the discussion in disguise. We now revisit these in greater detail.

For instance, in classifying light paths using the Heckbert notation, we effectively partition the space of paths into subspaces, each of which we consider differently. We know, for instance, that much of the light in a scene is direct light, carried along *LDE* paths, and that in a scene with point lights and hard-edged

objects, this direct light contains many of the discontinuities in the light field (corresponding to silhouettes, contours, and hard shadows).

The partitioning by Heckbert classes is useful, but rather coarse: Paths with multiple specular bounces may have high "throughput," but this only matters if they start at light sources. There may someday be other ways of classifying paths that allows us to delimit the "bright parts" of path space more efficiently.

As we consider computing the reflectance integral at some point, it's reasonable to ask, "How much do the variations in the field radiance matter?" If the surface is Lambertian, the answer is, "Generally not too much." If it's shiny, then variations in field radiance matter a lot. But having computed the reflectance integral to produce surface radiance, which may have considerable variation with respect to outgoing direction, we can ask, "When this arrives at another surface, how will that variation appear?" If we look at such a surface up close, moving our eyes a few centimeters may yield substantial variation in the appearance of the surface. But if we look at the same surface from a kilometer away, we'll have to move our eyes dozens of meters to see the same variation. This dispersal of high-frequency content in the light field (and other related phenomena) is discussed by Durand et al. in a thought-provoking paper [DHS+05] that ties together the frequency content of the radiance field, both in spatial and angular components, with ideas about appropriate rates for sampling in various rendering algorithms.

We've treated rendering as a problem of simulating sensor response to a radiance field, with the implicit goal of getting the "right" sensor value at each pixel. This may not always be the right goal. If the image is for human consumption, it's worth considering the end-to-end nature of the process, from model all the way to percept. Humans are notorious, for instance, for their inability to detect absolute levels of almost any sensation, but they are generally quite sensitive to variation. We can't tell how bright something is, but we can reliably say that it's a little brighter than another thing that's near it, for instance. This means that if you had a choice between a perfect image, corrupted by noise so that a typical pixel's value was shifted by, say, 5%, and the same perfect image, with every pixel's value multiplied by 1.1, you'd probably prefer the second, even though the first is closer to the perfect image in an L^2 sense.

Indeed, the human eye, while sensitive to absolute brightness, is much *more* sensitive to contrast. It might make sense, in the future, to try to render not the image itself, but rather its gradients, perhaps along with precise image values at a few points. The "final step" in such a rendering scheme would be to integrate the gradients to get an intensity field, subject to the constraints presented by the known values; such a constrained optimization might better capture the human notion of correctness of the image. We are not proposing this as a research direction, but rather to get you thinking about the big picture, and what aspects of that big picture current methods fail to address.

We've concentrated on the operator-theoretic solution of the rendering equation, but we've by no means exhausted these approaches. The solution says that $(I-T)^{-1}\mathbf{e} = (I+T+T^2+\ldots)\mathbf{e}$, where $\mathbf{e}$ describes the luminaires in the scene. If we slightly rewrite the right-hand side, we can discover other approaches based on this solution:

$$(I-T)^{-1}\mathbf{e} = (I+T+T^2+\ldots)\mathbf{e}, \tag{31.101}$$

$$= \mathbf{e} + (T+T^2+\ldots)\mathbf{e}, \text{ and} \tag{31.102}$$

$$= \mathbf{e} + (I+T+\ldots)T\mathbf{e}. \tag{31.103}$$

This says that aside from light reaching the eye directly from luminaires (the first **e** term), we can instead apply the transport operator once to the luminaires ($T\mathbf{e}$) to get a new set of luminaires which we can then render using the series solution ($I+T+\ldots$). This is the key idea in an approach called virtual point lights [Kel97]: The initial transport of the luminaires is performed by something very similar to photon tracing, except that instead of recording the field radiance at the intersection point, we record the resultant surface radiance after scattering. This surface radiance becomes one of the virtual point lights (or, in image-space photon mapping, the bounce map).

If, following Arvo, we further decompose T into the product KG, where G transports surface radiance at each point to field radiance at another, and K scatters field radiance at a point into surface radiance there, then we can consider breaking a term off the series solution in a slightly different way:

$$(I-T)^{-1}\mathbf{e} = (I+T+T^2+\ldots)\mathbf{e}, \tag{31.104}$$

$$= \mathbf{e} + (T+T^2+\ldots)T\mathbf{e}, \tag{31.105}$$

$$= \mathbf{e} + (I+T+\ldots)KG\mathbf{e}, \text{ and} \tag{31.106}$$

$$= \mathbf{e} + ((I+T+\ldots)K)(G\mathbf{e}). \tag{31.107}$$

In this form, we transport the radiance from the luminaires to become field radiance at the other surfaces in the scene ($G\mathbf{e}$). Subsequent processing involves tracing rays from the eye to various depths, and then scattering (the K term) the field radiance we find at intersection points. This can be regarded as a primitive form of photon mapping, in which the photon map contains only one-bounce photons.

Doubtless other factorizations of the series solution can lead to further algorithms as well.

This chapter has merely given a broad view of some topics in rendering, focusing attention on Monte Carlo methods because of our belief that these are likely to remain dominant for some time. To paraphrase Michael Spivak [Spi79b], we've introduced you to much of the foundational material, and "beyond all this lies a vast porridge of literature, and [we are] not glutton[s] enough to pick out all the raisins."

If you want to know more about the physical and mathematical basis of rendering, and especially Monte Carlo methods, we recommend Veach's dissertation [Vea97] as an education in itself. For those for whom the assumptions that lie at the foundation of rendering are important, Arvo's dissertation [Arv95] is an excellent starting point, particularly for the operator-theoretic point of view. Both, however, involve considerable mathematics. The SIGGRAPH course notes [JAF+01] give a slightly less demanding transition.

On the other hand, if efficient approximations to the ideal interest you, then *Real Time Rendering* [AMHH08] is an excellent reference.

For modern implementations of Monte Carlo methods, *Phyiscally Based Rendering* by Pharr and Humphreys [PH10] is detailed and comprehensive.

Most important, the best place to start is with current research in the field. Research in rendering is featured at almost every graphics conference. The Eurographics Symposium on Rendering deserves special mention, however, as its long-term focus on rendering has made it a particularly fertile ground for new ideas. We suggest that you grab a paper, start reading and chasing references, and be both open-minded and skeptical at all times.

It's also interesting to ask yourself, "What remains to be done?" Do current images lack realism because the modeling of materials is inadequate? Because certain classes of light paths are not being sampled? Because we aren't using enough spectral bands? Because important *information* is contained in high-bounce-count paths, even though very little light energy is there? Probably all of these matter to some degree. At the same time, we make some assumptions (the "ray optics" assumption, for one) that limit the phenomena we can hope to capture faithfully. Do these matter? How important is diffraction? How important are wave optics phenomena in general? Questions like these will be the foundation of future research in rendering.

31.21 Exercises

Exercise 31.1: Suppose that tracing a ray in your scene takes time A on average, while evaluating the BRDF on a pair of vectors takes time B. (a) In tracing N rays from the eye using path tracing, using a fixed attenuation rate r (so that a path is extended at each point with probability $(1-r)$), estimate the time taken in terms of A and B (assume all other operations are free).
(b) Consider tracing $N/2$ rays from the eye and $N/2$ rays from the single light source in a scene using bidirectional path tracing; do the same computation.

Exercise 31.2: The radiosity equation

$$(\mathbf{I}-\mathbf{F})\mathbf{b}=\mathbf{e} \tag{31.108}$$

has the form $\mathbf{Mb}=\mathbf{e}$, where $\mathbf{M}=\mathbf{I}-\mathbf{F}$. In practice, the largest eigenvalue or singular value of $\mathbf{M}$ is considerably less than 1.0; this means that powers of $\mathbf{M}$ tend to grow rapidly smaller. That can be used to solve the equation relatively quickly.
(a) Show that $(\mathbf{I}-\mathbf{F})^{-1}=\mathbf{I}+\mathbf{F}+\mathbf{F}^2+\ldots$ by multiplying the right-hand side by $\mathbf{I}-\mathbf{F}$ and canceling. This sort of cancellation is valid only if the right-hand side is an absolutely convergent series; fortunately, if the eigenvalues or singular values are small, it is, justifying this step.
(b) Show that

$$\mathbf{b}=\mathbf{e}+\mathbf{Fe}+\mathbf{F}^2\mathbf{e}+\ldots. \tag{31.109}$$

(c) Letting $\mathbf{b}_0=\mathbf{e}$ and $\mathbf{b}_1=\mathbf{e}+\mathbf{Fb}_0$, and generally letting $\mathbf{b}_k=\mathbf{e}+\mathbf{Fb}_{k-1}$, show that $\mathbf{b}_k$ is the sum of the first $k+1$ terms of the right-hand side of the equation for $\mathbf{b}$, and that thus $\mathbf{b}_k\rightarrow\mathbf{b}$ as $k\rightarrow\infty$. Thus, an algorithm for computing the radiosity vector $\mathbf{b}$ is to start with $\mathbf{b}=\mathbf{e}$, and then multiply by $\mathbf{F}$ and add $\mathbf{e}$ repeatedly until $\mathbf{b}$ has converged sufficiently.
(d) If we think of the ith entry of $\mathbf{b}$ as the radiosity at patch i, then multiplying by $\mathbf{F}$ distributes this radiosity among all other patches. Rather than computing $\mathbf{Fb}$ in its entirety, which can involve lots of multiplication, we can push the "unshot" radiosity from a single patch through the matrix. Various algorithms exploit this idea, seeking, for example, to push through the matrix the largest unshot radiosity. Implement one of these for a 2D radiosity model, and see how its runtime and

convergence compare with the naive algorithm that multiplies **b** by all of **F** at each step.

Exercise 31.3: Final gathering in photon mapping involves sampling from the hemisphere at a point P where we're trying to estimate field radiance. As we move from point to point, our varying choice of samples will introduce noise into the estimates. Argue both for and against a strategy in which we choose, once and for all, a fixed collection of directions to use in the final gathering step.

Chapter 32

Rendering in Practice

32.1 Introduction

In this chapter we show implementations of two renderers—a path tracer and a photon mapper—with some of the optimizations that make them worth using. Both approaches are currently in wide use, are fairly easy to understand, and form complete solutions to the rendering problem in the sense that they can be shown (under reasonable conditions) to provide consistent estimates of the values we seek (i.e., "properly" rendered images).

We're *not* recommending these as ideal renderers. Rather, we treat them as case studies. They are rich enough to exhibit of the complexities and features of a modern renderer; they provide the foundation necessary for you to read research papers on rendering.

We assume that you've implemented the basic ray tracer described in Chapter 15. Much of this chapter also depends heavily on Chapters 30 and 31.

In the course of implementing these renderers, we describe ways to structure the representation of geometry in a scene, of scattering, and of samples that contribute to a pixel. These are not always in a form immediately recognizable from the mathematical formulation of the previous chapters, as you'll know from Chapter 14.

In Section 32.8, we discuss the debugging of rendering programs, showing some example failures and their causes, and suggesting how you can learn to identify the *kind* of bug from the kind of visual artifacts you see.

32.2 Representations

As you build a ray-casting-based renderer, your choices of representations will have large-scale impacts. Is the scattering model you've chosen rich enough to represent the phenomena you wish to simulate? Is it easy to sample from a probability distribution proportional to $\omega \mapsto f_s(\omega_i, \omega)|\omega \cdot \mathbf{n}|$ for some fixed vector ω_i? Is your scattering energy-conservative? Does your scene representation make ray-scene

intersection fast and robust? Does your representation of luminaires make it easy to select points on a luminaire uniformly with respect to area?

Inline Exercise 32.1: For each of the questions above, describe how a basic ray tracer's output or running time might be affected by the answer being "Yes" or "No."

Beyond these choices, there are the practical matters of modeling. For instance, the scattering properties of a surface are usually defined or measured relative to the surface normal (and perhaps relative to a tangent basis as well), while we've treated the scattering model (or at least the bidirectional scattering distribution function or BSDF) as a function of a point in space and two direction vectors. In practice, of course, we trace a ray to find a point of some *surface,* and then find the BSDF at that point as a *surface* property, with its parameters being determined both by surface position and by various texture maps. We'll begin by discussing this particular simplification.

32.3 Surface Representations and Representing BSDFs Locally

Consider a patch of surface so small that it may locally be considered flat, and a local coordinate frame at a point P, with unit normal vector $\mathbf{n}$ and unit tangent vectors $\mathbf{u}$ and $\mathbf{v}$ such that $\mathbf{u}, \mathbf{v}, \mathbf{n}$ is an orthonormal basis of 3-space, as shown in Figure 32.1. This decomposition of a surface into a tangent space and a normal space depends on **local flatness;** it's problematic at edges and corners (like those of a cube), where it's not obvious which directions should be called "tangent" or "normal." This is a real problem for which graphics has yet to determine a definitive answer.

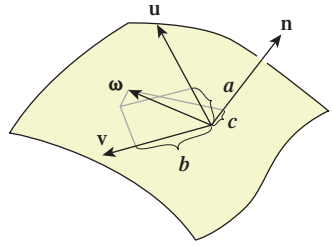


Figure 32.1: A local basis at a point P, consisting of mutually perpendicular unit vectors. The vector ω is shown being decomposed into a linear combination of these via dot products.

For any vector ω at P, we can easily represent $\omega = a\mathbf{u} + b\mathbf{v} + c\mathbf{n}$, by computing dot products: $a = \omega \cdot \mathbf{u}$, etc.

Inline Exercise 32.2: If $\omega, \mathbf{n}, \mathbf{u}$, and $\mathbf{v}$ are expressed in world coordinates, and $\mathbf{M}$ is a 3×3 matrix whose rows are $\mathbf{u}, \mathbf{v}$, and $\mathbf{n}$, then show that $\mathbf{M}\omega = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$. Knowing this lets us convert from world coordinates to local tangent-plus-normal coordinates.

Consider now the BSDF $f_s(P, \omega_i, \omega_o)$, which is a function of a surface point and two (unit) vectors at that point. If we write $\omega_i = a\mathbf{u} + b\mathbf{v} + c\mathbf{n}$ and $\omega_o = a'\mathbf{u} + b'\mathbf{v} + c'\mathbf{n}$, we can define a new function,

$$\bar{f_s}(P, a, b, c, a', b', c') = f_s(P, \omega_i, \omega_o). \tag{32.1}$$

We can go further, however. Since ω_i and ω_o are unit vectors, we can express them in polar coordinates using ϕ for longitude and θ for latitude, corresponding to the standard use of θ for the angle between ω and $\mathbf{n}$. This gives $\theta = \cos^{-1}(b)$ and $\phi = \text{atan2}(c, a)$, and we can write

$$\hat{f}_r(P, \theta_\mathrm{i}, \theta_\mathrm{o}, \phi_\mathrm{i}, \phi_\mathrm{o}) = f_\mathrm{s}(P, \omega_\mathrm{i}, \omega_\mathrm{o}). \tag{32.2}$$

The function $\hat{f}_r$ is what a gonioreflectometer actually measures. Notice that f_s and $\hat{f}_r$ are merely different representations of the same thing, like the rectangular- or polar-coordinate representations of a curve. (We discussed such shifts of representation in Chapter 14)

The function $\hat{f}_\mathrm{s}$ has a form in which certain common properties of BSDFs can be easily expressed. For instance, the Lambertian bidirectional reflectance distribution function (BRDF) is completely independent of θ_o, ϕ_i, and ϕ_o. Because of this, the particular choice of **u** and **v** is irrelevant for the Lambertian BRDF: The dot products of ω_i and ω_o with **u** and **v** are only used in computing ϕ_i and ϕ_o.

The Phong and Blinn-Phong BRDFs both depend on θ_i and θ_o, but their dependence on ϕ_i and ϕ_o is rather special: They depend only on the *difference* of $\phi_\mathrm{i} - \phi_\mathrm{o}$ (indeed, on this difference taken mod 2π).

Inline Exercise 32.3: (a) Explain the claim that the Blinn-Phong BRDF depends only on the difference of ϕ_i and ϕ_o.
(b) Show that in fact it depends only on the *magnitude* of the difference: The sign is irrelevant.

This dependence on the difference in angles again means that the BSDF expressed in (θ, ϕ) terms, $\hat{f}_\mathrm{s}$, is independent of the choice of **u** and **v**: If we rotated these in the tangent plane by some amount α, then both ϕ_i and ϕ_o would change by α (and possibly by an additional 2π), and their difference (mod 2π) would remain invariant. BSDFs with this property are said to be **isotropic,** and they can be represented by functions of the three variables $\theta_\mathrm{i}, \theta_\mathrm{o}$, and $\phi = (\phi_\mathrm{i} - \phi_\mathrm{o}) \bmod 2\pi$. The great majority of materials currently used in graphics are represented by BSDFs (indeed, BRDFs) that fall into this category; the exceptions (**anisotropic** materials) are things like brushed aluminum, in which the brushing direction introduces an anisotropy. Materials that are represented using subsurface scattering often have interior structure that makes them anisotropic as well, so the simplified representation is often inapplicable to those.

The preceding discussion has been in terms of the angles $\theta_\mathrm{i}, \phi_\mathrm{i}, \theta_\mathrm{o}$, and ϕ_o to emphasize that the BSDF is a function on a four-dimensional domain. In practice, however, it is the sines and cosines of these angles that most often enter into the computations, at least for analytically expressed BSDFs. (For tabulated BSDFs, we can tabulate based not on θ_i and θ_o, but on their cosines, so the same argument applies to those.) In practice, a BSDF implementation will typically take a point, P, and the two vectors ω_i and ω_o, and promptly express these vectors in terms of **u**, **v**, and **n**.

How does all this look in an implementation? Part of G3D's implementation of a generalized Blinn-Phong model is shown in Listing 32.1.

There are several design choices here. The first is that a `SurfaceElement` is used to represent the intersection of a ray with a surface in the scene. Among other things, it has data members `material` and `shading`. The `material` stores things like the Phong exponent, the reflectivities in the red, green, and blue spectral regions, etc. The `shading` stores the intersection point, the texture coordinates there, and the surface normal there. (It may help, when reading expressions like `p.shading.normal`, to treat "shading" as an adjective. Thus, `p.shading.normal` is the shading normal, while `p.geometric.normal` is the geometric normal.)

Listing 32.1: Part of an implementation of Blinn-Phong reflectance.

```
Color3 SurfaceElement::evaluateBSDFfinite(w_i, w_o) {
    n = shading.normal;
    cos_i = abs(w_i.dot(n));

    Color3 S(Color3::zero());
    Color3 F(Color3::zero());
    if ((material.glossyExponent != 0) && (material.glossyReflect.nonZero())) {
        // Glossy

        // Half-vector
        const Vector3& w_h = (w_i + w_o).direction();
        const float cos_h = max(0.0f, w_h.dot(n));

        // Schlick Fresnel approximation:
        F = computeF(material.glossyReflect, cos_i);
        if (material.glossyExponent == finf())
            S = Color3::zero()
        } else {
            S = F * (powf(cos_h, material.glossyExponent) * ...
        }
    }
    ...
```

The surface normal is used immediately to compute $\cos\theta_i$, an example of expressing one of the two input vectors in the local frame of reference. The half-vector (`direction()` returns a unit vector) is computed from ω_i and ω_o, which are called `w_i` and `w_o` in the code. The Schlick approximation of the Fresnel term is computed and used to determine the glossy reflection. The remainder of the elided code computes the diffuse reflection. Missing from this code are the evaluations of the mirror-reflection term and of transmittance based on Snell's law, each of which corresponds to an impulse in the scattering model. The splitting off of these impulse terms makes the computation of the reflected light much simpler. Recall that what we've been expressing as an integral, namely,

$$\int_{\omega_o \in \mathbf{S}^2_+(P)} L(P, -\omega_i) f_s(\omega_i, \omega_o)\, \omega_i \cdot \mathbf{n}\, d\omega_o, \tag{32.3}$$

is really shorthand for a linear operator being applied to L, one that is defined in part by a convolution integrand like the one above, and in part by impulse terms like mirror reflectance, where for a particular value of ω_i, the integrand is nonzero only for a specific direction ω_o; the value of the "integral" is some constant (the impulse coefficient) times $L(P, -\omega_i)$.

Trying to approximate terms like mirror reflectance by Monte Carlo integration is hopeless: We'll never pick the ideal outgoing direction at random. Fortunately, these terms are easy to evaluate directly, so no approximation is needed. The `SurfaceElement` class therefore provides a method (see Listing 32.2) that returns all the impulses needed to evaluate the reflected radiance (in this case, the mirror-reflection impulse and the transmission impulse, although if we were rendering a birefringent material, there would be two transmissive impulses, so returning an array of impulses is natural).

G3D is designed around triangle meshes. The `SurfaceElement` class therefore contains some mesh-related items as well (see Listing 32.3).

Listing 32.2: A method that returns the impulse parts of a scattering model.

```
void getBSDFImpulses (Vector3& w_i, Array<Impulse>& impulseArray) {
    const Vector3& n = shading.normal;

    Color3 F(0,0,0);

    if (material.glossyReflect.nonZero()) {
        // Cosine of the angle of incidence, for computing
        //Fresnel term
        const float cos_i = max(0.001f, w_i.dot(n));
        F = computeF(material.glossyReflect, cos_i);

        if (material.glossyExponent == inf()) {
            // Mirror
            Impulse& imp     = impulseArray.next();
            imp.w            = w_i.reflectAbout(n);
            imp.magnitude    = F;
            ...
```

Listing 32.3: Further members of the `SurfaceElement` *class.*

```
class SurfaceElement {
public:
    ...
    struct Interpolated {
        /** The interpolated vertex normal. */
        Vector3 normal;
        Vector3 tangent;
        Vector3 tangent2;
        Point2  texCoord;
    } interpolated;

    /** Information about the true surface geometry. */
    struct Geometric {
        /** For a triangle, this is the face normal. This is useful
         for ray bumping */
        Vector3 normal;

        /** Actual location on the surface (it may be changed by
            displacement or bump mapping later. */
        Vector3 location;
    } geometric;
    ...
```

The vectors `tangent` and `tangent2` correspond to $\mathbf{u}$ and $\mathbf{v}$ above; when the surface is modeled, these must be specified at every vertex. Using the derivative of texture coordinates is one way to generate these; if we have texture coordinates (u_i, v_i) at vertex i, we can find a linear approximation for u across the interior of the triangle, and from this determine a direction $\mathbf{u}$ in which u grows fastest. We can then define $\mathbf{v}$ as $\mathbf{n} \times \mathbf{u}$ to get an orthonormal basis. Note, however, that this is a per-triangle computation, and the computation of $\mathbf{u}$ is not guaranteed to be consistent across triangles. Indeed, because of the mapmaker's dilemma (you can't flatten the globe onto a single piece of paper preserving angles and distances), the consistent assignment of texture coordinates across a whole surface is generally impossible. It's important that any anisotropic BRDF be used only in areas where $\mathbf{u}$ and $\mathbf{v}$ are defined consistently.

The vector `geometric.normal` is the triangle-face normal rather than the surface normal that it approximates. Depending on how the surface was originally modeled, these may be identical, or the surface normal may be some weighted combination of face normals, or it may be determined by some other method entirely. The triangle-face normal is useful in ray-tracing algorithms because if P is a point that's supposed to lie on a triangle T (see Figure 32.2), it may be that a ray traced from P into the scene first hits the scene at a point of T, because a roundoff or representation error places P slightly to one side of T. By slightly displacing (**bumping**) P along the normal to T, that is, by replacing P with $P+\epsilon\mathbf{n}$ for some small ϵ, we can avoid such false intersections. (Perhaps "nudging" would be a better term, to avoid conflict with the notion of bump-mapping, but "bumping" is the term used by G3D.) How large should ϵ be? A good rule of thumb is "no more than 1% of the size of the smallest object you expect to see in the scene." This implicitly establishes a condition on your models: No significant object or feature should be less than 100 times the largest gap between two adjacent floating-point numbers of the size you will be using. For instance, if everything in your scene will have coordinates between -100 and 100, and you will use IEEE 32-bit floating-point numbers, then since the largest gap between two floating-point numbers near 100 is about 4×10^{-6}, you should not expect to model any feature smaller than 4×10^{-4} units.

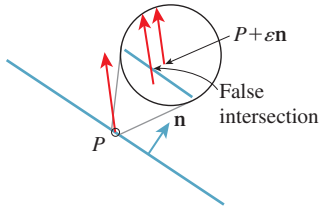


Figure 32.2: Bumping P out to $P' = P + \epsilon\mathbf{n}$ prevents rays starting at P from intersecting T.

Inline Exercise 32.4: If we're tracing a ray $P + t\mathbf{d}$, we could "bump" P *along* the ray, that is, bump it slightly in the direction $\mathbf{d}$. Argue that this is a bad idea by considering rays that are almost tangent to the surface.

Ray bumping is a design choice. It compensates for certain problems with fixed-accuracy representations of geometry. But as a design choice, it also has important representational consequences. For instance, any property of scene illumination that depends on features smaller than the bump size will potentially be misrepresented in our computations. This limits the things that our computational model can possibly produce with any accuracy. As an example, imagine looking at the sun through the gap between two nearly tangent, shiny cylinders (see Figure 32.3). A typical ray of sunlight will bounce many times as it passes between the cylinders. If the intercylinder distance is smaller than the bump size we use in our path tracer, the light can easily be absorbed rather than reflected. Is this a problem? If the bump size is small compared to a wavelength of light, it's not. Why not? Because for a gap of that scale, diffractive effects dominate, so our ray optics model was already inaccurate. Once again this is an instance of the Wise Modeling principle that when we simulate something, we need to be aware of the limits of our physical, mathematical, and computational models.

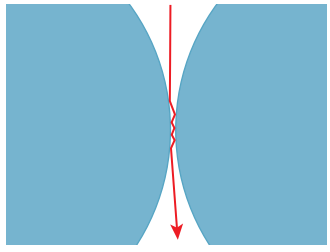


Figure 32.3: A ray from the sun shines through the gap between two nearly tangent, shiny cylinders.

32.3.1 Mirrors and Point Lights

If we allow point lights in the scene, then when we trace rays "from the eye," we'll hit the point light with probability zero, so that it'll almost never happen in practice. Similarly, if we allow perfect mirrors, we cannot write the scattering

operator as an integral to be approximated by naive sampling—we'll essentially never sample the direction of the mirror-reflected ray. When we combine the two, things are even worse.

Consider a scene consisting of a smooth mirrored ball illuminated by a point light. If we ray-trace from the eye through the pixel centers, we'll almost certainly miss the point light; if we ray-trace from the light, we'll miss the pixel centers. But if we suppose that the point light is in the scene as a proxy for a spherical light of some small radius r, then we know that we should see a highlight on the mirrored ball.

Losing that highlight is perceptually significant, even though the highlight might appear at only a single pixel of the image. We have three choices: We can abandon the convenient fiction of a point light, we can adjust the BRDF to compensate for the abstraction, or we can choose some other method for estimating the radiance arriving at the eye from that location. In an ideal world, with infinite rendering resources, we'd choose to use tiny point lights and cast a great many rays. Within the context of ray tracing, we can clamp the maximum shininess (i.e., the specular exponent) when we are combining a BRDF with a direct luminaire in the reflection operator. This ensures that with sufficiently fine sampling, the point light will produce a highlight. Of course, it also slightly blurs the reflection of every other object in the scene. The difference in appearance between a specular exponent of 10,000 and ∞ tends to be unnoticeable in general, so this is an acceptable compromise. On the other hand, if the specular exponent is 10,000, a very fine sampling around the highlight direction is required, or else we'll get high variance in our image. This leads us to the third alternative. It may make sense to separate out the impulse reflection of point lights (or even small lights) into a separate computation to avoid these sampling demands, but we will not pursue this approach here.

32.4 Representation of Light

In our theoretical discussion, we treated light as being defined by the radiance field $(P, \omega) \mapsto L(P, \omega)$: At any point P, in any direction ω, $L(P, \omega)$ represented the radiance along the ray through P in direction ω, measured with respect to a surface at P perpendicular to ω. When P is in empty space, this is a good abstraction. When P is a point exactly on a surface, there are two problems.

1. The precise relationship between geometric modeling and physics has been left undefined. We haven't said whether a solid is open (i.e., does not contain its boundary points, like an open interval) or closed; equivalently, we haven't said whether a ray leaving from a surface point of a closed surface intersects that surface or not.
2. When P is a point on the surface of a *transparent* solid, like a glass sphere, and ω points into the solid (see Figure 32.4), there are two possible meanings for $L(P, \omega)$: the light arriving at P from distant sources, or the light traveling from P into the interior of the surface. Because of Snell's law, material opacity, and internal reflection, these two are almost never the same.

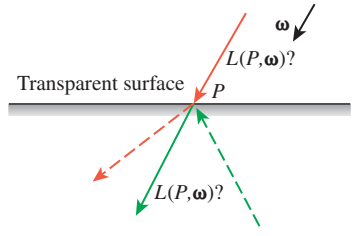


Figure 32.4: Is $L(P, \omega)$ the radiance along the solid red or the solid green arrow?

We addressed the second problem in Chapter 26, by defining an incoming and outgoing radiance for each pair (P, ω), where P was a surface point and ω was a unit

vector, by comparing ω with the normal $\mathbf{n}(P)$ to the surfaces; we also mentioned that Arvo's division of the radiance field into field radiance and surface radiance accomplishes the same thing.

For the first problem, we will say that a point on the boundary of a solid is actually part of that solid, so a point P on the surface of a glass ball is actually part of the ball (and for a varnished piece of wood, a point on the boundary is treated as being in both the varnish *and* the wood). This means that a ray leaving P in the direction $\mathbf{n}_P$ first intersects the ball at P. (As a practical matter, avoiding the intersection at $t = 0$ requires a comparison of a real number against zero, which is prone to floating-point errors, so including the first hit point is easier than avoiding it.) Thus, with this model of surface points, the notion of "bumping" is not merely a convenience for avoiding roundoff error problems, it's a necessity.

32.4.1 Representation of Luminaires

32.4.1.1 Area Lights

Our simple scene model supports a very basic kind of area light: We represent area light sources with a polygon mesh (often a single polygon) and an emitted power Φ. At each point P of a polygon, light is emitted in every direction ω with $\omega \cdot \mathbf{n}_P > 0$; the radiance along all such rays, in all directions, is constant over the entire luminaire.

The radiance along each ray can be computed by dividing the luminaire's power among the individual polygons by area; we thus reduce the problem to computing the radiance due to a single polygon of area A and power Φ. That radiance is $\frac{\Phi}{\pi A}$, as we saw in Section 26.7.3.

We will need to sample points uniformly at random (with respect to area) on a single area light. To do so, we compute the areas of all triangles, and form the cumulative sums $A_1, A_1 + A_2, \ldots, A_1 + \ldots + A_k = A$, where k is the number of triangles. To sample at random, we pick a uniform random value u between 0 and A; we find the triangle i such that $A_1 + \ldots + A_i \leq u$, and then generate a point uniformly at random on that triangle (see Exercise 32.6).

We'll also want to ask, for a given point P of the surface and direction ω with $\omega \cdot \mathbf{n}_P > 0$, what is the radiance $L(P, \omega)$? For our uniformly radiating luminaires, this is a constant function, but for more general sources, it may vary with position or direction.

32.4.1.2 Point Lights

Point-light sources are specified by a location[1] P and a power Φ. Light radiates uniformly from a point source in all directions. As we saw in Chapter 31, it doesn't make sense to talk about the radiance from a point source, but it *does* make sense to compute the reflected radiance from a point source that's reflected from a point Q of a diffuse surface. The result is

$$L(Q, \omega_o) = f_s(Q, \omega_i, \omega_o)\ \max(\omega_i \cdot \mathbf{n}_Q, 0) \frac{\Phi}{4\pi \|Q - P\|^2}. \tag{32.4}$$

1. We only allow finite locations; extending the renderer to correctly handle directional lights is left as a difficult exercise.

If a point light hits a mirror surface or transmits through a translucent surface, we can then compute the result of its scattering from the *next* diffuse surface, etc. This eventually becomes a serious bookkeeping problem, and since point lights are merely a convenient fiction, we ignore it: We compute only diffuse scattering of point lights. Although addressing this properly in a ray-tracing-based renderer is difficult, we'll see later that in the case of photon mapping, it's quite simple.

One useful compromise for point-light sources is to say that for the purpose of emission directly toward the eye, the point source is actually a glowing sphere of some small radius, r, while when it's used in the calculation of direct illumination, it's treated as a point. This compromise, however, has the drawback that it requires the design of the class for representing lights to know something about the kinds of rays that will be interacting with it (i.e., an eye ray will be intersection-tested against a small sphere, while a secondary ray will never meet the light source at all), which violates encapsulation.

32.5 A Basic Path Tracer

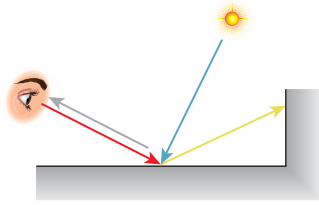


Figure 32.5: Ray tracing.

Recall the basic idea of ray tracing and path tracing: For each pixel of the image, we shoot several rays from the eye through the pixel area. A typical ray (the red one in Figure 32.5) hits the scene somewhere, and we compute the direct light arriving there (the nearly vertical blue ray), and how it scatters back along the ray toward the eye (gray). We then trace one or more recursive rays (such as the yellow ray that hits the wall), and compute the radiance flowing back along *them,* and how it scatters back toward the eye, etc. Having computed the radiance back toward the eye along each of the rays through our pixel, we take some sort of weighted average and call that the pixel value.

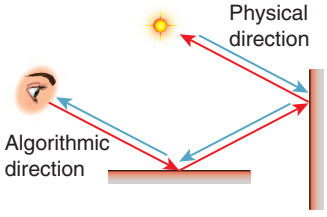


Figure 32.6: The algorithm works from the eye toward the light source (red); photons travel in the opposite direction (blue).

Because of the usual description of ray tracing ("Start from the eye, and follow a ray out into the scene until it hits a surface at some point P, and . . . "), we'll use the convention that the rays we discuss are always the result of tracing *from the eye,* that is, the first ray points away from the eye, the second ray points away from the first intersection toward a light or another intersection, etc. (see Figure 32.6).

On the other hand, the radiance we want to compute is the radiance that flows along the ray in the other direction. If the eye ray r starts at the eye, E, and goes in direction ω, meeting the scene at a point P, then we want to compute $L(P, -\omega)$, that is, we want to compute the radiance in the *opposite* of r's direction. We'll have various procedures like `Radiance3 estimateTotalRadiance(Ray r, ...)`; such functions always return the radiance flowing in the direction *opposite* that of r.

32.5.1 Preliminaries

We begin with a very simple path tracer, in which the image plane is divided into rectangular areas, each of which corresponds to a pixel. If a ray toward the eye passes through the (i,j)th rectangle, we treat the radiance as a sample of the radiance arriving at that rectangle. Despite the simplicity of the path tracer, we'll use a lot of symbols, which we list in Table 32.1; we'll define each one as we encounter it.

Table 32.1: Symbols used in the path tracer.

Symbol	Meaning
E	The eyepoint.
P	A surface point in the scene, often the first one encountered by a ray from the eye, but sometimes used generically.
Q, Q_j	A point on the surface of a luminaire or some other source of light arriving at P, such as an illuminated reflective surface.
$\mathbf{n}_P, \mathbf{n}_Q$	The unit normal vector at P, which we've denoted $\mathbf{n}(P)$ previously, or the same thing for Q.
ω_i	A unit vector pointing from P toward some source of light.
ω_o	A unit vector pointing from P in the direction in which reflected light from ω_i exits, typically toward E.
ω	A generic name for a unit vector, typically based at P.
$L(P, \omega)$	The radiance at a surface point P in direction ω. Note that in this chapter we only define L for surface points.
$L^\mathrm{e}(P, \omega)$	The light emitted at point P in direction ω; zero except when P is a point of a luminaire.
$L_j^\mathrm{e}(P, \omega)$	The light emitted by the jth luminaire.
$L^\mathrm{r}(P, \omega)$	The light reflected or transmitted (refracted) at P in direction ω. $L = L^\mathrm{e} + L^\mathrm{r}$.
$L^\mathrm{ref}(P, \omega)$	The light reflected at P in direction ω.
$L^\mathrm{trans}(P, \omega)$	The light transmitted at P in direction ω. $L^\mathrm{r} = L^\mathrm{ref} + L^\mathrm{trans}$.
f_s	The bidirectional scattering distribution function.
f_s^∞	The "impulse" part of f_s, corresponding to transmission or mirror reflection.
f_s^0	The finite part of f_s, corresponding to nonmirror reflection.

Let's suppose that there are k luminaires in the scene, each producing an emitted radiance field $(Q, \omega) \mapsto L_j^\mathrm{e}(Q, \omega), (j = 1, \ldots, k)$ which for any point-vector pair (Q, ω) with Q on a surface and $\omega \cdot \mathbf{n}_Q > 0$ is zero, except for points on the jth luminaire, and directions ω in which the light emits radiance. Most often this radiance field will be Lambertian, that is, $L_j^\mathrm{e}(Q, \omega)$ will be a constant for Q on the luminaire and any ω with $\omega \cdot \mathbf{n}_Q > 0$; it's zero otherwise. But for now, we'll just assume that it's a general light field.

Furthermore, let's assume that all surfaces are opaque—the only scattering that takes place is reflection. The change to include transmission will be relatively minor.

The rendering equation tells us that if P is the first point at which the ray $t \mapsto E - t\omega$ hits the geometry in the scene (see Figure 32.7), then

$$L(E, \omega) = L(P, \omega) \tag{32.5}$$

$$= \underbrace{L^\mathrm{e}(P, \omega)}_{\text{emitted}} + \underbrace{\int_{\omega_\mathrm{i} \in \mathbf{S}^2_+(\mathbf{n}_P)} f_\mathrm{s}(P, \omega_\mathrm{i}, \omega) L(P, -\omega_\mathrm{i}) \omega_\mathrm{i} \cdot \mathbf{n}_P \, d\omega_\mathrm{i}}_{\text{scattered}}. \tag{32.6}$$

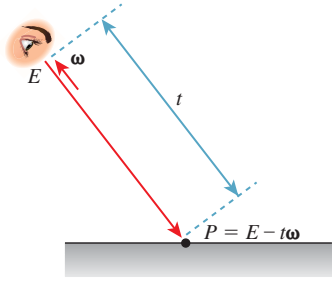


Figure 32.7: The eye E looks into the scene and sees P at distance t.

We can rewrite the second term as a sum by splitting the L in the integrand into two parts. As in Chapter 31, we let $L^{\mathrm{r}} = L - L^{\mathrm{e}}$ denote the *reflected* light (later, it will be reflected and refracted light) in the scene. At most surface points, $L^{\mathrm{r}} = L$, because most points are not emitters. At emitters, however, L^{e} is nonzero, so L^{r} and L differ. Thus,

$$\text{scattered} = \underbrace{\int_{\omega_{\mathrm{i}} \in \mathbf{S}^2_+(\mathbf{n}_P)} f_{\mathrm{s}}(P, \omega_{\mathrm{i}}, \omega) L^{\mathrm{e}}(P, -\omega_{\mathrm{i}}) \omega_{\mathrm{i}} \cdot \mathbf{n}_P \, d\omega_{\mathrm{i}}}_{\text{scattered direct}} \tag{32.7}$$

$$+ \underbrace{\int_{\omega_{\mathrm{i}} \in \mathbf{S}^2_+(\mathbf{n}_P)} f_{\mathrm{s}}(P, \omega_{\mathrm{i}}, \omega) L^{\mathrm{r}}(P, -\omega_{\mathrm{i}}) \omega_{\mathrm{i}} \cdot \mathbf{n}_P \, d\omega_{\mathrm{i}}}_{\text{scattered indirect}}. \tag{32.8}$$

Inline Exercise 32.5: Explain why, for a point Q on some luminaire and some direction ω, $L^{\mathrm{r}}(Q, \omega)$ might be nonzero.

The first integral, representing scattered direct light, can be further expanded. We write $L^{\mathrm{e}} = \sum_{j=1}^{k} L^{\mathrm{e}}_j$ as a sum of the illuminations due to the k individual luminaires, so that

$$\text{scattered direct} = \sum_{j=1}^{k} D_j(P, \omega), \text{where} \tag{32.9}$$

$$D_j(P, \omega) = \int_{\omega_{\mathrm{i}} \in \mathbf{S}^2_+(\mathbf{n}_P)} f_{\mathrm{s}}(P, \omega_{\mathrm{i}}, \omega) L^{\mathrm{e}}_j(P, -\omega_{\mathrm{i}}) \omega_{\mathrm{i}} \cdot \mathbf{n}_P \, d\omega_{\mathrm{i}}. \tag{32.10}$$

Thus, $D_j(P, \omega)$ represents the light reflected from P in direction ω due to direct light from source j.

Rather than computing D_j by integrating over all directions ω_{i} in $\mathbf{S}^2_+(\mathbf{n}_P)$, we can simplify by integrating over only those directions where there's a possibility that $L^{\mathrm{e}}(P, -\omega_{\mathrm{i}})$ will be nonzero, that is, directions pointing toward the jth luminaire. We do so by switching to an area integral over the region R_j constituting the jth luminaire; the change of variables introduces the Jacobian we saw in Section 26.6.5:

$$D_j = \int_{\omega_{\mathrm{i}} \in \mathbf{S}^2_+(\mathbf{n}_P)} f_{\mathrm{s}}(P, \omega_{\mathrm{i}}, \omega) L^{\mathrm{e}}_j(P, -\omega_{\mathrm{i}}) \omega_{\mathrm{i}} \cdot \mathbf{n}_P \, d\omega_{\mathrm{i}} \tag{32.11}$$

$$= \int_{Q \in R_j} f_{\mathrm{s}}(P, \omega_{\mathrm{i}}, \omega) E_j(Q, -\omega_{\mathrm{i}}) V(P, Q) \frac{(\omega_{\mathrm{i}} \cdot \mathbf{n}_P)(\omega_{\mathrm{i}} \cdot \mathbf{n}_Q)}{\|Q - P\|^2} \, dQ, \tag{32.12}$$

where $\omega_{\mathrm{i}} = \mathcal{S}(Q - P)$ is the unit vector from P toward Q, and we have introduced the visibility term $V(P, Q)$ in case the point Q is not visible from P. (Note that this transformation converts our version of the rendering equation into the form written by Kajiya [Kaj86].) The preceding argument only works for area luminaires. In the case of a point luminaire, this integral must be computed by a limit as in Chapter 31.

For an area luminaire, we estimate the integral with a single-sample Monte Carlo estimate:

$$D_j = \int_{Q \in R_j} f_s(P, \omega_i, \omega) E_j(Q, -\omega_i) V(P, Q) \frac{(\omega_i \cdot \mathbf{n}_P)(\omega_i \cdot \mathbf{n}_Q)}{\|Q - P\|^2} \, dQ \tag{32.13}$$

$$\approx Area(R_j) f_s(P, \omega_i, \omega) E_j(Q_j, E, -\omega_i) V(P, Q_j) \frac{(\omega_i \cdot \mathbf{n}_P)(\omega_i \cdot \mathbf{n}(Q_j))}{\|Q_j - P\|^2}, \tag{32.14}$$

where Q_j is a single point chosen uniformly with respect to area on the region R_j that constitutes the jth source.

The second integral, representing the scattering of indirect light, can also be split into two parts, by decomposing the function $\omega_i \mapsto f_s(P, \omega_i, \omega)$ in the integrand into a sum,

$$f_s(P, \omega_i, \omega) = f_s^\infty(P, \omega_i, \omega) + f_s^0(P, \omega_i, \omega), \tag{32.15}$$

where f_s^∞ represents the impulses like mirror reflection (and later, Snell's law transmission), and f_s^0 is the nonimpulse part of the scattering distribution (i.e., f_s is a real-valued function rather than a distribution). Each impulse can be represented by (1) a direction (the direction ω_i such that $-\omega_i$ either reflects or transmits to ω at P), and (2) an impulse magnitude $0 \leq k \leq 1$, by which the incoming radiance in direction $-\omega_i$ is multiplied to get the outgoing radiance in direction ω. We'll index these by the letter m (where $m = 1$ is reflection and $m = 2$ is transmission). Thus, we can write

$$\text{refl. indir. light} = \int_{\omega_i \in \mathbf{S}^2_+(\mathbf{n}_P)} f_s(P, \omega_i, \omega) L^r(P, -\omega_i) \omega_i \cdot \mathbf{n}_P \, d\omega_i, \tag{32.16}$$

$$= \left[\sum_m k_m L^r(P, -\omega_m) \right] + \tag{32.17}$$

$$\underbrace{\int_{\omega_i \in \mathbf{S}^2_+(\mathbf{n}_P)} f_s^0(P, \omega_i, \omega) L^r(P, -\omega_i) \omega_i \cdot \mathbf{n}_P \, d\omega_i}_{\text{diffusely reflected indirect light}}. \tag{32.18}$$

Finally, we can again estimate that last integral—the diffusely reflected indirect light—by a single-sample Monte Carlo estimate: We pick a direction ω_i according to some probability density on the hemisphere (or the whole sphere, when we're considering refraction as well as reflection), and estimate the integral with

$$\text{diff. refl. indir. light} = \frac{1}{\text{density}(\omega_i)} f_s^0(P, \omega_i, \omega) L^r(P, -\omega_i) \omega_i \cdot \mathbf{n}_P. \tag{32.19}$$

Note that while the BRDF doesn't literally make sense for an impulse-like mirror reflection, the computation we perform to compute mirror-reflected radiance has a form remarkably similar to that of Equation 32.19. We wrote it (Equation 32.17) in the form

$$k_1 L(P, -\omega_1), \tag{32.20}$$

where ω_1 was the reflection of ω_i (ω_2 was the *transmitted* direction). The coefficient k_1 plays the same role as the coefficient

$$\frac{1}{\text{density}(\omega_i)} f_s^0(P, \omega_i, \omega) |\omega_i \cdot \mathbf{n}_P| \tag{32.21}$$

of the radiance in the current case. In each case, we simply need our representation of the BRDF to be able to return the appropriate coefficient.

32.5.2 Path-Tracer Code

The central code in the path tracer is shown in Listing 32.4.

Listing 32.4: The core procedure in a path tracer.

```
Radiance3 App::pathTrace(const Ray& ray, bool isEyeRay) {
    // Compute the radiance BACK along the given ray.
    // In the event that the ray is an eye-ray, include light emitted
    // by the first surface encountered. For subsequent rays, such
    // light has already been counted in the computation of direct
    // lighting at prior hits.

    Radiance3 L_o(0.0f);

    SurfaceElement surfel;
    float dist = inf();
    if (m_world->intersect(ray, dist, surfel)) {
        // this point could be an emitter...
        if (isEyeRay && m_emit)
            L_o += surfel.material.emit;

        // Shade this point (direct illumination)
        if ( (!isEyeRay) || m_direct) {
            L_o += estimateDirectLightFromPointLights(surfel, ray);
            L_o += estimateDirectLightFromAreaLights(surfel, ray);
        }
        if (!(isEyeRay) || m_indirect) {
            L_o += estimateIndirectLight(surfel, ray, isEyeRay);
        }
    }

    return L_o;
}
```

The broad strokes of this procedure match the path-tracing algorithm fairly closely. Not shown is the outer loop that, for each pixel in the image, creates a ray from the eye through that pixel and then calls the `pathTrace` procedure (perhaps doing so multiple times per pixel and taking a [possibly weighted] average of the results).

The computation consists of five parts: finding where the ray meets the scene (and storing the intersection in a `SurfaceElement` called `surfel`), and then summing up emitted radiance, radiance due to direct lighting from area lights, radiance due to direct lighting from point lights, and a recursive term, all evaluated at the intersection point. The inclusion of each term is governed by a flag (`m_emit`, `m_direct`, `m_indirect`) that lets us experiment with the program easily when we're debugging. If we turn off direct and indirect light, it's really easy to tell whether the lamps themselves look correct, for instance.

Let's look at the four terms individually. The emissive term simply takes the emitted radiance at the surface point, called `surfel.geometric.position` in the code, but which we'll call P in this description, and adds it to the computed radiance. This assumes that the surface is a Lambertian emitter so that the outgoing radiance in every direction from P is the same. If instead of having a constant

outgoing radiance, the emitted radiance depended on direction, we might have written:

```
    if (includeEmissive) {
        L_o += surfel.material.emittedRadianceFunction(-ray.direction);
    }
```

where the emitted radiance function describes the emission pattern. Notice that we compute the emission in the opposite of the ray direction; the ray goes from the eye toward the surface, but we want to know the radiance from the surface toward the eye.

We add to this emitted radiance the reflection of direct light (i.e., light that goes from a luminaire directly to *P*, and that scatters back along our ray), and the reflection of *indirect* light (i.e., all light leaving the intersection point that's neither emitted light nor scattered direct light).

To compute the direct lighting from point lights (see Listing 32.5), we determine a unit vector `w_i` from the surface to the luminaire, and check visibility; if the luminaire is visible from the surface, we use `w_i` in computing the reflected light. This follows a convention we'll use consistently: The variable `w_i` corresponds to the mathematical entity ω_i; the letter "i" indicates "incoming"; the ray ω_i points from the surface toward the *source* of the light, and ω_o points in the direction along which it's scattered. This means that the variable `w_i` will be the first argument to `surfel.evaluateBSDF(...)`, and often a variable `w_o` will be the second argument. This convention matters: While the finite part of the BRDF is typically symmetric in its two arguments, both the mirror-reflectance and transmissive portions of scattering are often represented by nonsymmetric functions.

Listing 32.5: Reflecting illumination from point lights.

```
Radiance3 App::estimateDirectLightFromPointLights(
    const SurfaceElement& surfel, const Ray& ray){

    Radiance3 L_o(0.0f);

    if (m_pointLights) {
        for (int L = 0; L < m_world->lightArray.size(); ++L) {
            const GLight& light = m_world->lightArray[L];
            // Shadow rays
            if (m_world->lineOfSight(
                surfel.geometric.location + surfel.geometric.normal * 0.0001f,
                   light.position.xyz())) {
                Vector3 w_i = light.position.xyz() - surfel.shading.location;
                const float distance2 = w_i.squaredLength();
                w_i /= sqrt(distance2);

                // Attenuated radiance
                const Irradiance3& E_i = light.color / (4.0f * pif() * distance2);

                L_o += (surfel.evaluateBSDF(w_i, -ray.direction()) * E_i *
                        max(0.0f, w_i.dot(surfel.shading.normal)));
                debugAssert(radiance.isFinite());
            }
        }
    }
    return L_o;
}
```

There are three slightly subtle points highlighted in the code. The first is that we don't ask whether the luminaire is visible from the surface point; as we discussed earlier, we have to ask whether it's visible from a slightly displaced surface point, which we compute by adding a small multiple of the surface normal to the surface-point location. The second is that we make sure that the direction from P to the luminaire and the surface normal at P point in the same hemisphere; otherwise, the surface can't be lit by the luminaire. This test might seem redundant, but it's not, for two reasons (see Figure 32.8). One is that the surface point might be at the very edge of a surface, and therefore be *visible* to a luminaire that's below the plane of the surface. The other is that the normal vector we use in this "checking for illumination" step is the *shading* normal rather than the geometric normal. Since we actually compute the dot product with the shading normal, this can result in smoothly varying shading over a not-very-finely tessellated surface.

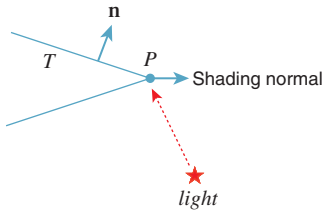


Figure 32.8: P is visible to the light, but not lit by it.

This is another general pattern: During computations of visibility, we'll use the geometric data associated with the surface element. But during computations of light scattering, we'll use `surfel.shading.location`. In general, our representation of the surface point has both geometric and shading data: The geometric data is that of the raw underlying mesh, while the shading data is what's used in scattering computations. For instance, if the surface is displacement-mapped, the shading location may differ slightly from the geometric location. Similarly, while the geometric normal vector is constant across each triangular face, the shading normal may be barycentrically interpolated from the three vertex normals at the triangular face's vertices.

The third subtlety is the computation of the radiance. As we discussed in Chapter 31, if we treat the point luminaire as a limiting case of a small, uniformly emitting spherical luminaire, the outgoing radiance resulting from reflecting this light is a product of a BRDF term, a cosine, and a radiance that varies with the distance from the luminaire; we called that `E_i` in the program. (We've also, as promised, ignored specular scattering of point lights.)

When we turn our attention to *area* luminaires (see Listing 32.6), much of the code is identical. Once again, we have a flag, `m_areaLights`, to determine whether to include the contribution of area lights. To estimate the radiance from the area luminaire, we sample one random point on the source, that is, we form a single-sample estimate of the illumination. Of course, this has high variance compared to sampling many points on the luminaire, but in a path tracer we typically trace many primary rays per pixel so that the variance is reduced in the final image. When testing visibility, we again slightly displace the point on the source as well as the point on the surface. Other than that, the only subtlety is in the estimation of the outgoing radiance. Since our light's `samplePoint` samples uniformly with respect to area, we have to do a change of variables, and include not only the cosine at the surface point but also the corresponding cosine at the luminaire point, and the reciprocal square of the distance between them. By line 23, we've used these ideas to estimate the radiance from the area light scattered at P, *except* for impulse scattering, because `evaluateBSDF` returns only the finite portion of the BSDF.

At line 26 we take a different approach for impulse scattering: We compute the impulse direction, and trace along it to see whether we encounter an emitter, and if so, multiply the emitted radiance by the impulse magnitude to get the scattered radiance.

Listing 32.6: Reflecting illumination from area lights.

```
Radiance3 App::estimateDirectLightFromAreaLights(const SurfaceElement& surfel,
    const Ray& ray){
    Radiance3 L_o(0.0f);
    // Estimate radiance back along ray due to
    // direct illumination from AreaLights
    if (m_areaLights) {
        for (int L = 0; L < m_world->lightArray2.size(); ++L) {
            AreaLight::Ref light = m_world->lightArray2[L];
            SurfaceElement lightsurfel = light->samplePoint(rnd);
            Point3 Q = lightsurfel.geometric.location;

            if (m_world->lineOfSight(surfel.geometric.location +
                  surfel.geometric.normal * 0.0001f,
                  Q + 0.0001f * lightsurfel.geometric.normal)) {
                Vector3 w_i = Q - surfel.geometric.location;
                const float distance2 = w_i.squaredLength();
                w_i /= sqrt(distance2);

                L_o += (surfel.evaluateBSDF(w_i, -ray.direction()) *
                    (light->power()/pif()) * max(0.0f, w_i.dot(surfel.shading.normal))
                    * max(0.0f, -w_i.dot(lightsurfel.geometric.normal)/distance2));
                debugAssert(L_o.isFinite());
            }
        }
        if (m_direct_s) {
            // now add in impulse-reflected light, too.
            SmallArray<SurfaceElement::Impulse, 3> impulseArray;
            surfel.getBSDFImpulses(-ray.direction(), impulseArray);
            for (int i = 0; i < impulseArray.size(); ++i) {
                const SurfaceElement::Impulse& impulse = impulseArray[i];
                Ray secondaryRay = Ray::fromOriginAndDirection(
                    surfel.geometric.location, impulse.w).bumpedRay(0.0001f);
                SurfaceElement surfel2;
                float dist = inf();
                if (m_world->intersect(secondaryRay, dist, surfel2)) {
                    // this point could be an emitter...
                    if (m_emit) {
                        radiance += surfel2.material.emit * impulse.magnitude;
                    }
                }
            }
        }
    }
    return L_o;
}
```

In each case—the impulse and the finite-part reflection of area lights, and the finite-part reflection of point lights—we picked some direction ω_i and multiplied *some* measure of light arriving at P in direction $-\omega_i$ by *some* factor based on the BSDF: either the impulse magnitude or the finite part of the BSDF. It's possible to restructure the code so that the measure of light in each case corresponds to the *biradiance* described in Chapter 14, which helps explain how the first ray tracers, which didn't really use physical units, actually managed to produce good-looking pictures.

At this point we've computed the emissive term of the rendering equation, and the reflected term, at least for light arriving at P directly from luminaires. We now

must consider light arriving at P from all other sources, that is, light from some point Q that arrives at P *having been reflected at* Q *rather than emitted*. Such light is reflected at P to contribute to the outgoing radiance from P back toward the eye. Once again, we estimate this incoming indirect radiance with a single sample. To do so, we use our path-tracing code recursively. We build a ray starting at (or very near) P, going in some random direction ω into the scene; we use our path tracer to tell us the indirect radiance back along this ray, and reflect this, via the BRDF, into radiance transported from P toward the eye. Of course, in this case, we must *not* include in the computed radiance the light emitted directly toward P—we've already accounted for that. We therefore set `includeEmissive` to `false` at line 24. Listing 32.7 show this.

Listing 32.7: Estimating the indirect light scattered back along a ray.

```
1  Radiance3 App::estimateIndirectLight(
2              const SurfaceElement& surfel, const Ray& ray, bool isEyeRay){
3      Radiance3 L_o(0.0f);
4      // Use recursion to estimate light running back along ray
5      // from surfel, but ONLY light that arrives from
6      // INDIRECT sources, by making a single-sample estimate
7      // of the arriving light.
8
9      Vector3 w_o = -ray.direction();
10     Vector3 w_i;
11     Color3 coeff;
12     float  eta_o(0.0f);
13     Color3 extinction_o(0.0f);
14     float  ignore(0.0f);
15
16     if (!(isEyeRay) || m_indirect) {
17         if (surfel.scatter(w_i, w_o, coeff, eta_o, extinction_o, rnd, ignore)) {
18             float eta_i = surfel.material.etaReflect;
19             float refractiveScale = (eta_i / eta_o) * (eta_i / eta_o);
20
21             L_o += refractiveScale * coeff *
22              pathTrace(Ray(surfel.geometric.location, w_o).bumpedRay(0.0001f *
23                  sign(surfel.geometric.normal.dot( w_o)),
24                surfel.geometric.normal), false);
25         }
26     }
27     return L_o;
28 }
```

The great bulk of the work is done in `surfel.scatter()`, which takes a ray r arriving at a point and either absorbs it or determines an outgoing direction r' for it, and a coefficient by which the radiance *arriving* along r' (i.e., the radiance $L(P, -r')$) should be multiplied to generate a single-sample estimate of the scattered radiance at P in direction $-r$.

Before examining the `scatter()` code, let's review that description more closely. First, `scatter()` can be used either in ray/path tracing or in photon tracing. The second use is perhaps more intuitive: We have a bit of light energy arriving at a surface, and it is either absorbed or scattered in one or more directions. The `scatter()` procedure is used to simulate this process. If the absorption at the surface is, say, 0.3, then 30% of the time `scatter()` will return `false`. The other 70% of the time it will return `true` and set the value of ω_o. Given the direction ω_i toward the source of the light, the probability of picking a particular direction ω_o

(at least for surfaces with no mirror terms or transmissive terms) for the scattered light is roughly proportional to $f_s(\omega_i, \omega_o)$. In an ideal world, it would be exactly proportional. In ours, it's generally not, but the returned coefficient contains a $f_s(\omega_i, \omega_o)/p(\omega_o)$ factor, where $p(\omega_o)$ is the probability of sampling ω_o, which compensates appropriately.

What happens if there *is* a mirror reflection? Let's say that 30% of the time the incoming light is absorbed, 50% of the time it's mirror-reflected, and the remaining 20% of the time it's scattered according to a Lambertian scattering model. In this situation, `scatter()` will return `false` 30% of the time. Fifty percent of the time it will return `true` and set ω_o to be the mirror-reflection direction, and the remaining 20% of the time ω_o will be distributed on the hemisphere with a cosine-weighted distribution (i.e., with high probability of being emitted in the normal direction and low probability of being emitted in a tangent direction).

Let's see this in practice, and look at the start of G3D's `scatter` method in Listing 32.8.

Listing 32.8: The start of the scatter method.

```
bool SurfaceElement::scatter
(const Vector3& w_i,
 Vector3&       w_o,
 Color3&        weight_o, // coeff by which to multiply sample in path-tracing
 float&         eta_o,
 Color3&        extinction_o,
 Random&        random,
 float&         density) const {

    const Vector3& n = shading.normal;

    // Choose a random number on [0, 1], then reduce it by each kind of
    // scattering's probability until it becomes negative (i.e., scatters).
    float r = random.uniform();

    if (material.lambertianReflect.nonZero()) {
        float p_LambertianAvg = material.lambertianReflect.average();
        r -= p_LambertianAvg;

        if (r < 0.0f) {
            // Lambertian scatter
            weight_o     = material.lambertianReflect / p_LambertianAvg;
            w_o          = Vector3::cosHemiRandom(n, random);
            density      = ...
            eta_o        = material.etaReflect;
            extinction_o = material.extinctionReflect;
            debugAssert(power_o.r >= 0.0f);

            return true;
        }
    }
    ...
```

As you can see, the material has a `lambertianReflect` member, which indicates reflectance in each of three color bands;[2] the average of these gives a

2. We'll call these "red," "green," and "blue" in keeping with convention, but with no important change in the implementation, we could record five or seven or 20 spectral samples.

probability p of a Lambertian scattering of the incoming light. If the random value r is less than p, we produce a Lambertian-scattered ray; if not, we subtract that probability from r and move on to the next kind of scattering.

Inline Exercise 32.6: Convince yourself that this approach has a probability p of producing a Lambertian-scattered ray.

The actual scattering is fairly straightforward: The `cosHemiRandom` method produces a vector with a cosine-weighted distribution in the hemisphere whose pole is at **n**. The method also returns the index of refraction (both real and imaginary parts) of the material on the **n** side of the intersection point, and a coefficient, (called `weight_o` here) that is precisely the number we'll need to use when we do Monte Carlo estimation of the reflected radiance. (The returned value `density` is *not* the probability density, but a rather different value included for the benefit of other algorithms, and we ignore it.)

The remainder of the scattering code is similar. Recall that the reflection model we're using is a weighted sum of a Lambertian, a glossy component, and a

Listing 32.9: Further scattering code.

```
    Color3 F(0, 0, 0);
    bool Finit = false;

    if (material.glossyReflect.nonZero()) {

        // Cosine of the angle of incidence, for computing Fresnel term
        const float cos_i = max(0.001f, w_i.dot(n));
        F = computeF(material.glossyReflect, cos_i);
        Finit = true;

        const Color3& p_specular = F;
        const float p_specularAvg = p_specular.average();

        r -= p_specularAvg;
        if (r < 0.0f) { // Glossy (non-mirror) case
            if (material.glossyExponent != finf()) {
                float intensity = (glossyScatter(w_i, material.glossyExponent,
                    random, w_o) / p_specularAvg);
                if (intensity <= 0.0f) {
                    // Absorb
                    return false;
                }
                weight_o = p_specular * intensity;
                density = ...

            } else {
                // Mirror

                w_o = w_i.reflectAbout(n);
                weight_o = p_specular * (1.0f / p_specularAvg);
                density = ...
            }

            eta_o = material.etaReflect;
            extinction_o = material.extinctionReflect;
            return true;
        }
    }
```

transmissive component, where the weights sum to one or less. If they sum to less than one, there's some absorption. The weights are specified for R, G, and B, and the sum must be no more than one in each component.

The glossy portion of the model has an exponent that can be any positive number or infinity. If it's infinity, then we have a mirror reflection; otherwise, we have a Blinn-Phong-like reflection, which is scaled by a Fresnel term, F. Listing 32.9 shows this code.

Finally, we compute the transmissive scattering due to refraction, with the code shown in Listing 32.10. The only subtle point is that the Fresnel coefficient for the transmitted light is one minus the coefficient for the reflected light.

Listing 32.10: Scattering due to transmission.

```
    ...
    if (material.transmit.nonZero()) {
        // Fresnel transmissive coefficient
        Color3 F_t;

        if (Finit) {
            F_t = (Color3::one() - F);
        } else {
            // Cosine of the angle of incidence, for computing F
            const float cos_i = max(0.001f, w_i.dot(n));
      // Schlick approximation.
            F_t.r = F_t.g = F_t.b = 1.0f - pow5(1.0f - cos_i);
        }

        const Color3& T0          = material.transmit;

        const Color3& p_transmit  = F_t * T0;
        const float p_transmitAvg = p_transmit.average();

        r -= p_transmitAvg;
        if (r < 0.0f) {
            weight_o     = p_transmit * (1.0f / p_transmitAvg);
            w_o          = (-w_i).refractionDirection(n, material.etaTransmit,
                material.etaReflect);
            density      = p_transmitAvg;
            eta_o        = material.etaTransmit;
            extinction_o = material.extinctionTransmit;

            // w_o is zero on total internal refraction
            return ! w_o.isZero();
        }
    }

    // Absorbed
    return false;
}
```

The code in Listing 32.10 is messy. It's full of branches, and there are several approximations and apparently ad hoc tricks, like the Schlick approximation of the Fresnel coefficient, and the setting of the cosine of the incident angle to be at least 0.001, embedded in it. This is typical of scattering code. Scattering is a messy process, and we must expect the code to reflect this, but the messiness also arises from the challenges of floating-point arithmetic on machines of finite precision. Perhaps a more positive view is that the code will be called many times with

many different parameter values, and it's important that it be robust regardless of this wide range of inputs.

There *is* an alternative, however. If we actually know microgeometry perfectly, and we know the index of refraction for every material, we can compute scattering relatively simply—there's a reflection term and a transmission term, and what's neither reflected nor transmitted is absorbed. As long as the microgeometry is of a scale somewhat larger than the wavelength of light that we're scattering, this provides a complete model. Unfortunately, at present it's an impractical one, for several reasons. First, representing microgeometry at the scale of microfacets requires either an enormous amount of data or, if it's generated procedurally, an enormous amount of computation. Second, if we accurately represent microgeometry, then every surface becomes mirrorlike at a small enough scale. To get the appearance of diffuse reflection at a point P requires that thousands of rays hit the surface near P, each scattering in its own direction. Ray-tracing a piece of chalk suddenly requires a thousand rays per pixel instead of just one! Third, the exact index of refraction for many materials is unknown or hard to measure, especially the coefficient of extinction.

Our representation of scattering by summary statistics like the diffuse coefficient is a way to take this intractable model and make it workable, with only slight losses in fidelity, based on the observation that the precise microgeometry almost never matters in the final rendering; if we render 20 pieces of chalk with the same macrogeometry, they'll all look essentially identical.

There's a third alternative between these two: You can store measured BRDF data. Storing such data, at a reasonably fine level of detail, can be expensive. (If your material has a glossy highlight that resembles the one produced by a Phong exponent of 1000, then the BRDF drops from its peak value to half that value in about 7°, suggesting that you might need to sample at least every 2° to faithfully capture it, requiring about 17,000 samples.) Drawing a direction ω with probability proportional to $\omega \mapsto f_s(\omega_i, \omega)$ is far more problematic, but it is feasible. You might think that you could have the best of both worlds by choosing an explicit parametric representation (e.g., spherical harmonics, or perhaps some generalized Phong-like model) and finding the best fit to the measured data. This is a fairly common practice in the film industry today, and it works well for some materials like metals, but it can produce huge errors for diffuse surfaces when you use common analytic models that fail to model subsurface effects accurately [NDM05]. Nonetheless, it's currently an active area of research, one with considerable promise for simplifying the computation of the scattering integral.

32.5.3 Results and Discussion

Figure 32.9 shows four simple scenes we'll use in evaluating renderers, both drawn and ray-traced. The first, with its diffuse floor and back wall, and brightly colored semidiffuse sphere, provides a nice, simple test of bounding volume hierarchy, visibility, and rendering with reflection but not transmission. Since most of the scattering in the scene is diffuse, it only provides limited testing of scattering from multiple surfaces: We can't visually check multiple interobject reflections the way we could in a scene with 12 mirrored spheres, for instance.

In the second scene, we've added a transparent sphere whose refractive index is somewhat greater than that of air, to let us verify that transmissive

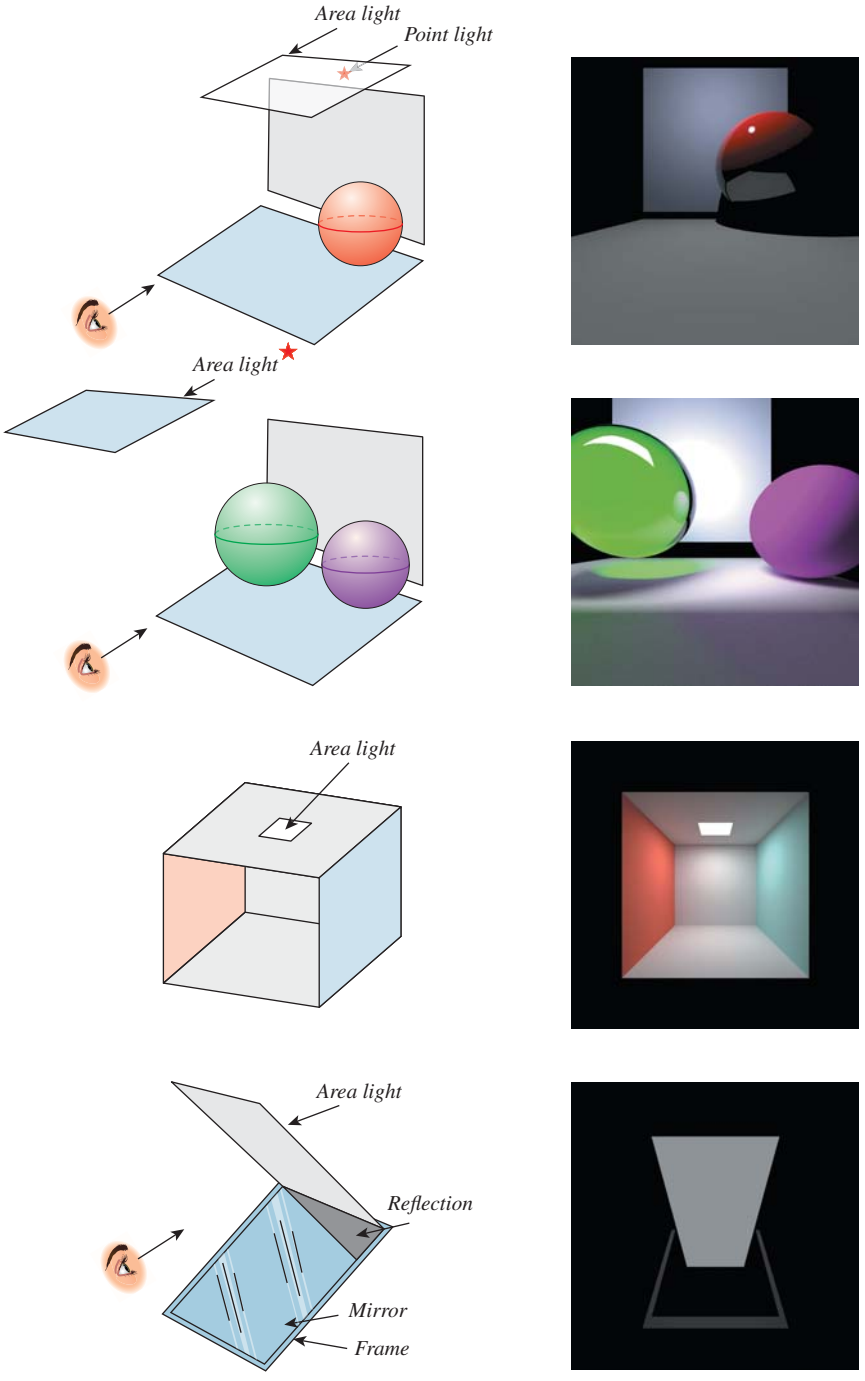


Figure 32.9: Four simple scenes, with their rendered versions shown to the right. The scene at the top is ray-traced (and hence the light comes from the point source; the area source is ignored); the three remaining scenes are path-traced to show the effects of the area luminaires.

scattering is working properly. (By the way, you can add a perfectly transmissive and completely nonreflective sphere with refractive index 1.0 to a scene, and it should make no difference to the scene's appearance. Of course, if your renderer bounds the number of ray-surface interactions, it may have an effect on the rendering nonetheless.) The transparent sphere reflects some light from the other sphere, reflects some light onto it, and generates a diffuse pattern of light on the floor. None of these effects (color bleeding between diffuse surfaces, the caustic, or reflected light on the solid sphere) would be visible in a ray-traced version of the scene.

The third scene is the Cornell box, a standard test scene with diffuse surfaces, and an area light, in which color bleeding and multiple inter-reflections are evident in an accurate rendering, but are missing from the ray tracing.

The final scene consists of a large area light tilted toward the viewer, and a large mirror below it, also tilted toward the viewer, with a diffuse rectangle behind it to form a border for the mirror. The viewer sees not only the light, but also its reflection in the perfect mirror. Together these give the appearance of a single long continuous rectangle.

Figure 32.10 shows a path-traced version of the first scene. There are some obvious differences between this and the ray-traced scene. First, the area light, which we ignored in the ray tracing, has been included in the path tracing. Second, there's *noise* in the path-traced image—everything looks somewhat speckled. We'll return to this presently. Third, the shadows are softer. Light is reflecting from the floor onto the sphere, lighting the lower half somewhat, which in turn helps light the shadowed part of the floor. Fourth, there is color bleeding: The pinkish color on the floor and back wall is from light that's reflected from the sphere.

The softened shadows and color bleeding are what you should expect when you consider how path tracing works. The noise, however, seems like a serious drawback. On the other hand, the ray-traced version exhibits aliasing, especially on the shadow edges. That's because in the ray tracer, the rays from the eye through two nearby pixels end up reflecting from (or refracting through) the sphere in almost parallel directions. In the path tracer, there's a coin toss: About 80% of

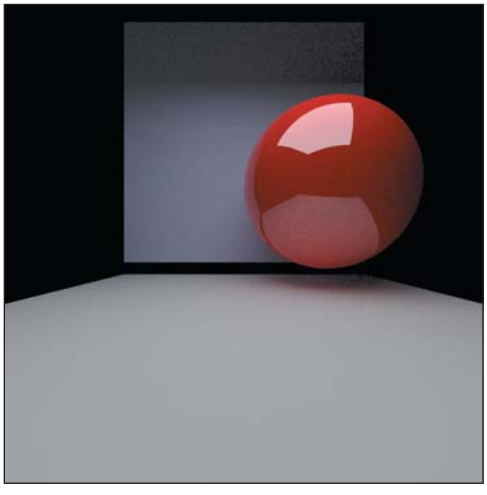


Figure 32.10: A path-traced scene.

the time a ray hitting the sphere, for instance, is reflected in a specific direction, and just as in ray tracing, nearby rays are refracted to nearby rays. But there's also a 20% chance of absorption. If we trace, say, ten primary rays per pixel, it's reasonable to expect seven, eight, nine, or ten of these rays to be reflected (i.e., from zero to four of them to be absorbed). That'll lead to adjacent pixels having quite different radiance sums. To reduce this variance between adjacent pixels, we need to send quite a lot of primary rays (perhaps hundreds or thousands per pixel). You can even use the notion of confidence intervals from statistics to determine a number of samples so large that the fraction of absorbed rays is very nearly the absorption probability so that interpixel variation is small enough to be beneath the threshold of observation. In fact, Figure 32.10 was rendered with 100 primary rays per pixel, and despite this, the reflection of the floor in the red sphere appears slightly speckled. Figure 32.11 shows the speckle more dramatically.

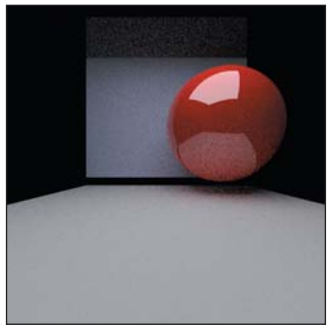


Figure 32.11: Path tracing with ten rays per pixel.

32.6 Photon Mapping

Let's now move on to a basic implementation of photon mapping. Recall that the main idea in photon mapping is to estimate the indirect light scattered from diffuse surfaces by shooting photons[3] from the luminaires into the scene, recording where they arrive (and from what direction), and then reflecting them onward to be further recorded in subsequent bounces, eventually attenuated by absorption or by having the recursion depth-limited. When it comes time to estimate scattering at a point P of a diffuse surface, we search for nearby photons and use them as estimates of the arriving light at P, which we then push through the reflectance process to estimate the light leaving P.

Not surprisingly, much of the technology used in the path-tracing code can be reused for photon mapping. In our implementation, we have built a photon-map data structure based on a **hash grid** (see Chapter 37); as we mentioned in our discussion of photon mapping, any spatial data structure that supports rapid insertion and rapid neighborhood queries can be used instead.

We've defined two rather similar classes, `EPhoton` and `IPhoton`, to represent photons as they are *emitted* and when they arrive; the "I" in `IPhoton` stands for "incoming." An `EPhoton` has a position from which it was emitted, and a direction of propagation, which are stored together in a propagation ray, and a *power*, representing the photon's power in each of three spectral bands. An `IPhoton`, by contrast, has a *position* at which it arrived, a *direction to the photon source* from that position, and a power. Making distinct classes helps us keep separate the two distinct ways in which the term "photon" is used. In our implementation, an `EPhoton` is emitted, and its travels through the scene result in one or more `IPhoton`s being stored in the photon map.

The basic structure of the code is to first build the photon map, and then render the scene using it. Listing 32.11 shows the building of the photon map: We construct an array `ephotons` of photons to be emitted, and then emit each into the scene to generate an array `iphotons` of incoming photons, and store these in the map `m_photonMap`.

3. Recall that a "photon" in photon mapping represents a bit of power emitted by the light, typically representing many physical photons.

Listing 32.11: The large-scale structure of the photon-mapping code.

```
main(){
  set up image and display, and load scene
  buildPhotonMap();
  call photonRender for each pixel
  display the resulting image
}

void App::buildPhotonMap(){
    G3D::Array<EPhoton> ephotons;
    LightList lightList(&(m_world->lightArray), &(m_world->lightArray2), rnd);
    for (int i = 0; i < m_nPhotons; i++) {
        ephotons.append(lightList.emitPhoton(m_nPhotons));
    }

    Array<IPhoton> ips;
    for (int i = 0; i < ephotons.size(); i++) {
        EPhoton ep = ephotons[i];
        photonTrace(ep, ips);
        m_photonMap.insert(ips);
    }
}
```

The `LightList` represents a collection of all point lights and area lights in the scene, and can produce emitted photons from these, with the number of photons emitted from each source being proportional to the power of that source. Listings 32.12 and 32.13 show a little bit of how this is done: We sum the power (in the R, G, and B bands) for each light to get a total power emitted. The probability that a photon is emitted by the *i*th light is then the ratio of its average power (over all bands) to the average of the total power over all bands. These probabilities are stored in an array, with one entry per luminaire.

Listing 32.12: Initialization of the `LightList` *class.*

```
void LightList::initialize(void)
{
    // Compute total power in all spectral bands.
    foreach point or area light
        m_totalPower += light.power() //totalPower is RGB vector

    // Compute probability of emission from each light
    foreach point or area light
        m_probability.append(light.power().average() / m_totalPower.average());
}
```

With these probabilities computed, the only subtlety remaining is selecting a random point on the surface of an area light (see Listing 32.13). If the area light has some known geometric shape (cube, sphere, ...), we can use the obvious methods to sample from it (see Exercise 32.12). On the other hand, if it's represented by a triangle mesh, we can first pick a triangle at random, with the probability of picking a triangle T proportional to the area of T, and then pick a point in that triangle uniformly at random. Exercise 32.6 shows that generating samples uniformly on a triangle may not be as simple as you think.

Inline Exercise 32.7: The `areaLightEmit` code uses `cosHemiRandom` to generate a photon in direction (x, y, z) with probability $\cos(\theta)$, where θ is the angle between (x, y, z) and the surface normal. Why?

Listing 32.13: Photon emission in the `LightList` *class.*

```
1  // emit a EPhoton; argument is total number of photons to emit
2  EPhoton LightList::emitPhoton(int nEmitted)
3  {
4      u = uniform random variable between 0 and 1
5      find the light i with p_0 + ... + p_{i-1} <= u < p_0 + ... + p_i.
6      if (i < m_nPointLights)
7          return pointLightEmit((*m_pointLightArray)[i], nEmitted, m_probability[i]);
8      else
9          return areaLightEmit((*m_areaLightArray)[i - m_nPointLights], nEmitted,
10                              m_probability[i]);
11 }
12
13 EPhoton LightList::pointLightEmit(GLight light, int nEmitted, float prob){
14     // uniformly randomly select a point (x,y,z) on the unit sphere
15     Vector3 direction(x, y, z);
16     Power3 power = light.power() / (nEmitted * prob);
17     Vector3 location = location of the light
18     return EPhoton(location, direction, power);
19 }
20
21 EPhoton LightList::areaLightEmit(AreaLight::Ref light, int nEmitted, float prob){
22     SurfaceElement surfel = light->samplePoint(m_rnd);
23     Power3 power = light->power() / (nEmitted * prob);
24     // select a direction with cosine-weighted distribution around
25     // surface normal. m_rnd is a random number generator.
26     Vector3 direction = Vector3::cosHemiRandom(surfel.geometric.normal, m_rnd);
27
28     return EPhoton(surfel.geometric.location, direction, power);
29 }
```

What remains is the photon tracing itself (see Listing 32.14). We use a G3D helper class, the `Array<IPhoton>`, to accumulate incident photons. This class has a `fastClear` method that simply sets the number of stored values to zero rather than actually deallocating the array; this saves substantial allocation/deallocation overhead. The `photonTraceHelper` procedure keeps track of the number of bounces that the photon has undergone so far so that the bounce process can be terminated when this reaches the user-specified maximum. Note that in contrast to Jensen's original algorithm, we store a photon at every bounce, whether it's diffuse or specular. For estimating radiance at surface points with purely impulsive scattering models (e.g., mirrors), these photons will never be used, however, so there's no impact on the results.

Once again, there are no real surprises in the program. The `scatter` method does all the work. We've hidden one detail here: The `scatter` method *should* be different from the one used in path tracing. If we're only studying reflection, and only using symmetric BRDFs (i.e., all materials satisfy Helmholtz reciprocity), then the two scattering methods are the same. But in the case of asymmetric scattering (such as Fresnel-weighted transmission), it's possible that the probability that a photon arriving at P in direction ω_i scatters in direction ω_o is completely

Listing 32.14: Tracing photons, which is rather similar to tracing rays or paths.

```
void App::photonTrace(const EPhoton& ep, Array<IPhoton>& ips) {
    ips.fastClear();
    photonTraceHelper(ep, ips, 0);
}

/**
   Recursively trace an EPhoton through the scene, accumulating
   IPhotons at each diffuse bounce
*/
void App::photonTraceHelper(const EPhoton& ep, Array<IPhoton>& ips, int bounces) {
    // Trace an EPhoton (assumed to be bumped if necessary)
    // through the scene. At each intersection,
    // * store an IPhoton in "ips"
    // * scatter or die.
    // * if scatter, "bump" the outgoing ray to get an EPhoton
    // to use in recursive trace.

    if (bounces > m_maxBounces) {
        return;
    }

    SurfaceElement surfel;
    float dist = inf();
    Ray ray(ep.position(), ep.direction());

    if (m_world->intersect(ray, dist, surfel)) {
        if (bounces > 0) { // don't store direct light!
            ips.append(IPhoton(surfel.geometric.location, -ray.direction(), ep.power()));
        }
        // Recursive rays
        Vector3 w_i = -ray.direction();
        Vector3 w_o;
        Color3 coeff;
        float eta_o(0.0f);
        Color3 extinction_o(0.0f);
        float ignore(0.0f);

        if (surfel.scatter(w_i, w_o, coeff, eta_o, extinction_o, rnd, ignore)) {
            // managed to bounce, so push it onwards
            Ray r(surfel.geometric.location, w_o);
            r = r.bumpedRay(0.0001f * sign(surfel.geometric.normal.dot( w_o)),
                 surfel.geometric.normal);
            EPhoton ep2(r, ep.power() * coeff);
            photonTraceHelper(ep2, ips, bounces+1);
        }
    }
}
```

different from the probability that one arriving in direction ω_o scatters in direction ω_i. Our surface really needs to provide two different scattering methods, one for each situation. In our code, we've used the same method for both. That's wrong, but it's also very common practice, in part because the effects of making the code right are (a) generally small, and (b) generally not something we're perceptually sensitive to. You might want to spend a little while trying to imagine a scene in which the distinction between the two scattering rules matters.

One difference between photon propagation and radiance propagation is that at the interface between media with different refractive indices, the radiance changes

(because a solid angle on one side becomes a different solid angle on the other), while for photons, which represent power transport through the scene, there is no such change. Thus, there's no η_i/η_o factor in the photon-tracing code.

Having built the photon map, we must render a picture based on it. Our first version will closely resemble the path-tracing code, in the sense that we'll break up the computation into direct and indirect light, and handle diffuse and impulse scattering individually. We'll use a ray-tracing approach (i.e., recursively trace rays until some fixed depth); making the corresponding path-tracing approach is left as an exercise for the reader. The photon map is used only to estimate the diffusely reflected indirect light arriving at a point.

Computing the light arriving at pixel (x, y) of the image, using a ray-tracing and photon-mapping hybrid, is the job of the `photonRender` procedure, shown in Listing 32.15, along with some of the methods it calls.

Listing 32.15: Generating an image sample for pixel (x, y) from the photon map.

```
void App::photonRender(int x, int y) {
    Radiance3 L_o(0.0f);
    for (int i = 0; i < m_primaryRaysPerPixel; i++) {
        const Ray r = defaultCamera.worldRay(x + rnd.uniform(),
            y + rnd.uniform(), m_currentImage->rect2DBounds());
        L_o += estimateTotalRadiance(r, 0);
    }
    m_currentImage->set(x, y, L_o / m_primaryRaysPerPixel);
}

Radiance3 App::estimateTotalRadiance(const Ray& r, int depth) {
    Radiance3 L_o(0.0f);
    if (m_emit) L_o += estimateEmittedLight(r);

    L_o += estimateTotalScatteredRadiance(r, depth);
    return L_o;
}

Radiance3 App::estimateEmittedLight(Ray r){
    ...declarations...
    if (m_world->intersect(r, dist, surfel))
        L_o += surfel.material.emit;
    return L_o;
}
```

To generate a measurement at pixel (x, y) we shoot `m_primaryRaysPerPixel` into the scene, and estimate the radiance returning along each ray. It's possible that a ray hits a light source; if so, the source's radiance (`EmittedLight`) must be counted in the total radiance returning along the ray, along with any light reflected from the luminaire.

We've ignored the case where the ray hits a point luminaire (point sources have no geometry in our scene descriptions, so such an intersection is never reported). There are two reasons for this. The first is that the intersection of the ray and the point light is an event with (mathematical) probability zero, so in an ideal program with perfect precision, it should never occur. This is a frequently used but somewhat specious argument: First, the discrete nature of floating-point numbers makes probability-zero events occur with very small, but nonzero, frequency. Second, models like point-lighting are usually taken as a kind of "limit" of nonzero-probability things (like small, disk-shaped lights); if the limit is to make any sense,

the effect of the limiting case should be the limit of the effects in the nonlimiting cases. If the disk-shaped lights are given values that increase with the inverse square of the radius, then these nonlimiting cases may produce nonzero effects, which should show up in the limiting case as well.

The other, far more practical, reason for not letting rays hit point lights is that point lights are an abstract approximation to reality, used for convenience, and if you want them to be visible in your scene you can model them as small spherical lights. In our test cases, there are no point lights visible from the eye, so the issue is moot.

In general, not only might light be emitted by the place where the ray meets the scene, but also it may be scattered from there as well. The `estimateTotalScatteredRadiance` procedure handles this (see Listing 32.16) by summing up the direct light, impulse-scattered indirect light, and diffusely reflected direct light.

Listing 32.16: Estimating the total radiance scattered back toward the course of the ray r.

```
Radiance3 App::estimateTotalScatteredRadiance(const Ray& r, int depth){
    ...
    if (m_world->intersect(r, dist, surfel)) {
        L_o += estimateReflectedDirectLight(r, surfel, depth);
        if (m_IImp || depth > 0) L_o +=
             estimateImpulseScatteredIndirectLight(r, surfel, depth + 1);
        if (m_IDiff || depth > 0) L_o += estimateDiffuselyReflectedIndirectLight(r, surfel);
    }
    return L_o;
}
```

In each case, there's a Boolean control for whether to include this aspect of the light. We've chosen to apply this control *only* to the first reflection (i.e., if `m_IImp` is false, then impulse-scattered indirect light is ignored only when it goes directly to the eye).

Listing 32.17: Impulse-scattered indirect light.

```
Radiance3 App::estimateImpulseScatteredIndirectLight(const Ray& ray,
        const SurfaceElement& surfel, int depth){
    Radiance3 L_o(0.0f);

    if (depth > m_maxBounces) {
        return L_o;
    }

    SmallArray<SurfaceElement::Impulse, 3> impulseArray;

    surfel.getBSDFImpulses(-ray.direction(), impulseArray);
    foreach impulse
        const SurfaceElement::Impulse& impulse = impulseArray[i];

        Ray r(surfel.geometric.location, impulse.w);
        r = r.bumpedRay(0.0001f * sign(surfel.geometric.normal.dot( r.direction())),
                surfel.geometric.normal);
        L_o += impulse.magnitude * estimateTotalScatteredRadiance(r, depth + 1);

    return L_o;
}
```

The direct-lighting computation is essentially identical to that in the path tracer, except that we estimate the direct light from each area light with several (`m_diffuseDirectSamples`) samples rather than relying on multiple primary rays to ensure adequate sampling. (The alternative is quite viable, however.) The similarity is so great that we omit the code. This leaves only the impulse-scattered and diffusely reflected indirect light to consider. The first is easy, and it closely resembles the corresponding path-tracing code: We simply compute the total arriving radiance for each impulse recursively, and sum (see Listing 32.17).

This leaves only the computation of diffusely reflected indirect light, which is where the photon map finally comes into play (see Listing 32.18). Central to this code is the photon map's *kernel,* the function that says how much weight to give each nearby photon's contribution.

Listing 32.18: The photon map used to compute diffusely reflected indirect light.

```
Radiance3 App::estimateDiffuselyReflectedIndirectLight(const Ray& r,
       const SurfaceElement& surfel){
    Radiance3 L_o(0.0f);
    Array<IPhoton> photonArray;
    // find nearby photons
    m_photonMap.getIntersectingMembers(
       Sphere(surfel.geometric.location, m_photonMap.gatherRadius()), photonArray);

    foreach photon p in the array
        const float cos_theta_i =
        L_o += p.power() * m_photonMap.kernel(p.position(), surfel.shading.location) *
             surfel.evaluateBSDF(p.directionToSource(), -r.direction(), finf());
    return L_o;
}
```

We sum up the contributions of all sufficiently nearby photons, weighting them by our kernel and pushing each one through the BSDF to determine the light it contributes traveling toward the source of the ray *r*.

Note that what is being multiplied by the BRDF term is an area-weighted radiance, which is what we called **biradiance** in Chapter 14.

32.6.1 Results and Discussion

Naturally, as we discussed in Chapter 31, there's an interaction between the gather radius, the number of photons stored in the photon map, and the quality of the estimated radiance. When we're estimating the reflected indirect radiance at *P*, if we use a **cylinder kernel** (a photon is counted if it's within some distance of *P*, and ignored otherwise) and there are no photons within the gather radius of *P*, the radiance estimate for that ray will be zero. This can happen if the gather radius is too small or if there are too few photons. How many photons are enough? Well, we'd generally like nearby pixel rays hitting the same surface to get similar radiance estimates. But one ray might be within the radius of 20 photons, while the nearby ray might be within the radius of 21 photons. Even if all the photons share the same incoming power, the second ray will get a radiance that's 5% greater. To reduce the noise in areas of constant radiance, it appears that we might need hundreds of photons within the gather radius of *P*.

There are several ways to improve the results without needing that many photons. One is to alter the photon map's **kernel**—the weighting function, centered

Figure 32.12: The Cornell box rendered with 100, 10,000, and 1,000,000 photons.

at P, that determines how much each photon's contribution should matter in the radiance estimate. If we change from the cylinder kernel to a conelike function (photons near the gather radius have their contributions reduced to near zero), then each arrival or departure of a photon within the gather radius (as we vary P) has a gradual impact. This is the approach used in our renderer.

A very different approach is described by Jensen: Rather than gathering photons within a fixed radius, we enlarge the radius enough to get a certain number of photons, and then use the area of the resultant disk in the conversion from power (stored at incoming photons) to outgoing radiance.

The ray-trace-with-photon-mapping renderer we've written has several parameters (the number of samples to use to estimate lighting from area lights, the radius of the kernel, the number of photons to shoot into the scene), each of which has an effect on the final result. Figure 32.12 shows that with only 100 photons, the Cornell box scene looks blotchy, because most points in the scene are not within the kernel radius of any photons at all. With one million photons, each point is near thousands of photons, and the estimate of diffusely reflected indirect light is very smooth. (We've used a large number of samples for area lights to reduce that source of variation in the image.)

With a larger kernel radius, the low-photon-count image would look smoother, but then distant photons would affect the appearance of the scene at any point. Clearly there's a tradeoff between photon count N and kernel radius r. In Jensen's original photon-mapping algorithm, the radiance estimate was provided by using a fixed *number, k*, of photons, enlarging the radius as necessary to find that many photons. This has the advantage of **scale invariance**—if you double the scene size, you needn't change anything—but it still leaves the problem of choosing N and k.

The number of samples used in estimating direct lighting from area lights also has an impact on noise in the image. Figure 32.13 shows this, again using the Cornell box. With only one sample per light, the image is very noisy; with 100, the noise is lower than that of the photon-mapped estimate of reflected indirect radiance in Figure 32.12.

Inline Exercise 32.8: In the one-sample-per-source image, how is the noisiness correlated with the brightness, and why? In which areas of the image is the human visual system most sensitive to the noise?

It's typical in renderers like this one to use many primary rays per pixel. If we use 100 primary rays, then we need not use very many rays (per primary ray) to estimate direct lighting from area sources. We still, however, as Inline Exercise 32.8 shows, must address the noise in *some* way, particularly if there are dark areas in the scene. Fortunately in the case of the Cornell box, indirectly

Figure 32.13: One-sample-per-pixel rendering of the emitted direct light from area lights only, using 1, 10, and 100 samples to estimate the light from the area source.

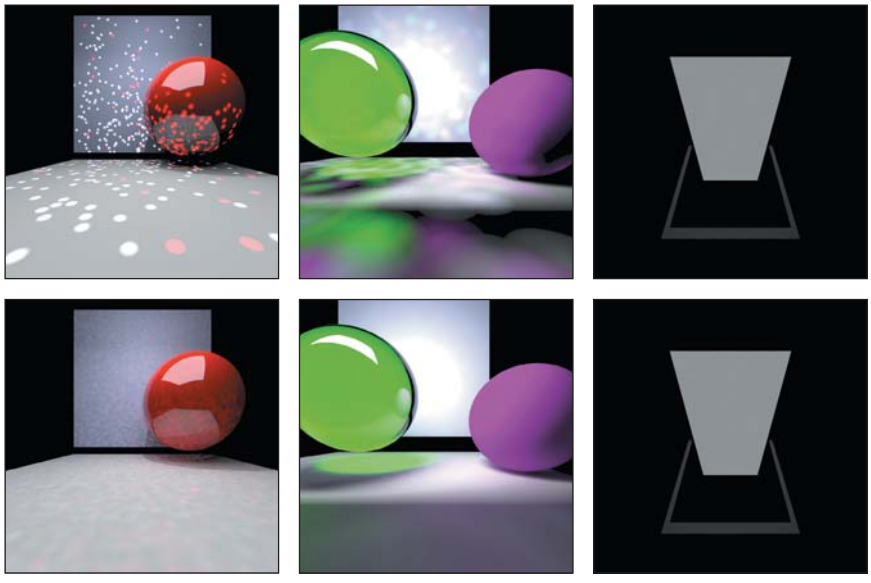


Figure 32.14: The three other test scenes, rendered with 100 primary rays per pixel, ten thousand (top row) and one million (bottom row) shot photons, and one sample per source for area lights.

reflected light dominates the direct light near the top, and so the noise there ends up less perceptually significant when we include indirect light.

Looking at the other test scenes (see Figure 32.14), we see that photon mapping with one million photons handles the first scene very well. The third scene, with the transparent sphere, exhibits a caustic highlight beneath the sphere. In the fourth scene, the emitting luminaire is (as a reflector) completely Lambertian and black. The mirror is pure specular, and the panel beneath the mirror is dark and Lambertian. Even with one million photons shot, only a few photons end up in the scene, and they're used only in estimating the brightness of the panel beneath the mirror, which is overwhelmingly determined by direct lighting, so the computation of the photon map was almost pointless.

In all versions of photon mapping, we're converting from *samples* of the arriving power near P to an *estimate* of that power *at* P. This is extremely closely related to the problem of **density estimation** in statistics. The most basic form of density estimate is **nearest neighbor,** in which the value at P is taken to be

the value at the sample nearest to P. For a continuous density, this converges to the correct value as the number of samples gets large, but the convergence is not exactly the kind we'd like: For any fixed set of samples, the nearest-neighbor estimate of density is piecewise-constant. (On a plane, the regions of constancy are the Voronoi cells associated to the samples.) This leads to a great many discontinuities. On the other hand, as the number of samples increases, the values in adjacent cells get closer and closer, so while the set of discontinuities grows, the magnitude of the discontinuities decreases [WHSG97].

There are many other forms of density estimation aside from nearest-neighbor interpolation. Weighted averaging by some filter kernel is one; Jensen's method of gathering a fixed number of samples and then dividing by a value that depends on those samples is another. Whole books have been written on density estimation, and the statistics and mathematics quickly become increasingly complex as the sophistication of the methods increases.

32.6.2 Further Photon Mapping

The hybrid ray-tracing-photon-mapper approach described above is very basic. Just as we factored out direct lighting, computing it with ray-tracing rather than the photon map, it's possible to build special photon maps that contain only photons that have passed through particular sequences of scattering, such as caustics and shadows (carefully omitting these photons from the generic photon map so that they're not double-counted). Such specialized maps can be used to more accurately generate such phenomena, at the cost of ever-growing code complexity.

There is one technique, however, that applies not just to photon mapping, but to many algorithms: a **final gather** step, in which the incoming radiance at the eye-ray/scene-intersection point is estimated by tracing several rays from that intersection and using some estimation technique at the *secondary* intersections to determine the radiance along those rays; these secondary estimates are then combined to form an estimate along the eye ray. For instance, if the eye ray meets the world at P, we trace 20 rays from P that meet the world at locations $Q_1, \ldots, Q_{20}$. At each Q_i, we can use our photon map to estimate the radiance leaving Q_i in the direction toward P, and then combine these 20 samples of arriving radiance at P to get the radiance exiting from P toward the eye. By carefully selecting the directions along which to sample, we can produce an estimate of the outgoing radiance at P that has far fewer artifacts than would arise from the direct photon-map estimate. For instance, if the photon map is using a disk-shaped reconstruction filter, and not very many photons, there will be many sharp discontinuities in the reconstructed radiance estimates used at the points Q_i. But when these are averaged by the reflectance equation at P to produce the outgoing radiance there, the result is far fewer artifacts in the final rendering.

The alteration in the code is quite minor, as shown in Listing 32.19: The code for estimating the diffusely reflected indirect light at the point P gets a new argument—`useGather`—that is set to `true` for primary rays but `false` for all subsequent ones. When it's false, we use the photon map as above. But when it's true, we essentially perform one-level ray tracing, with a large number of secondary rays, using `photonRender` to estimate the incoming radiance along these.

Listing 32.19: Using final gathering to improve the estimate of indirect radiance arriving at a point.

```
Radiance3 App::estimateDiffuselyReflectedIndirectLight(..., boolean useGather)
    // estimate arriving radiance at "surfel" with final gather.
    Radiance3 L(0.0f);

    if (!useGather)
       ... use the previous photon-mapping code ...
    else {
        for (int i = 0; i < m_gatherRaysPerSample; i++) {
            // draw a cosine-weighted sample direction from the surface point
            Vector3 w_i = -r.direction();
            Vector3 w_o = Vector3::cosHemiRandom(surfel.geometric.normal, rnd);
            Ray gatherRay = Ray(surfel.geometric.location, w_o).bumpedRay(...)

            Color3 coeff;
            L += π * surfel.evaluateBSDF(w_i, w_o, finf()) *
                 estimateTotalScatteredRadiance(gatherRay, depth+1, false);
        }
    }
    return L / m_gatherRaysPerSample;
}
```

The improvement in results is dramatic, at least if we're willing to use a large number of rays in our final gathering. If we're typically averaging contributions from 20 nearby photons in the nongather version of the code, we should be using at least 20 gather rays to estimate the radiance in the gather version. Figure 32.15 shows the effect of the final gather.

32.7 Generalizations

Because photon mapping provides a consistent estimate of reflected indirect radiance, we can take any rendering algorithm that needs such an estimate (ray tracing, path tracing) and replace its estimator with a photon-mapping implementation. A similar argument allows one to replace direct evaluation by a final-gather step. Similarly, it's possible to break up the rendering integral into various pieces (as we did for impulse-scattered indirect light, diffusely scattered direct light, etc.), and

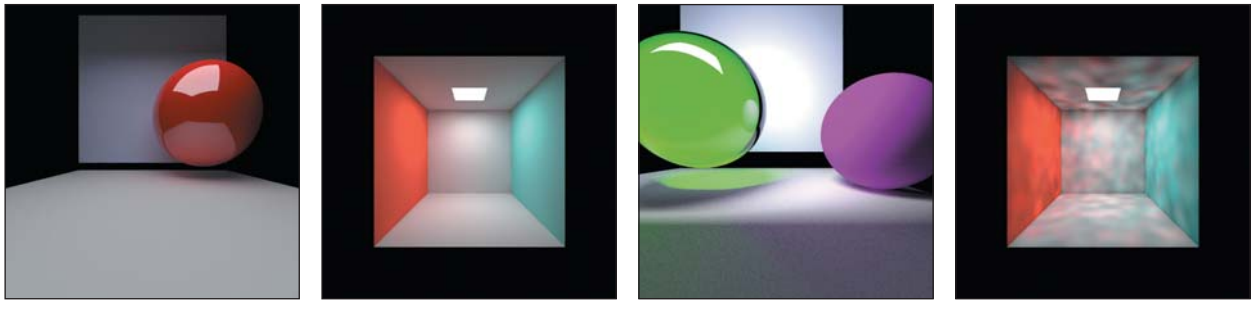


Figure 32.15: Final gathering with 30 samples in our first three test scenes. The photon maps were generated with 10,000 shot photons, resulting in 1,100 photons (first scene) to 7,900 photons (second scene). The last image is the Cornell box, rendered with the same parameters but no final gather. The improvement is substantial.

estimate each part separately. In some cases, direct *evaluation* rather than estimation is possible (e.g., mirror reflection of direct light from area lights); removing this component of the integrand makes the remaining scattered light a smoother function of the outgoing direction and/or the scattering location, and thus allows better stochastic estimation with fewer samples.

Certain classes of illumination and reflection are amenable to particular approaches. For instance, it's very difficult to use Monte Carlo methods based on ray tracing from the eye to render a scene that's lit by tiny luminaires: The chance that a ray (primary or scattered) will hit the light is so small that the variance in the radiance estimate ends up large. On the other hand, if we trace rays from the luminaire to build up a photon map, it becomes easy to estimate the diffusely scattered radiance from small sources, or even scattering along light paths of the form LS^+DE. A dual situation is the rendering of area lights that are multiply scattered by highly specular surfaces. It's very difficult to pick a point of such a light, and a direction for emission, with the property that the resultant light path eventually reaches the eye. Such a situation is far more amenable to ray tracing. To handle both cases, you find you want to trace both from the eye and from luminaires, and you're in the realm of bidirectional path tracing [LW93, Vea97]. Whether you choose to glue paths together or use a density-estimation strategy like photon mapping depends on the classes of paths you encounter. In fact, it's natural to begin thinking that for each possible path type, you might want to choose different ways of working with those paths, and then combine the results at the end. Choosing a reasonable way to combine the multiple sampling approaches (or, thinking in the opposite direction, a reasonable way to break up the integrand or domain of integration) leads quite naturally to methods similar to those used for Metropolis light transport.

Broadly speaking, there's a whole collection of possible approaches that you can take in building a rendering algorithm, and you can put the pieces together in many different ways. Considerations of classes of phenomena that are important to you (caustics, shadows) may make one method preferable over another. Considerations of efficiency, either in the big-O running time of a procedure, or in coherence of memory access or careful use of bandwidth, may also be an influence. Don't be constrained by what others have done: Evaluate the specific rendering problem you need to solve, and then combine techniques as necessary to optimize.

32.8 Rendering and Debugging

In Chapter 5 we discussed how very sensitive the human visual system can be to certain kinds of artifacts in images. Renderers give you the opportunity to put that sensitivity to work. Not only do they provide millions of parallel executions of the same bit of code (typically one or more executions per image pixel), but it's easy to construct scenes that have a natural coherence to them so that any anomaly stands out.

Let's look at several examples to see how you can debug a renderer. Suppose you're writing a path tracer, and you find that increasing the number of primary rays causes the image to get dimmer, but does not improve the aliasing artifacts around shadows. The dimming of the image suggests that you're dividing by the number of primary rays, as you should, but that you're failing to accumulate radi-

ance in proportion to the number of primary rays. Probably the part of your code shown in Listing 32.20 has `L = estimateTotalRadiance(...)` rather than `L += estimateTotalRadiance(...)`. That accounts for the presence of the aliasing artifacts: You're still working with only one sample per pixel!

Listing 32.20: Averaging several primary rays.

```
1     for (int i = 0; i < m_primaryRaysPerPixel; i++) {
2         const Ray r = defaultCamera.worldRay(x + rnd.uniform(), y + rnd.uniform(),..)
3         L += estimateTotalRadiance(r, 0);
4     }
5     m_currentImage->set(x, y, L / m_primaryRaysPerPixel);
```

Suppose, on the other hand, that increasing the number of primary samples causes the image to grow *brighter*. Then perhaps you're accumulating radiance, but failing to divide by the number of primary rays.

Now suppose that your objects mostly look good, but one sphere seems to be missing its left third—you can see right through where that part of the sphere should be. What could be wrong? Well, that's a failure to correctly compute the intersection of a ray with the world, so it's got to be a problem with the bounding volume hierarchy (BVH) if you're using one. The fact that it looks as if the sphere was chopped off by a plane suggests that some bounding-plane test is failing. Perhaps switching to a different BVH will get you the correct results, and thus help you diagnose the problem in the BVH code. By the way, if some plane has just a single *line* of pixels that you can see through (or perhaps a line where you see through just a few of the pixels), it suggests a failure of a floating-point comparison: Perhaps one side of a dividing plane is using a less-than test while the other is using a greater-than test, and the few pixels where the test reveals *equality* aren't handled by either side of the plane (see Figure 32.16).

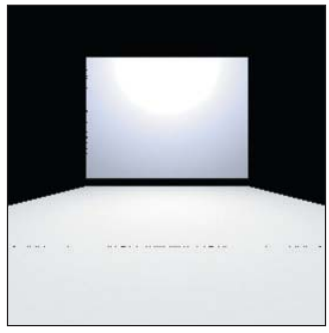


Figure 32.16: Floating-point comparison failure in BVH.

Most Monte Carlo rendering approaches produce high-variance results when you have relatively few samples per pixel. But if you render a Cornell box, for instance, then a typical secondary ray will hit one of the sides of the box, all of which are of comparable brightness. Sure, a ray *could* go into one of the dark corners, and some rays will come out of the front of the box into empty space. But in general, if there are a few secondary rays per pixel, the resultant appearance should be fairly smooth. If you find yourself confronting a "speckled" rendering like the one shown in Figure 32.17 (produced with one primary ray per pixel) you're probably confronting a visibility problem. For contrast, Figure 32.18 shows the same rendering with the visibility problem fixed; there's still lots of noise from the stochastic nature of the path tracer, but no really dark pixels.

Figure 32.17: Speckles in a rendering.

The path for diagnosis is relatively simple. If you eliminate all but direct lighting and still have speckles, then *some* rays from the hit point to the area-light point must be failing their visibility test. There are two possibilities. The first is that perhaps the points generated on the area light are actually not visible. If you've made a small square hole in the ceiling and placed a large square light slightly above it, then many random light points won't be visible from the interior. This is just a modeling mistake. On the other hand, perhaps the light-sampling code has a bug, and the light points being generated do not actually lie on the light itself. The other possibility is that the hit point and light point really are *not* visible from each other because of a failure to bump one or both. Figure 32.17 was generated by removing the `bumpedRay` calls in the path tracer, for instance. The failure of an unbumped

Figure 32.18: Noise in a rendering.

ray to see some other point is typically the result of some floating-point computation going wrong: A number you expected to be slightly greater than zero was in fact slightly less. The dependence of this error on the point locations is likely to be tied to the floating-point representation, and to generate random results like speckle. But sometimes there's a strong enough dependence on a single parameter that there's regularity in the results. Figure 32.19 shows how the floor of the Cornell box is irregularly illuminated (a triangle in the back-right corner is too dark) when we fail to bump just the light-point position.

Figure 32.19: A diagonal stripe on the floor from failed visibility testing.

Suppose that your photon mapper shows nice, smooth gradations of reflected light in the Cornell box, but there's no color bleeding: The left side of the floor isn't slightly red and the right side isn't slightly blue. Think for a moment about what might be wrong, and then read on.

One thing to ask is, "Are the photons correct?" You can plot each photon as a point in your scene to verify that they're well scattered, and you can even plot each one as a small arrow whose base is at the photon location, and which points toward the source of the light. If all the photons point toward the area light, there's a problem: You're not getting any scattered photons in your photon map. (You're also storing first-hit photons, which may be intentional, but if you're computing direct illumination in a separate step, as we did, it's a mistake: You'll end up double-counting the direct illumination!) Interpolating values from the photons you have will tend to give smoothly varying light, however. But you won't get color bleeding. Let's assume, though, that the photons are well scattered, only store indirect light, and seem to get their indirect light from many different places.

A photon on the floor near the red wall probably got there by having light hit the red wall and reflect to the floor. The problem *has* to be in the reflection process, that is, the multiplication by the BRDF and a cosine. Since the magnitude looks right, the absorption/reflection part of the code must be right. But the spectral distribution of the outgoing photon's power is evidently wrong. What color *should* that photon be? Fairly red! By drawing your photons as small disks colored by the color of the arriving light, you can rapidly tell whether they're correct or not. When you discover that all your photons are white, you've found the problem: During multiple bounces, the photon power wasn't being multiplied by the color of the surface from which it was reflecting.

As another example, suppose that you've decided to improve your photon mapper with a final-gather step. You wisely keep the no-gather part of the program, and include a checkbox in the user interface to determine whether gathering is used or not. When you switch from no-gather to gather, the picture looks almost the same, but a bit dimmer. If you turn off direct light, it's evidently *a lot* dimmer. In fact, by inspecting and comparing individual pixels, you find that the pixel values are all dropping by a factor of about three. Once again, think briefly about where the error must be, and then read on.

When you encounter a number near 3 in a renderer, surprisingly often it's really π. (Similarly, a factor of 10 is often $\pi^2 \approx 9.87$.) In this case, when this problem arose for one author it was because he used cosine-weighted samples for his gathering, multiplied each by the appropriate $f_s(\omega_i, \omega_o)$ factor, but not the cosine $(\omega_i \cdot \mathbf{n})$, because the cosine was included in the sampling weights, and then averaged the results. The difficulty was the failure to divide by π (the result of doing a cosine-weighted sampling of the constant function 1 on the upper hemisphere).

Suppose that in path-tracing our second model—the one with the glass sphere—we'd seen the reflection of the adjacent solid sphere and of the area light

on the surface of the glass sphere, but that there was no sign of any transmitted light. There are many possible problems. Maybe the surface element's method for returning all BSDF impulses is failing to return the transmissive ray. Maybe every transmitted ray is somehow suffering total internal reflection. Maybe the number of bounces is being truncated at two, and since light would have to bounce off the back wall, then through two air-glass interfaces, and to the eye, we're never seeing it. How should you debug?

First, it's very useful to be able to trace a single ray, using the pixel coordinates as arguments, as in `pathTrace(int x, int y)`. You can identify a single pixel in which the sphere is visible, add code to path-trace that one pixel, and use a debugger to stop in that code at the first intersection. You check the number of impulses returned, and find that there are two. One has a positive z-component (it's a reflection back somewhat toward the eye), and the other has a negative z-component (it's transmission into the sphere). (Here you can see the advantage of having a view that looks along the z-axis, at least in your test program.)

Second, continuing your debugging, following that second impulse, you find that it does, in fact, intersect the green glass sphere a second time. Multiple scattering seems to be working fine. But the second intersection is surprisingly near the first one—all three coordinates are almost the same. If you chose, as your (x, y) pixel, one that shows a point near the center of the sphere's image in the picture, then you'd expect the first and second intersections to be on the front and then the back of the sphere, nearly a diameter apart. What's wrong?

Once again, the problem has to do with bumping. It's true that the transmitted ray needs to be bumped in the direction of the surface normal, but it needs its starting point to be bumped *into* the sphere rather than *out* of it. To determine which way a ray should be bumped, we need to know which side of the surface it's on. That gets tested with a dot product between the ray direction and the surface normal. If your scattering code says

```
1 Ray r(surfel.geometric.location, impulse.w);
2 r = r.bumpedRay(0.0001f, surfel.geometric.normal);
```

then every recursive ray will be bumped toward the exterior of the sphere. Instead, you need to write:

```
1 Ray r(surfel.geometric.location, impulse.w);
2 r = r.bumpedRay(0.0001f *
3     sign(surfel.geometric.normal.dot(r.direction())),
4     surfel.geometric.normal);
```

In general, debugging by following rays is quite difficult. It helps to have scenes in which things are simple. A scene with one plane, at $z = -20$, and one sphere, of radius 10, centered at the origin, is easy to work with: Whenever you have a ray-scene intersection, it's obvious which surface you're on, and if your intersection point doesn't have $z = -20$, it's easy to mentally add up $x^2 + y^2 + z^2$ to see whether it's approximately 10. It's even better to have scenes so simple that you know the exact answer you expect to get at any point. That's why, for instance, our fourth scene consists of an area light that's completely absorptive, and a mirror (we added the frame to the mirror to make the scene a little easier to understand visually, but removed it during debugging). An eye ray is going to hit the light (for which the computed radiance back along the ray will be the emitted radiance of the light) or miss the scene completely, or hit the mirror, from which it will reflect

in a simple fashion 100% of the time: Because the mirror has $(0, 1, 1)$ as its normal, an incoming ray in direction (x, y, z) becomes an outgoing ray in direction $(x, -z, -y)$. It's easy to mentally work through any such interaction. And if we choose a pixel at the center of the scene, then all x-coordinates will be very near 0 and can be neglected.

Doubtless you'll develop your own approaches to debugging, but because rendering code is often closely tied to particular phenomena, an approach in which it's easy to turn on or off certain parts of computed radiance, and to reason about what remains, makes for much easier debugging.

32.9 Discussion and Further Reading

As we promised at the start of this chapter, we've described basic implementations of a path tracer and a photon-map/ray-tracing hybrid, showing some design choices and pitfalls along the way. Each renderer produces an array containing radiance values, the value at pixel (x, y) being an average of the radiance values for eye rays passing through a square centered at (x, y), whose side length is one interpixel spacing. This models a perfect-square radiance sensor, which is a fair approximation of the CCD cells of a typical digital camera. The approximation is only "fair" because at low radiance values, noise in the CCD system may dominate, and for larger radiance values, the response of the sensor is nonlinear: It saturates at some point. And even between these limits, the sensor response isn't really linear.

What we do with these radiance images depends on our goals. If we want to build an environment map, then a radiance image is a fine thing to work with. If we want to display the image on a conventional monitor using a standard image-display program, we need to convert each radiance value to the sort of value recorded by an ordinary camera in response to this amount of radiance. As we discussed in Chapter 28, these values are typically *not* proportional to radiance. If the radiance values cover a very wide range, an ordinary camera might truncate the lowest and highest values. Because we have the raw values, we may be able to do something more sophisticated, tricking the visual system into perceiving a wider range of brightness than is actually displayed. This is the area of study called **tone mapping** [RPG99, RSSF02, FLW02, MM06], which is an active area of current research.

Rather than simply storing the average radiance for each location, we could instead accumulate the samples themselves for later processing, allowing us to simulate the responses of several different kinds of sensors, for instance, or more generally, using them as data for a density-estimation problem, the "density" in this case being the pixel values. Our simple approach of averaging samples amounts to convolution with a box filter, but other filtering approaches yield better results for different applications [MN88]. Not surprisingly, if we know what filter we'll be using, we can collect samples in a way that lets us best estimate the convolved value (i.e., we can do importance sampling based on the convolution filter). In general, sampling and reconstruction should be designed hand in hand whenever possible.

The notion of taking multiple samples to estimate the sensor response at a pixel was first extensively developed in Cook's paper on distribution ray tracing [CPC84]. We've applied it here in its minimal form—uniform sampling over a square representing the pixel—but for animation, for instance, we also

need to integrate over time. The camera shutter is open for some brief period (or its electronic sensor is reset to black and allowed to accumulate light energy for some period), and during that time, the arriving light at a pixel sensor may vary. We can simulate the sensor's response by integrating over time, that is, by picking rays through random image-plane points as before, but also with associated random time values in the interval for which the shutter is open. The time associated to a ray is used to determine the geometry of the world into which it is shot: A ray at one moment may hit some object, but a geometrically identical ray a moment later may miss the object, because it has moved.

Naturally, it's very inefficient to regenerate an entire model for each ray. Instead, it makes more sense to treat the model as four-dimensional, and work with four-dimensional bounding volume hierarchies. The sample rays we shoot are then somewhat axis-aligned (their t-coordinate is constant), allowing the possibility of some optimization in the BVH structure.

Taking multiple samples in space and time helps generate motion blur; other phenomena can also be generated by considering larger sampling domains. For instance, we can change from a pinhole camera to a lens camera by tracing rays from each pixel to many points of a lens, and then combining these samples. With a good lens-and-aperture model, we can simulate effects like focus, chromatic aberration, and lens flare. All that's required is lots and lots of samples and a strategy for combining them.

When we sample rays passing through the points of a pixel square with a uniform distribution, we get to estimate the pixel-sensor response with a Monte Carlo integration. We showed in Chapter 31 that the variance of the estimate falls off like $1/N$, where N is the number of samples, assuming that the samples are independent and identically distributed. One of the reasons for the inverse-linear falloff is that when we draw many samples independently, they will tend to fall into clusters, that is, it's increasingly likely that some pair of samples are quite close to each other, or even groups of three or four or more. It's natural to think that if we chose our samples so that no two were *too* close, we'd get "better coverage" and therefore a better estimate of the integral. This conjecture is correct.

A simple implementation, the most basic form of **stratified sampling,** divides the pixel square into a $k \times k$ grid of smaller squares, where $k \approx \sqrt{N}$, and then chooses one sample uniformly at random from each smaller square. With this strategy, the variance falls off like $1/N^2$, which is an enormous improvement.

Inline Exercise 32.9: Suppose that you have 25 samples to use at one pixel. You can
(a) distribute them in a 5×5 grid,
(b) distribute them uniformly and independently, or
(c) use the stratified sampling strategy just described, dividing the pixel square into small squares and choosing one sample per small square. We've said that choice (c) is better than choice (b), but even with choice (c), we can get pairs of samples (in adjacent small squares) that are very close to each other. Does this mean that choice (a) is better?

Regardless of what approach you take to generating samples, it's worth thinking about the result you'll get when the function you're integrating has a sharp edge, such as the light reflected by adjacent squares of a chessboard—one (white) square reflects well, the adjacent (black) square does not. If the edge between

adjacent squares crosses the pixel grid diagonally, and we use a single ray through each pixel center to estimate the reflected light, we get aliasing, as we saw in Chapter 18. Using distribution ray tracing (or, equivalently, using Monte Carlo integration) tends to replace this aliasing with noise, which is less visually distracting. So one way to choose a sampling strategy is to ask, "What kinds of noise do we prefer, if we have to have noise at all?"

Yellot [Yel83] suggests that the frequency spectrum of the generated samples can be used to predict the kinds of noise we'll see. If there's lots of energy at some frequency f, and the signal we're sampling also has energy at or near f, we'll tend to see lots of aliasing rather than noise. And if there's lots of low-frequency energy in the spectrum, the aliases produced will tend to be low-frequency, which are more noticeable than high-frequency ones. In graphics, a sampling pattern is said to be a **blue noise distribution** if it lacks low-frequency energy and lacks any energy spikes. (The term is generally used for something more specific, namely, one in which the spectral power in each octave increases by a fixed amount so that the power density is proportional to the frequency.) Yellot gives evidence that the pattern of retinal cells in the eye follows a blue-noise distribution. And the good antialiasing properties certainly suggest that such distributions are good candidates for sampling, as Cook noted. Mitchell [Mit87] notes that the stratified sampling Cook proposes has the blue-noise property, at least weakly, but that other processes can generate much better blue noise. For instance, the Poisson disk process (initialize a kept list to be empty; repeatedly pick a point uniformly at random; reject it if it's too near any other points you've kept, otherwise keep it) generates very nice blue noise. It's unfortunately somewhat slow. Mitchell presents a faster algorithm, and Fattal [Fat11] has developed a very fast alternative that represents the current state of the art.

In our rendering code, we've divided light into "diffusely scattered" and "impulse scattered," on the grounds that the spikes in the BSDF for a mirror or an air-glass interface have values that are *so* much larger than those nearby that they are qualitatively different. But this fails to address the important phenomenon of very glossy reflection (like the reflection from a well-waxed floor). The glossier your materials are, the more difficult efficient sampling becomes. When we want to compute scattered rays from a surface element, we can always sample outgoing directions ω_o with a uniform or cosine-weighted distribution, and then assign a weight to the sample that's proportional to the scattering value $f_s(\omega_i, \omega_o)$, but such samples will be ineffective for estimating the integral when $\omega_o \mapsto f_s(\omega_i, \omega_o)$ is highly spiked (assuming the incoming radiance is fairly uniform). At the very least, it's best if your BSDF model provides a sampling function that can generate samples in proportion to $\omega_o \mapsto f_s(\omega_i, \omega_o)$, although to accurately estimate the reflectance integral, you must also pay attention to the distribution of arriving radiance, which itself is dependent on the emitted radiance and the visibility function. The only algorithm we know that is designed to simultaneously consider all three—the variation in the BSDF, the emitted radiance, and the visibility—is Metropolis light transport, but it comes with its own challenges, such as start-up bias and the difficulty of designing effective mutations and correctly computing their probabilities.

To return to the matter of path-tracer/ray-tracer-style rendering, the goal to keep in mind is *variance reduction*: If you can accurately estimate some *part* of an integral by a direct technique, you may be able to substantially reduce the variance of the overall estimate. Of course, it's important to reduce variance while

keeping the mean estimate correct (or at least consistent, i.e., approaching the correct answer as the number of samples is increased). After the most obvious optimizations, however, this leads to diminishing returns. If you use your "domain knowledge" to say "most rendering happens in scenes where the lighting doesn't vary very fast," you'll soon find yourself needing to render a picture of the night sky, where essentially all lighting changes are discontinuities rather than gradual gradients.

One of the most promising recent developments in Monte Carlo rendering is to use the gathered samples in a different way. Rather than computing an average of samples, or a weighted average, we can treat the samples we've gathered as providing information about the *function* that they're sampling. Let's begin with a very simple example: Suppose we tell you we have a function on the interval $[0, 1]$ and that it's of the form $f(x) = ax + b$ for some values a and b (but you don't know a and b). It's easy to show that the average value of f on the unit interval is $(a/2) + b$. Now let's suppose we ask you to estimate that average using a Monte Carlo integral. You might take ten or 20 samples, average them together, and declare that to be an estimate of the average value of f; this is exactly analogous to what we've been doing in all our rendering so far. But suppose that you looked more carefully at your samples, and for each one, you know *both* x and $f(x)$. For instance, maybe the first sample is $(0.1, 7)$ and the second is $(0.3, 8)$. From these two samples alone, you can determine that $a = 5$ and $b = 6.5$, so the average value is 9. From just two samples, we've generated a *perfect* estimate of the average. Of course, we were only able to do so because we knew something about the x-to-$f(x)$ relationship.

This idea has been applied to rendering by Sen and Darabi [SD11]. They posit that the sample values for a particular pixel bear some functional relation to the random values used in generating the samples. For instance, the sample value might be a simple function of the displacement from the center of the pixel or of the time value of the sample in a motion-blur rendering in which we have to integrate over a small time interval. Because we use random numbers to select the ray (or the time), we get random variations in the resultant samples. Sen and Darabi estimate the relationship of the sample values to the random values used to generate the samples. Their estimate of the relationship is not as simple as the $ax + b$ example above; indeed, they estimate statistical properties of the relationship rather than any exact parameters. From this, they distinguish variation due to position (which they regard as the underlying thing we're trying to estimate) from the variation due to other injected randomness, and then use this to better guess the pixel values, in a process they call **random parametric filtering** (RPF). Figure 32.20 shows an example of the results.

In this chapter, we've developed two renderers, but they are by no means state of the art. The book by Pharr and Humphreys [PH10] (which is nearly as large as this book) discusses physically based rendering in great detail, and is a fine choice for those who want to study rendering more deeply. The SIGGRAPH proceedings, other issues of the ACM Transactions on Graphics, and the proceedings of the Eurographics Symposium on Rendering give the student the opportunity to see how the ideas in this chapter originally developed, and which avenues of research proved to be dead ends and which have stood the test of time.

Figure 32.20: The eight samples per pixel that were used to generate the Monte Carlo rendering on top are filtered with RPF to produce the improved rendering on the bottom, which is virtually indistinguishable from a Monte Carlo rendering generated with 8,192 samples per pixel. (Courtesy of Pradeep Sen and Soheil Darabi, ©2012 ACM, Inc. Reprinted by permission.)

32.10 Exercises

Exercise 32.1: Up through Listing 32.10, we addressed the effects of the refractive index in two places in our program: in approximating the Fresnel term, and in computing the change in radiance at an interface between surfaces of different refractive indices. We did not, however, use the coefficient of extinction at all: Every material we considered transmits light perfectly, which means that all absorption takes place only at the boundaries between materials. Modify the path tracer to account for the coefficient of extinction of a material.

Exercise 32.2: (a) Let $A = (0,0)$, $B = (1,0)$, and $C = (1,1)$, and let T be the triangle ABC. Suppose that the texture coordinates for A are $(u_A, v_A) = (0.2, 0.6)$, for B they're $(0.3, 0.3)$, and for C they're $(0.5, 0.1)$. Assume that texture coordinates are interpolated linearly across the triangle. Find the unit vector $\mathbf{u}$ such that the u-coordinate increases fastest in direction $\mathbf{u}$.
(b) Generalize to arbitrary point and texture coordinates at the three vertices.

Exercise 32.3: (a) We used photon mapping to estimate the diffuse scattering of indirect light. Jensen suggests not even storing photons except on diffuse surfaces. And we also omit direct lighting from the photon mapping calculation—that gets handled in a separate step. But what would happen if you used photon mapping for specular reflections and direct lighting as well? What would you expect? Would a final gather be of any use?
(b) If you've written a photon-mapping renderer, try modifying it to handle each of these individually. Were your predictions correct?

Exercise 32.4: Our path tracer and photon mapper both assume that all light sources are "on the outside"—we don't allow for a glowing lamp embedded in a glass sphere. Where in the code is the assumption embedded?

Exercise 32.5: (a) We merged the photon map with a ray tracer. Can you think of how to merge it with a path tracer instead?
(b) The photon mapper stops pushing around photons after some maximum depth; that introduces a bias. How? For a photon map that allows only n bounces, construct a scene where the resultant radiance estimate is drastically wrong, no matter how many photons you send into the scene.
(c) Can you take the idea from the path tracer—"just continue tracing until things stop"—and use it in the propagation of photons? Will it solve the problem with the scene you constructed in part (b)?

Exercise 32.6: (a) Given a random number generator that produces values uniformly in the interval $[0, 1]$, describe how to generate uniform random points in the unit square.
(b) If you generate the point (x, y) with $x < y$, you can replace it with (y, x), and otherwise leave it unchanged. Show that this generates points uniformly in the triangle with vertices $(0, 0), (1, 0), (1, 1)$.
(c) Let $u = 1 - x, v = 1 - y$, and $w = 1 - (u + v)$. Show that applying this transformation to the results of part (b) generates points uniformly at random in barycentric coordinates on the triangle $u + v + w = 1, 0 \leq u, v, w, \leq 1$.
(d) Show that for any triangle PQR, the points $uP + vQ + wR$ are distributed uniformly at random in the triangle, when uvw are generated according to part (c).

Exercise 32.7: (a) Write a WPF program that uses the method in Exercise 32.6 to generate points in a triangle. The user should be able to drag the three triangle vertices, and press buttons to generate either a single point or 100 points, each of which should be displayed as a colored dot within the triangle. Another button should clear the points.
(b) Extend your program to handle meshes. Generate a 2D mesh (perhaps the Delaunay triangulation for a random set of points), and then improve your program to pick a point (or 100 points) in the mesh uniformly at random. Do so by precomputing the triangle areas, summing them, and then assigning each triangle a probability given by its area divided by the total area. Put the triangles in some order, and compute probability sums $s[0] = p[0]$, $s[1] = p[0] + p[1]$, $s[2] = p[0] + p[1] + p[2]$, etc. Given a uniform random variable u, you can now identify the largest index i with $u \leq s[i]$. To generate a random mesh point, you can pick a uniform random number u, identify the last triangle i with $u \leq s[i]$, and then generate a random point in that triangle (see Exercise 32.6).
(c) Briefly discuss how to make the triangle-selection process faster than $O(n)$, where n is the number of triangles in the mesh.
(d) Suppose that in ordering the triangles, we place the largest ones first. Then in a search of the list, we're likely to examine relatively few triangles to find the "right" one. Would you, in working with a typical graphics model, expect this to have a large impact on the sampling time? Why or why not?

Exercise 32.8: We've argued that an eye ray hitting a point source is a probability-zero event, so we can ignore point sources. But since a point source is generally used to represent a limit of ever-tinier spherical sources, and for any such spherical source there's a nonzero probability of an eye ray hitting it, the "can ignore" argument depends on what happens in these approximating cases. The radiance emitted by such a small approximating sphere is proportional to the inverse square of its radius (to maintain constant power); the probability of an eye ray hitting it is proportional to its squared radius; hence, the expected contribution to the pixel (once it's small enough for its image to be completely contained in

the pixel square on the film plane) is constant. We're therefore treating the limit of a constant as zero. Adjust all the renderers in this chapter to make an extra pass to "correctly" render the effect of point lights by tracing a ray from each point light to the eye, and adding the appropriate radiance to the appropriate pixel if the point light's actually visible. In doing this, you're basically doing a special case of bidirectional path tracing.

Exercise 32.9: In the ray-trace-with-photon-mapping renderer, we estimate the direct light arriving from an area light with several samples. If the area light's far away, this is probably overkill; if it's nearby, it's probably inadequate. For Lambertian reflectance of a uniformly emitting planar, one-sided area light, only the cosine term varies with the light-sample position. Suppose (see Figure 32.21) that the area light is completely contained in a sphere of radius s about some point Q, we're reflecting it at a point P of some Lambertian surface, the "bright side" is completely visible from P, and the length of the vector $\mathbf{r} = Q - P$ from P to Q is d. Under what conditions on d and s can we approximate the reflected radiance due to the luminaire directly multiplying the projected area (orthogonal to $\mathbf{r}$) of the source, the cosine of the angle θ between $\mathbf{r}$ and $\mathbf{n}_P$ and the Lambertian reflectance, and $1/\|\mathbf{r}\|^2$ and be certain the result is correct within 1%? The several assumptions—that the source be planar, uniform, completely visible, etc., and that the reflecting surface be Lambertian—are not really as restrictive as they might sound. In particular, the same idea works for convex nonplanar sources, although the computation of the projected area may be nearly as complex as using multiple samples to estimate the reflected radiance.

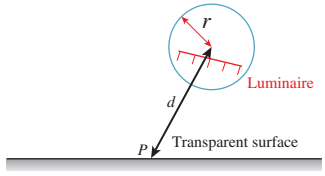


Figure 32.21: The rule of five, revisited.

◈ (b) Suppose that the reflecting surface has a BRDF (for a fixed outgoing direction) whose variation, as a function of the incoming light direction θ, is nicely bounded, that is, $|f(\theta) - f(\theta')| < K|\theta - \theta'|$. Can you do an analogous analysis? See also Exercise 26.12 in Chapter 26.

Exercise 32.10: When we sampled points on an area light, we sampled uniformly with respect to area. We can instead presample a light source, using stratified sampling to generate a collection of samples that we can reuse. The stratified sampling helps ensure that the estimate of the average effect of the point light is accurate, although the estimates for adjacent pixels are likely to be highly correlated, which may be a problem in some cases. Essentially we're replacing an area light with a collection of "micro-light" point lights. If there are too few, the shadows cast by each may generate a noticeable artifact.

(a) Build a ray tracer that replaces each area light with multiple point lights in this way. Do you see any artifacts? How is the running time affected as you increase the number of samples on the luminaire?

(b) Instead of using all the micro-lights to illuminate each surface point, we can pick one at random, essentially doing a single-sample estimate of the radiance transfer from the light to the surface point. Doing this once per primary ray can generate nice soft shadows. If we're shooting 25 primary rays (using stratified sampling), and the area light is represented with 25 micro-lights, we'd like to use each micro-light once. How should you pair up the primary rays and the micro-lights? Do you foresee any problems?

(c) Implement your approach and critique the results.

Exercise 32.11: (a) Use the Poisson disk process to generate blue-noise samples on a line: Generate samples in the interval $[0, 1]$ in which all are at least $r = .001$ apart, until there is no more room to fit a new sample.

(b) Discretize the interval into 10,000 bins, and record a 1 in each bin that contains a sample, and a 0 otherwise.

(c) Compute the fast Fourier transform of the resultant array. Does it appear to be a blue-noise distribution?
(d) Try this process again with various values for r. Below what frequency (in terms of r) is there relatively little energy?
(e) Now generate a similar occupancy array using stratified sampling, and compare the frequency spectra of the two processes. Describe any differences you find.
(f) Generalize to 2D.
(g) Implement Mitchell's [Mit87] point diffusion algorithm for generating blue noise, and compare its results to the others.

Exercise 32.12: For a rectangular area light, write code to sample a point from the light uniformly with respect to area. Do the same for a spherical source. For the sphere, recall that the projection $(x, y, z) \mapsto (x/r, y, z/r)$, where $r = \sqrt{x^2 + z^2}$ is an area-preserving map from the unit-radius cylinder about the y-axis, extending from $y = -1$ to $y = 1$, onto the unit sphere.

Chapter 33

Shaders

33.1 Introduction

This chapter is about **shaders,** pieces of code written in a **shading language,** a specialized language that's designed to make shader writing easy. Shaders describe how to process data in the graphics pipeline. Shading languages are evolving so fast (as is the programmability of the pipeline itself) that this chapter will be out of date before its last sentence is written, let alone before you read it. Despite this rapid evolution, there are some things that are invariant across several generations of shading languages, and that we anticipate will remain in future versions for at least a decade. There's some reason to believe this. The evolution of the graphics libraries that link a program on a CPU to one or more programs on the GPU has been from the *specific* (in early years) to the *general,* to the point where much of GL 4 resembles an *operating system* rather than a graphics library: It's concerned with linking together executable pieces of programs on the CPU and GPU, passing data between processes, starting and stopping threads of execution, etc. There's no obvious further generalization that can happen, at least in the near term. Perhaps in five years you'll write shaders directly in C# rather than in a specialized shading language, and those shaders will run on 500-core machines. But in many ways they'll continue to look the same: The first thing we usually do with vertex data is to transform it to world-space coordinates using multiplication by some matrix. That will look the same, no matter what the language.

We'll therefore describe a few shaders, using GL 4 as our reference system, and trusting that you, the reader, will be able to interpret the *ideas* of this chapter into whatever shading language you're using.

33.2 The Graphics Pipeline in Several Forms

Figures 33.1 and 33.2 show the various steps involved in either a rasterizing renderer or a ray tracer, as described in Chapter 15.

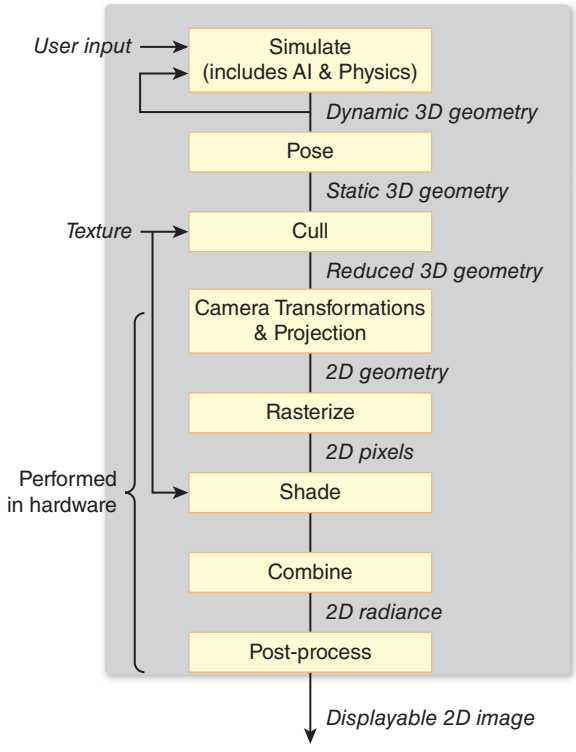


Figure 33.1: The steps involved in a basic rasterizing renderer.

There are many operations—transforming objects from object space to world space to camera space, shading, placing pixel values in a buffer for later use (either in environment mapping, for instance, or in compositing with some other precomputed imagery)—that occur in both pipelines.

As software engineers, we know that when there's commonality, there's an opportunity for abstraction and the development of an interface to the common portions of the code. The particular *form* of the interface can vary: Some designs employ virtual methods, others use callbacks, etc. In some cases the thing being abstracted is complex enough that the way in which it's used is itself complex. In these cases, it makes sense to create a language in which the use pattern is described via small programs. We've seen an example of this with WPF in earlier chapters: XAML provides a language for describing objects, their geometric properties, their relationships, and how data is passed among them, for instance. A C# program usually combines with XAML code to constitute an entire graphics project.

In the case of the graphics pipelines shown in the preceding figures, there's a common structure: The geometric and material descriptions of objects in a scene undergo similar transformations, for instance, in a similar order in both pipelines. But the details of what goes on at certain stages vary. **Shaders** are small programs that specify how the duties of certain portions of the pipeline are to be carried out.

The sidebar in this chapter describes informally how we got from individual renderers written in research laboratories to the software design of packages like GL 4.

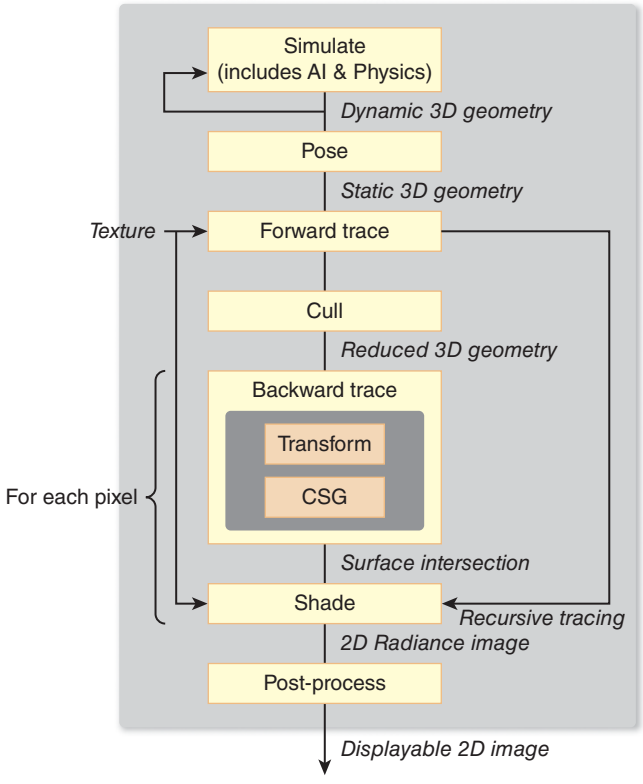


Figure 33.2: The steps involved in a basic ray tracer.

33.3 Historical Development

Immediate-mode packages like GL in its early forms provided ways to represent, in the sequence of instructions issued to the package (typically by function calls), something about the *structure* of the objects to be drawn. If an object was modeled with a hierarchy of transformations, then the sequence of GL calls would reflect this, pushing and popping matrix transformations from a stack that represented a current transformation to be applied to all subsequent vertices. These transformed vertices, together with vertex-index triples representing triangles, formed the core of what was to be rendered. The rendering process followed a fairly straightforward path, which can be coarsely summarized by saying that a collection of triangles with per-vertex and per-triangle attributes were described to the system, often with various transformations applied to the vertices. The resultant triangles were then transformed to the standard perspective view volume, and clipped against the near clipping plane. They were then transformed to the standard parallel view volume, and clipped against the remaining clipping planes. The resultant triangles were then rasterized, the rasterized pixels were shaded (i.e., some computation was done to determine their color, a computation that often involved texture lookup), and the triangles were placed into a Z-buffer, with only the frontmost remaining in the final image. Sometimes the resultant image was combined with some preexisting image via a compositing operation so that multiple objects

could be rendered in separate passes and then a single image could be produced at the end.

As graphics developed, the particular choices of transformations to be applied to triangles, or how values computed at vertices were to be interpolated across triangles, or even how high-level descriptions of objects were to be converted to triangle lists, all varied. But there were a few things that were shared by essentially all programs: vector math, clipping, rasterization, and some amount of per-pixel compositing and blending. The development of GPUs has reflected this—GPUs have become more and more like general-purpose processors, except that (a) vector and matrix operations are well supported, and (b) clipping and rasterization units remain a part of the design. The modern interface to the GPU now consists of one or more small programs that are applied to geometric data (these are called **vertex shaders**), followed by clipping and rasterization, and one or more small programs that are applied to the "fragments" produced by rasterization, which are called **pixel shaders** or **fragment shaders.** A more appropriate name for what's currently done—computing shading values for one or more samples associated to a pixel—might be **sample shaders.** The programmer writes these shaders in a separate language, and then tells the GPU in which order to use them, and how to link them together (i.e., how to pass data from one to the other). Typically some packages (like GL 4) provide facilities for describing the linking process, compiling and loading the shaders onto the GPU, and then passing data, in the form of triangle lists, texture maps, etc., to the GPU.

Why are these programs called shaders? In the GL version of the Lambertian lighting model, similar to the one presented in Chapter 6, the color of a point is computed (using GL notation) by

$$C = k_d C_d L(\boldsymbol{\ell} \cdot \mathbf{n}), \tag{33.1}$$

where $\boldsymbol{\ell}$ is the unit direction vector to the light source, k_d is a representation of the reflectance of the material, C_d is the color of the material (i.e., a red-green-blue triple saying how much light the surface reflects in each of these wavelength bands), L is the color of the light (again an RGB triple, which is multiplied term by term with C_d), and $\mathbf{n}$ is the surface normal. In Phong lighting, another term, involving the view vector as well, and k_s, a specular constant, C_s, a specular color, and n_s, the specular exponent, are added.[1] Increasingly complex combinations of data like this, including texture data to describe surface color or surface-normal direction, etc., got added, and the formulas for computing the color at a point got to look more and more like general programs. Cook [Coo84] introduced the idea that the user could write a small program as part of the modeling process, and the rendering program could compile this program into something that executed the proper operations. Cook called this programmable shading, although perhaps programmable lighting would be a better term for the process we've just described. In that era, the computation entailed by lighting models was often so great that it made sense to do much of the computation on a per-vertex basis, and then interpolate values across triangles; the interpolation process was called shading, and it varied from the interpolation of the colors to the interpolation of values to be used in computing colors. Since papers describing lighting models often also described

1. In the terminology we've used from Chapter 14 onward, these would be the "glossy" constant, color, and exponent.

such shading approaches, the two ideas became conflated, and the term "shader" was used for the new notion.

Modern shaders are really graphics programs rather than being restricted to computing colors of points. There are **geometry shaders,** which can alter the list of triangles to be processed in subsequent stages, and **tessellation shaders,** which take high-level descriptions of surfaces and produce triangle lists from them; an example is a subdivision surface shader, which might take as input the vertices and mesh structure of a subdivision surface's control mesh, and produce as output a collection of tiny triangles that form a good approximation of the limit surface. There are also **vertex shaders** that serve only to transform the vertex locations, and generally have nothing to do with eventual color.

While the typical graphics program might have a geometry shader, a tessellation shader, a vertex shader, and a fragment shader, there is also the ability to turn off any portion of the pipeline and say, "Just compute this far and then stop." Thus, a program might run its geometry and tessellation shader, and then return data to the CPU, which could modify it in some way before returning it to the GPU to be processed by the rasterization and clipping unit and then a fragment shader.

We'll describe some basic vertex and fragment shaders to give you a feel for how shaders are related to the ideas you've seen throughout this book.

What follows is a rough and informal description of the history of raster graphics, from a high level.

- At the start of graphics, no one had any idea how to do anything, so we found a way to create rasterized lines, for instance, and to draw surfaces with flat shading.
- The next year, we thought of a new way to rasterize, and thought about curves rather than just lines, and someone came up with a new lighting model.
- Pretty soon, we realized that there was a higher-level problem—rasterization of primitives—to be solved, and that lighting models would evolve every year, and that we needed an architecture in which that sort of thing was possible. On the other hand, there were parts of almost every graphics program—clipping, for instance—that would probably remain fairly constant, and appear in the same place in the program; this was the start of the "pipeline" idea.
- Making a general-purpose language for describing lighting was too expensive when *most* lighting was going to use the Phong model. So we split into two camps: fixed function and programmable. The programmable camp's rendering was slow, but very general-purpose. The fixed-function camp rendered things fast, but was constrained in what sorts of rendering it could do. The only reason for the split was the difference in how people wanted to control what went on in the computer: Some, who loved interactivity, said, "You can adjust the constants, and I'll burn the algorithm into silicon"; the others said, "Interactivity isn't so important to me . . . but I *really* want expressiveness. I can always get more computers, but I want a *programming* language to describe my output." The first gang went on to develop the fixed-function approach, and from an industry point of view, they were clustered around Silicon

Graphics, Inc., and a few other makers of graphics workstations. The second group formed the core of the special effects and computer-based animation industry, centered around firms like Lucasfilm and Pixar.

- The good news for the reader is that *both* approaches won: The success of the fixed-function approach led to the development of commodity graphics cards. In less than 20 years, the cost of graphics performance dropped by a factor of more than a thousand. Meanwhile, the programmable shading approach showed the world just how much could be done with graphics that wasn't constrained by interactivity considerations. Finally, Moore's Law meant that processor speeds were improving enormously, and one year's noninteractive program was next year's interactive program. The result has been that the movie industry now uses GPUs, that is, stock graphics hardware, while the game industry now routinely uses programmable shading.
- The convergence was gradual, however. As the environment changed, year by year, the tradeoffs between the two approaches could be evaluated in the context of current hardware, model size, etc.; the fixed-function approach gradually lost out to programmability, and all the ideas from Cook's programmable shaders paper gradually entered the hardware.
- As a final stage (so far!), it became clear that graphics now looked like "linking together bits of program in more and more complex pipelines," which sounds more like an operating systems problem than a graphics problem. The design of GL4 reflects this: It's a system for defining bits of program and linking them together into complete assemblies; the graphics-specific parts of the design are only a small part, and many of the graphics-specific ideas of early versions of GL are now deprecated.

33.4 A Simple Graphics Program with Shaders

As we said earlier, the job of a modern graphics system is rather like an operating system. Three separate entities need to communicate:

- A program running on the CPU (the host program)
- The graphics pipeline: some implementation of the processing of data from the host program including things like geometric transformations, clipping and rasterization, compositing, etc.
- The shader programs that run on the GPU

Part of the graphics pipeline may be implemented on the CPU as a library; some parts may be implemented on the GPU. Part of the function of the graphics system is to isolate the developer of the host program from these details (which may vary from computer to computer, and from graphics card to graphics card). Of course, the developer of the host program is typically also the person who develops the shader programs. That developer must ask the question, "How do I connect a variable in my C#/C++/Java/Python program with a corresponding variable in

the vertex shader?" for instance. That's the role of GL (or DirectX, or any other graphics API).

The details of the linking process that associates host-program variables with shader variables are messy and complicated, and the design of GL is extremely general. Almost every developer will want to work with a **shader wrapper**—a program that once and for all chooses a particular way to use GL to hook a host program to shaders, and provides features like automatic recompilation of shaders, etc. Graphics card manufacturers typically provide shader-wrapper programs to allow the easy development of programs that fit the most common paradigms. Only those who need the finest level of control (or those developing shader-wrapper programs) should actually work with most of the tools GL provides for linking host programs to shader programs.

We'll use such a shader wrapper—G3D—in writing our example shaders in this chapter. G3D is an open source graphics system developed by one of the authors [McG12], and provides a convenient interface to GL. But the shaders in this chapter can in fact be used with other shader wrappers as well, with essentially no changes.

Let's look at a first example: a shader that provides Gouraud shading, computed once per vertex, and linearly interpolated across triangles. The host program in this case loads a model in which each vertex has an associated normal vector, and provides a linear transformation from model coordinates to world coordinates, and a camera specification. Listing 33.1 shows the declaration of an `App` class

Listing 33.1: The class definition and initialization of a simple program that uses a shader.

```
1  class App : public GApp {
2  private:
3      GLight              light;
4      IFSModel::Ref       model;
5
6      /** Material properties and shader */
7      ShaderRef           myShader;
8      float               diffuse;
9      Color3              diffuseColor;
10
11     void configureShaderArgs();
12
13 public:
14     App();
15     virtual void onInit();
16     virtual void onGraphics(RenderDevice* rd,
17                             Array<SurfaceRef>& posed3D);
18 };
19
20 App::App() : diffuse(0.6f), diffuseColor(Color3::blue()),
21    light(GLight::directional(Vector3(2, 1, 1), Radiance3(0.8f), false)) {}
22
23
24 void App::onInit() {
25     myShader = Shader::fromFiles("gouraud.vrt", "gouraud.pix");
26     model = IFSModel::fromFile("icosa.ifs");
27
28     defaultCamera.setPosition(Point3(1.0f, 1.0f, 1.5f));
29     defaultCamera.lookAt(Vector3::zero());
30
31     ... further initializations ...
32 }
```

derived from a generic graphics application (`GApp`) class: The `App` contains a reference to an indexed-face-set model and a single directional light (specified by its direction and radiance), values for the diffuse color of the surface and the diffuse reflection coefficient, and a reference to a shader object.

When the `run()` method on a `GApp` is invoked, it first invokes `GInit`, and then repeatedly invokes `onGraphics()`, whose job is to describe what should be rendered.

As you can see, the initialization of the application instance is fairly straightforward: In lines 20 and 21, we assign a diffuse reflectance and color to be used for a surface, and create a representation of a directional light (a direction and radiance value).

During initialization, G3D's `Shader` class is used to read the vertex and pixel shaders (as text) from their text files (line 25), and we load a model of an icosahedron from a file (line 26), and set the camera's position and view (lines 28 and 29).

At each frame, the `onGraphics` method is called (see Listing 33.2). The `setProjectionAndCameraMatrix` method (line 2) invokes several GL operations to establish values for predefined variables like `gl_ModelViewProjectionMatrix`. The next two lines clear the image on the GPU to a constant color.

Listing 33.2: The graphics-drawing procedure and `main`.

```
1  void App::onGraphics(RenderDevice* rd, Array<SurfaceRef>& posed3D){
2      rd->setProjectionAndCameraMatrix(defaultCamera);
3      rd->setColorClearValue(Color3(0.1f, 0.2f, 0.4f));
4      rd->clear(true, true, true);
5      rd->pushState(); {
6          Surface::Ref surface = model->pose(G3D::CoordinateFrame());
7
8          // Enable the shader
9          configureShaderArgs(light);
10         rd->setShader(myShader);
11
12         // Send model geometry to the graphics card
13         rd->setObjectToWorldMatrix(surface->coordinateFrame());
14         surface->sendGeometry(rd);
15     } rd->popState();
16 }
17
18 void App::configureShaderArgs() {
19     myShader->args.set("wsLight",light.position.xyz().direction());
20     myShader->args.set("lightColor", light.color);
21     myShader->args.set("wsEyePosition",
22         defaultCamera.coordinateFrame().translation);
23
24     myShader->args.set("diffuseColor", diffuseColor);
25     myShader->args.set("diffuse", diffuse);
26 }
27
28 G3D_START_AT_MAIN();
29
30 int main(int argc, char** argv) {
31     return App().run();
32 }
```

Between the `pushState` and `popState` calls, the program says which shader should be used for rendering this part of the scene (line 10) and which variable values are to be passed to the shader (line 9), establishes the model-to-world transformation for this model (line 13), and then (line 14) sends the geometry of the model to the graphics pipeline.

In the case of this simple shader, we send (lines 19–25) the world-space coordinates of the light, the color of the light, the position of the eye, the diffuse color we're using for our icosahedron, and the diffuse reflectivity constant. The `args.set` procedure establishes the link between the host program's value for, say, the directional light's world-space direction vector and the shader program's value for the variable called `wsLight`.

Finally, following the call to `onGraphics`, the shader wrapper tells the pipeline to process the vertices of the mesh one at a time with the vertex shader, assemble these into triangles which are then rasterized and clipped, and then process the rasterized fragments with the fragment shader. Let's look at the GLSL vertex shader code (Listing 33.3) to see what it does.

Listing 33.3: The vertex shader for the Gouraud shading program.

```
1  /** How well-lit is this vertex? */
2  varying float gouraudFactor;
3
4  /** Unit world space direction to the (infinite, directional) light source */
5  uniform vec3 wsLight;
6
7  void main(void) {
8      vec3 wsNormal;
9      wsNormal = normalize(g3d_ObjectToWorldNormalMatrix * gl_Normal);
10     gouraudFactor = dot(wsNormal, wsLight);
11     gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
12 }
```

As you can infer from the code, certain variables are predefined in GLSL, as are some useful functions, like `normalize` and `dot`. Table 33.1 lists a few of these.

Built-in data types include the C-like `float` and `int`, and others that help us perform vector operations like `vec3` and `mat33`. GLSL also provides tools for accessing any portion of a `vec3` or `vec4` through a construction called **slicing:** If `v` is a `vec3`, we can use `v.x` to access its first entry, and `v.yz` to access its second and third entries, for instance. Since a `vec3` is also used to represent colors (and a `vec4` is used to store colors with alpha), we can also write `myColor.rga` to access the red, green, and alpha portions of a color, for instance. Mixing *xyzw*-slicing and *rgba*-slicing is allowed, but seldom makes sense.

As described in Chapter 16, each kind of shader is responsible for establishing values used by later shaders. For instance, a vertex shader like the one used here gets `gl_Vertex`, the 3D world-space location of the vertex, as input; each time the shader is called, `gl_Vertex` has a new value, the world-space coordinates for another vertex, in the input. The vertex shader is responsible for assigning a value to `gl_Position`, which is meant to represent the position of the vertex in camera coordinates, or, expressed alternatively, the position of the vertex after the camera transformation has been applied to move the view frustum to the standard perspective view volume. In the case of our shader, this is accomplished by line 11, which multiplies the vertex coordinates by the appropriate matrix.

Table 33.1: Predefined items in GLSL.

Name	Type	Meaning
`gl_Vertex`	`vec4`	The homogeneous position of the current vertex
`gl_Normal`	`vec4`	The normal at the current vertex
`gl_FragColor`	`vec4`	The RGBA color of the current fragment
`gl_ModelViewProjectionMatrix`	`mat44`	The transformation from modeling coordinates to normalized device coordinates
`gl_Position`	`vec4`	The homogeneous normalized device coordinates of the current fragment, before perspective divide, i.e., `gl_Position.w` may not be `1.0`.[2]
`pow(x, y)`		Raises `x` to the `y` power; if `x` is a vector, do so termwise
`max(x, y)`		Returns the larger of `x` and `y`
`dot(x, y)`		Dot product of vectors; `x` and `y` must be of the same size (i.e., `vec2` or `vec3` or `vec4`)

A vertex shader may also get *other* input data, in one of two forms. First, there may be other per-vertex information, like the normal vector, or texture coordinates. Second, there may be information that's specified *per object*. In our case, the diffuse reflectivity is one such item (we don't use it in the vertex shader), and the world-space position of the light is another. That world-space light position is declared `uniform vec3 wsLight;` the keyword `uniform` tells GL that the value is set once per object. The declaration of the variable before `main` indicates that it needs to be linked to the rest of the program. In this case, it's linked to the host program by a call in `ConfigureShaderArgs`.

Finally, a vertex shader may *set* values to be used by other shaders. These values are computed once per vertex; during the rasterization and clipping phase, they're interpolated to get values at each fragment. The default is perspective interpolation (i.e., barycentric interpolation in camera coordinates), but image-space barycentric interpolation is also an option. The resultant values vary from point to point on the triangle, and so they are declared `varying`.

Our shader computes one of these, `gouraudFactor` (line 2), which is the dot product of the unit surface normal and the incoming light direction to the world-space light source.

Having computed this dot product (line 10) and the location of each vertex, we move on to the fragment shader (see Listing 33.4).

2. In OpenGL these are called "Clip coordinates," while normalized device coordinates are those after the perspective divide.

Listing 33.4: The fragment shader for the Gouraud shading program.

```
/** Diffuse/ambient surface color */
uniform vec3 diffuseColor;

/** Intensity of the diffuse term. */
uniform float diffuse;

/** Color of the light source */
uniform vec3 lightColor;

/** dot product of surf normal with light */
varying float gouraudFactor;

void main() {
    gl_FragColor.rgb = diffuse * diffuseColor *
                       (max(gouraudFactor, 0.0) * lightColor);
}
```

Once again, three uniform variables, whose values were established in the host program, get used in the fragment shader: the diffuse reflectivity, the color of the surface, and the color of the light.

We also, in the fragment shader, have access to the `gouraudFactor` that was computed in the vertex shader. At any fragment, the value for this variable is the result of interpolating the values at the three vertices of the triangle. In the shader, we do a very simple operation: We multiply the diffuse reflectivity by the diffuse color to get a `vec3`, and multiply the `gouraudFactor` (if it's positive) by the light color, giving another `vec3`. We then take the term-by-term product of these two (using the `*` operator) and assign it to the `gl_FragColor.rgb`. If the light is pure red and the surface is pure blue, then the product will be all zeroes. But in general, we are taking the product of the amount of red light and how well the surface reflects red light (and how well it reflects in this direction at all), and similarly for green and blue, to get a color for the fragment.

Every fragment shader is responsible for setting the value of `gl_FragColor`, which is used by the remainder of the graphics pipeline.

That's it! This simple host program and two simple shaders implement Gouraud shading. The results are shown in Figure 33.3. In the version of the program available on the book's website, we've added one GUI that allows you to pick a diffuse color and set the reflectivity interactively, and another that allows you to rotate the icosahedron to any position you like, but the essential ideas are unchanged.

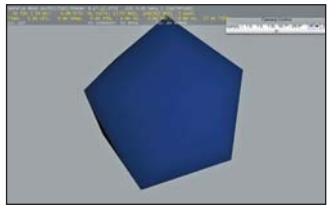


Figure 33.3: An icosahedron rendered by our first shader.

33.5 A Phong Shader

In generalizing this to implement the Phong model, there are no real surprises. We have to declare a few more variables in the host program, such as the specular exponent `shine`, the specular reflection coefficient `specular`, and the specular color `specularColor`, and we also include ambient light as `ambientLightColor`, but we're confident that you can do this without seeing the code.

Recall that the basic Phong model of Chapter 6 tells us to compute the pixel color using

$$color = k_d O_d I_a + k_d O_d I_d(\mathbf{n} \cdot \boldsymbol{\ell}) + k_s O_s(\mathbf{r} \cdot \mathbf{n})^{n_s} I_d, \tag{33.2}$$

where k_d and k_s are the diffuse and specular reflection coefficients, I_a is the ambient light color, I_d is the diffuse light color (i.e., the color of our directional light), O_d is the diffuse color of the object, ℓ is a unit vector in the direction from the surface point toward the light source, $\mathbf{r}$ is the (unit) reflection of the eye vector (the vector from the surface to the eye) through the surface normal, and n_s is the specular exponent, or shininess: A small value gives a spread-out highlight; a large value like $n_s = 500$ gives a very concentrated highlight. The formula is only valid if $\mathbf{r} \cdot \mathbf{n} > 0$; if it's negative, the last term gets eliminated.

In the vertex shader (see Listing 33.5), we compute the normal vector at each vertex, and the ray from the surface point to the eye (at each vertex). We don't normalize either one yet.

Listing 33.5: The vertex shader for the Phong shading program.

```
1  /** Camera origin in world space */
2  uniform vec3 wsEyePosition;
3
4  /** Non-unit vector to the eye from the vertex */
5  varying vec3 wsInterpolatedEye;
6
7  /** Surface normal in world space */
8  varying vec3 wsInterpolatedNormal;
9
10 void main(void) {
11     wsInterpolatedNormal = g3d_ObjectToWorldNormalMatrix *
12                            gl_Normal;
13     wsInterpolatedEye    = wsEyePosition -
14                      g3d_ObjectToWorldMatrix * gl_Vertex).xyz;
15
16     gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
17 }
```

In the pixel shader, we take the interpolated values of the normal and eye-ray vectors and use them to evaluate the Phong lighting equation. Even if the normal vector at each vertex is a unit vector, the result of interpolating these will generally not be a unit vector. That's why we didn't bother normalizing them in the vertex shader: We'll need to do a normalization at each pixel anyhow. After normalizing these, we compute the reflected eye vector $\mathbf{r}$, and use it in the Phong equation to evaluate the pixel color. Note the use of `max` (line 32) to eliminate the case where the reflected eye vector is not in the same half-space as the ray to the light source. (See Listing 33.6.)

Listing 33.6: The fragment shader for the Phong shading program.

```
1  /** Diffuse/ambient surface color */
2  uniform vec3 diffuseColor;
3  /** Specular surface color, for glossy and mirror refl'n. */
4  uniform vec3 specularColor;
5  /** Intensity of the diffuse term. */
6  uniform float diffuse;
7  /** Intensity of the specular term. */
8  uniform float specular;
9  /** Phong exponent; 100 = sharp highlight, 1 = broad highlight */
10 uniform float shine;
11 /** Unit world space dir'n to (infinite, directional) light */
12 uniform vec3 wsLight;
```

```
/** Color of the light source */
uniform vec3 lightColor;
/** Color of ambient light */
uniform vec3 ambientLightColor;
varying vec3 wsInterpolatedNormal;
varying vec3 wsInterpolatedEye;

void main() {
    // Unit normal in world space
    vec3 wsNormal = normalize(wsInterpolatedNormal);

    // Unit vector from the pixel to the eye in world space
    vec3 wsEye = normalize(wsInterpolatedEye);

    // Unit vector giving the dir'n of perfect reflection into eye
    vec3 wsReflect = 2.0 * dot(wsEye, wsNormal) * wsNormal - wsEye;

    gl_FragColor.rgb = diffuse * diffuseColor *
        (ambientLightColor +
           (max(dot(wsNormal, wsLight), 0.0) * lightColor)) +
        specular * specularColor *
        pow(max(dot(wsReflect, wsLight), 0.0), shine) * lightColor;
}
```

33.6 Environment Mapping

To implement environment mapping (see Section 20.2.1), we can use the same vertex shader as before to compute the interpolated eye vector and normal vector. Rather than computing the diffuse or specular lighting, we can use the reflected vector to index into an environment map, which is specified by a set of six texture maps (see Figure 33.4). The host program must load these six maps and make them available to the shader; to do so, we declare a new member variable in the `App` class and then, during initialization of the application, invoke

```
environmentMap = Texture::fromFile("uffizi*,png", ...)
```

to load the cube map with one of G3D's built-in procedures. Within the `configureShaderArgs` procedure, we must add

```
myShader->args.set("environmentMap", environmentMap);
```

to link the host-program variable to a shader variable.

The fragment shader (see Listing 33.7) is very simple: A fragment is colored by using its normal vector as an index into the cube map, via a GLSL built-in. The color that's returned is multiplied by the specular color for the model (which we set to a very pale gold) so that the reflections take on the color of the surface, simulating a metallic surface, rather than retaining their own color, as would occur with a plastic surface.

The result (see Figure 33.5) shows a shiny teapot reflecting the plaza of the Uffizi gallery using the environment map we showed earlier.

Anytime a shader uses a texture, the texture is automatically MIP-mapped for you by GL (unless you explicitly request that it not be). The semantics of GL are such that the derivative of *any* quantity with respect to pixel coordinates can be

Figure 33.4: An environment map of the Uffizi, specified by six texture maps, one each for the top, bottom, and four vertical sides of a cube, displayed here in a cross-layout. (Courtesy of Paul Debevec. Photographs used with permission. ©2012 University of Southern California, Institute for Creative Technologies.)

computed at any point where the quantity is defined. So at each point where the teapot appears in the image, the rates of change of the coordinates of the normal vector with respect to the pixel coordinates are computed and used to select a MIP-mapping level that's appropriate.

Figure 33.5: A shiny teapot reflects its surroundings, the plaza of the Uffizi gallery.

Listing 33.7: The fragment shader for the Phong shading program.

```
/** Unit world space direction to the (infinite, directional)
    light source */
uniform vec3 wsLight;

/** Environment cube map used for reflections */
uniform samplerCube environmentMap;

/** Color for specular reflections */
uniform vec3 specularColor;

varying vec3 wsInterpolatedNormal;
varying vec3 wsInterpolatedEye;

void main() {
    // Unit normal in world space
    vec3 wsNormal = normalize(wsInterpolatedNormal);

    // Unit vector from the pixel to the eye in world space
    vec3 wsEye = normalize(wsInterpolatedEye);

    // Unit vector giving direction of reflection into the eye
    vec3 wsReflect = 2.0 * dot(wsEye, wsNormal)
                         * wsNormal - wsEye;

    gl_FragColor.rgb =
        specularColor * textureCube(environmentMap,
                                    wsReflect).rgb;
}
```

33.7 Two Versions of Toon Shading

We now turn to a rather different style, the toon shading of Chapter 34. In toon shading, we compute the dot product of the normal and the light direction (as we would for any Lambertian surface), but then choose a color value by *thresholding* the result so that the resultant picture is drawn with just two or three colors, much as a cartoon might be. There are, of course, many possible variations: We could do thresholded shading using the Phong model, or any other; we could use two or five thresholds; we could have varying light intensity rather than the simple "single bright light" model we're using here.

The first (and not very wise) approach we'll take is to compute the intensity (the dot product of the normal and light vectors) at each vertex in the vertex shader (Listing 33.8), and let GL interpolate this value across each triangle and then threshold the resulting intensities (Listing 33.9).

Listing 33.8: The vertex shader for the first toon-shading program.

```
/* Camera origin in world space */
uniform vec3 wsEyePosition;
/* Non-unit vector to eye from vertex */
varying vec3 wsInterpolatedEye;
```

```
/* Non-unit surface normal in world space */
varying vec3 wsInterpolatedNormal;
/* Unit world space dir'n to directional light source */
uniform vec3 wsLight;
/* the "intensity" that we'll threshold */
varying float intensity;

void main(void) {
    wsInterpolatedNormal =
       normalize(g3d_ObjectToWorldNormalMatrix * gl_Normal);
    wsInterpolatedEye =
       wsEyePosition - (g3d_ObjectToWorldMatrix * gl_Vertex).xyz;

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    intensity = dot(wsInterpolatedNormal, wsLight);
}
```

Listing 33.9: The pixel shader for the first toon-shading program.

```
 ... same declarations ...
void main() {
    if (intensity > 0.95)
        gl_FragColor.rgb = diffuseColor;
    else if (intensity > 0.5)
        gl_FragColor.rgb = diffuseColor * 0.6;
    else if (intensity > 0.25)
        gl_FragColor.rgb = diffuseColor * 0.4;
    else
        gl_FragColor.rgb = diffuseColor * 0.2;
}
```

Figure 33.6: Toon shading using a vertex shader; notice the sharp corners of the highlight area.

The results, shown in Figure 33.6, are unsatisfactory: When the intensity is linearly interpolated across a triangle and then thresholded, the result is a straight-line boundary between the two color regions. When this is done for every polygon, the result is that each color region has a visibly polygonal boundary.

We can improve this substantially by using the *interpolated* surface normal and *interpolated* light vector in the fragment shader to compute an intensity value that varies smoothly across the polygon, and which, when thresholded, produces a smooth boundary between color regions. In our case, with a directional light, only the interpolation of the surface normal has an effect, but the program would also work for more general lights.

Inline Exercise 33.1: This program gives yet another instance of the principle that not every pair of operations commutes, and swapping the order for simplicity or efficiency only works acceptably in some cases. Explain which two operations are not commuting in this example.

Figure 33.7: Toon shading using a fragment shader. Notice the smooth boundary between the shades of red.

The revised program can use exactly the same vertex shader, except that we no longer need to declare or compute `intensity`. The revised fragment shader is shown in Listing 33.10, and the results are shown in Figure 33.7.

Notice that in the fragment shader, we took the normal vector that was computed at each vertex, and then interpolated to the current fragment, and *normalized* it.

Listing 33.10: The fragment shader for the improved toon-shading program.

```
uniform vec3 diffuseColor; /* Surface color */
uniform vec3 wsLight;      /* Unit world sp. dir'n to light */
varying vec3 wsInterpolatedNormal; /* Surface normal. */

void main() {
    float intensity = dot(normalize(wsInterpolatedNormal),wsLight);
    if (intensity > 0.95)
        gl_FragColor.rgb = diffuseColor;
    else if (intensity > 0.4)
        gl_FragColor.rgb = diffuseColor * 0.6;
    else
        gl_FragColor.rgb = diffuseColor * 0.2;
}
```

Inline Exercise 33.2: Suppose we had omitted the normalization in the fragment shader in Listing 33.10. How would the resultant image have differed? How would it have differed from our first-draft toon shader? If you don't know, implement both and compare.

33.8 Basic XToon Shading

Finally, we provide an implementation of a tiny portion of XToon shading: a shader where a 2D texture map (see Figure 33.8) is used to govern appearance, but in a somewhat unusual way. We index into the vertical coordinate using distance from the eye so that more-distant points are bluer, resulting in a weak approximation of **atmospheric perspective,** which is based on the observation that in outdoor scenes, more-distant objects (e.g., mountains) tend to look bluer, and hence we can provide a distance cue by mimicking this. We index into the horizontal coordinate using the dot product of the view vector with the normal vector. Whenever this dot product is zero (i.e., on a contour), we index into mid texture (i.e., the black area), resulting in black contour lines being drawn. In our texture, we've made the black line larger at the bottom than the top, resulting in wider contours at distant points than at nearby ones; endless other stylistic variations are possible.

The shader code is once again very simple. In the vertex shader, we compute the distance to the eye at each vertex; see Listing 33.11.

The results are shown in Figure 33.9. The teapot's contours are drawn with gray-to-black lines, thicker in the distance; the handle of the teapot is slightly bluer than the spout.

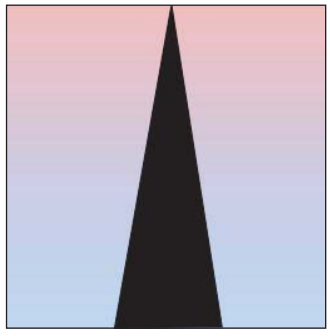


Figure 33.8: The 2D texture map for XToon shading. We use distance as an index into the vertical direction, and $\mathbf{v}\cdot\mathbf{n}$ *as the horizontal index.*

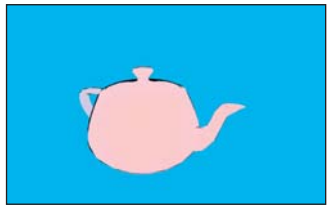


Figure 33.9: XToon-shaded teapot, using the texture map from the previous figure.

Listing 33.11: The vertex and fragment shaders for the XToon shading program.

```
... Vertex Shader ...
uniform vec3 wsEyePosition;
varying vec3 wsInterpolatedEye;
varying vec3 wsInterpolatedNormal;

varying float dist;

void main(void) {
```

```
    wsInterpolatedNormal =
        normalize(g3d_ObjectToWorldNormalMatrix * gl_Normal);
    wsInterpolatedEye = wsEyePosition -
        (g3d_ObjectToWorldMatrix * gl_Vertex).xyz;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    dist = sqrt(dot(wsInterpolatedEye, wsInterpolatedEye));
}

... Fragment Shader ...
varying vec3 wsInterpolatedNormal;
varying vec3 wsInterpolatedEye;
varying float dist;

void main() {
    vec3 wsNormal = normalize(wsInterpolatedNormal);
    vec3 wsEye = normalize(wsInterpolatedEye);
    vec2 selector; // index into texture map
    selector.x = (1.0 + dot(wsNormal, wsEye))/2.0; // in [0 1]
    selector.y = dist/2; // scaled to account for size of teapot
    gl_FragColor.rgb = texture2D(xtoonMap, selector).rgb;
}
```

33.9 Discussion and Further Reading

As we discussed in this chapter and what we should include as examples, one of us said, "I worry that what you guys call shaders are what I call graphics!" His point was a good one: Phong shading involves the same computation whether you do it on the CPU or on the GPU. The choice of where to implement a particular aspect of your graphics program is a matter of engineering: What works best for your particular situation? Since GPUs are becoming increasingly parallelized, and branching tends to damage throughput, a rough guideline is that branch-intensive code should run on the CPU and straight-line code on the GPU. But with tricks like hiding an `if` statement by adding arithmetic, as in using

```
x = (u == 1) * y + (u != 1) * z
```

as a replacement for

```
if (u == 1) then x = y else x = z
```

you can see that there's no hard-and-fast rule. The factors that may weigh in the decision are software development costs, bandwidth to/from the GPU, and the amount of data that must be passed between various shaders on the GPU.

New books of shader tricks are being published all the time. Many of the tricks described in these books are ways to get around the limitations of current GPU hardware or software architecture, and they tend to be out of date almost as soon as the books are published. Others have longer-term value, demonstrating how the work in some algorithm is best partitioned among various shader stages.

33.10 Exercises

For all these exercises, you'll need a GL wrapper like G3D, or else a shader development tool like RenderMonkey [AMD12], to experiment with.

Exercise 33.1: Write a vertex shader that alters the x-coordinate of every point on an object as a sinusoidal function of its y-coordinate.

Exercise 33.2: Write a vertex shader that alters the x-coordinate of each surface point as a sinusoidal function of y and *time* (which you'll need to pass to the shader from the host program, which will get the time from the system clock). You'll write something like

```
gl_Position.x += sin(k1 * gl_Position.y - k2 * t);
```

which will produce waves of wavelength $\frac{2\pi}{k_1}$, moving with velocity $\frac{2\pi}{k_2}$.

Exercise 33.3: Write a vertex shader that draws each triangle in a different (flat) color, specified by the host program. This can be very useful for debugging.

Chapter 34

Expressive Rendering

34.1 Introduction

In the early days of computer graphics, researchers sought to make any picture that resembled our eyes' view of a real object; photographs were considered completely realistic, and hence the goal of photorealism emerged. With further thought, one realizes that each photograph is just one possible condensation of the light field arriving at the camera lens; different lens and shutter and exposure settings, film or sensor types, etc., all change the captured image. Nonetheless, the term "photorealism" survives. When researchers began to think about other forms of imagery, they used the term "nonphotorealistic rendering" or NPR to describe it. Stanislaw Ulam said that talking about nonlinear science is like talking about nonelephant animals; we correspondingly prefer the term **expressive rendering,** which captures the notion of *intent* in creating such a rendering: The picture is meant to communicate something more than the raw facts of the incoming light.

Most traditional "rendering" (as in "an artist's rendering of a scene") has not aimed for strict photorealism. Given the wealth of experience gathered by artists and illustrators about effective ways to portray things, we can learn much by examining their work. In doing so, we must consider the artists' *intent:* while some have aimed for photorealism, others have tried to convey an impression that some scene made upon them, while still others have aimed for condensed communication (think of illustrators of auto-repair manuals) or highly abstract representations (see Figure 34.1). The intent of the work influences the choices made: The stylistic choices made by Toulouse-Lautrec, conveying the mood of a Parisian nightclub, are very different from those made by Leonardo depicting the musculature of the human arm.

We characterize expressive rendering as work that is concerned with *style, intent, message,* and *abstraction,* none of which can be easily defined precisely. Scene *modeling* can also create a style or support an artistic intent (think of set design in theatre or scene design in films), and careful composition can convey intent or exhibit abstraction even in photographs. So there's no clear line of demarcation between "expressive rendering" and "photorealism." Nonetheless, there are things that seem to fall naturally into one or the other category, and this

Figure 34.1: Various styles of art and illustration (top to bottom): photorealism (Harmen Steenwyck, Still Life with Fruit and Dead Fowl, 1630), impressionism (Monet, Impression, Sunrise, 1872), and technical illustration.

chapter discusses several techniques that fit the "expressive" mold. Broadly, work that focuses on abstraction and intent falls in the category of "illustration," while "fine art" may include work that emphasizes style, message, or media as well. Thus, much work in scientific visualization involves abstraction and intent; an illustration tries to convey the flow of blood, not the color of the cells nor the flow of any one particular cell. Herman and Duke [HD01] make a strong case for such use of expressive rendering in visualization applications.

There is naturally a certain overlap between photorealistic solutions to the rendering equation and artistic technique. Illustration books, for instance, teach students about various kinds of shadows (see Figure 34.2), each of which corresponds to some part of the solution to the rendering equation. **Direct shadows,** for instance, correspond to the first visibility term in the series expansion solution of the rendering equation, while **curvature shadows** correspond to the first bidirectional reflectance distribution function (BRDF) term (and subsequent ones, to a

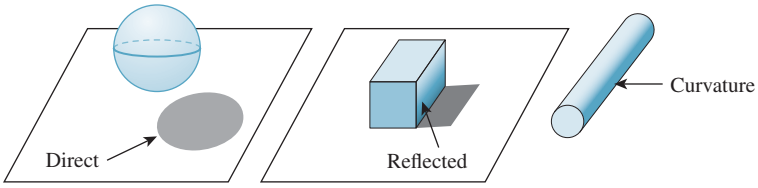


Figure 34.2: Three kinds of shadows (left to right): direct shadow, reflected shadow, and curvature shadow.

lesser degree). But aside from this, a *choice* is made in everything a human does in creating a picture. The choices all influence what the picture communicates to the viewer. Sometimes the choices are about **style**—the characteristics of the work that make it personal, or the work of *that particular person,* or something that conveys a certain tone or mood in a work—but mostly they're about **abstraction,** a mechanism for representing the essence of an object with no unnecessary detail.

When pictures fall into this second category, they indirectly tell us something about what matters to our visual systems: Just as when we are telling a story we try to give pertinent details and leave out the irrelevant, in making a picture there's good reason to omit the things that have less impact, or might have large impact but are not what's important. Thus, various simplified picture-making techniques reveal to us something about perception: People often communicate shape by drawing outlines or contours, suggesting that these are important cues about shape. They sometimes draw stick figures, suggesting that poses may be well communicated by relatively simple information about bone positions. To indicate relative positions (is he standing on the ground, or in mid-jump above it?), they sometimes use shadows, although the precise shape of the shadow seems less important than its presence, as we saw in Chapter 5.

Perceptual relevance is only one influence in expressive rendering. The most important is abstraction, the removal of irrelevant information and the consequent emphasis of what is important (to the creator of the image). There are three kinds of abstraction to consider in expressive rendering [BTT07].

- Simplification: The removal of redundant detail, such as drawing only a few bricks in a brick wall, or the largest wrinkles in a wrinkled shirt that's far from the viewer.
- Factorization: Separating the generic from the specific. In drawing a picture of a short-tailed Manx cat, you can either draw a *particular* cat or you can draw a generic cat—one that's recognizable as a Manx, but not as a particular one. In this case, you have factored out the identity of the cat from its type.
- Schematization: Representing something with a carefully chosen substitute that may bear little relation to the original, as in the schematic representation of a transistor in an electrical circuit (Figure 34.3) or a stick-figure drawing of a human.

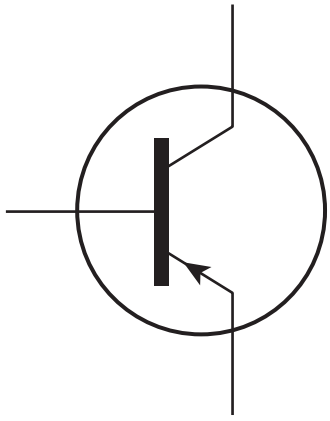


Figure 34.3: The schematic representation of a transistor encodes function and the fact that there are three conductors, but little else.

As Scott McCloud [McC94] observes, "By stripping down an image to its essential 'meaning,' an artist can *amplify* that meaning in a way that realistic art *can't*. . . . The more cartoony a face is, for instance, the more people it could be said to *describe*."

Research in expressive rendering is relatively new. Many early papers concentrated on emulating traditional media—pen-and-ink, watercolor, stained glass, mosaic tiles, etc. Some of the pictures produced were rather surprising, when

considered as computer graphics, but disappointing when considered as art: They managed to capture only the surface veneer of the art. Nonetheless, there was considerable value in this work, not only as a foundation for later work where intent and abstraction were incorporated, but in immediate applications as well. For instance, rendering with a rough pencil-sketch appearance conveys implicitly the idea that the rendering (or the thing being rendered) is incomplete, or that details are not important. A user-interface mockup drawn in a pencil-sketch style for initial testing can help get users to say, "I really need a brightness knob," rather than "I don't like the glossy highlights on the knobs," for instance.

34.1.1 Examples of Expressive Rendering

Before discussing style and abstraction further, we'll examine some early examples of expressive rendering (see Figure 34.4). These use various kinds of input, from imagery to purely geometric models to human-annotated models.

The first example comes from the work of Saito and Takahashi [ST90], who recognized that in the course of rendering an image, one could also record at each pixel a depth value for the object visible at the pixel, or the texture coordinates on the object at that point, or any other property. Using image-processing methods to detect discontinuities in depth allowed them to extract contours of shapes; similarly, searching for derivative discontinuities allowed them to detect edges (like the edges of a cube). By rendering these contour and edge pixels in highlight colors over the original, they created renderings that they characterized as "comprehensible," indicating their belief that the additional lines helped the visual system to better understand the thing being seen. The second example—a pen-and-ink rendering made from a more complex model—shows the application of **indication,** a technique in which something recurrent (like the pattern of shingles on a roof, or bricks in a wall) is suggested to the eye by just drawing a small portion of it. The third shows both the visible and the hidden contours of a polyhedral model; these have been extracted at real-time rates and assembled into long arcs, and then these arcs have been rendered with a "style" giving a richer appearance than a simple pen stroke. The fourth shows an example of stylistic imitation: The rendering starts from geometric models that have been enhanced with finely randomly sampled points; attached to each point is a brushstroke (one of several scanned images of actual oil-paint strokes) and a color (determined by a reference image, a lighted and shaded rendering of the original scene). Rendering consists of drawing (i.e., compositing into the final image) the strokes in a back-to-front order. There are many details remaining (the orientation of strokes, the creation of reference images, etc.), but the essential result is to give the appearance of a painting. If the strokes are similar to Monet's, and the reference image's coloring is similar to Monet's, and the chosen scene is similar to something that Monet might have painted, the final result will resemble a Monet painting.

34.1.2 Organization of This Chapter

Because expressive rendering is comparatively new, the overarching principles and structures for the area have not yet become apparent. The remainder of this chapter therefore consists of some general material that applies to enough different techniques that it deserves discussion, and a tour of some specific techniques that we think illustrate various important points, followed by some brief conjectures about future directions and related work.

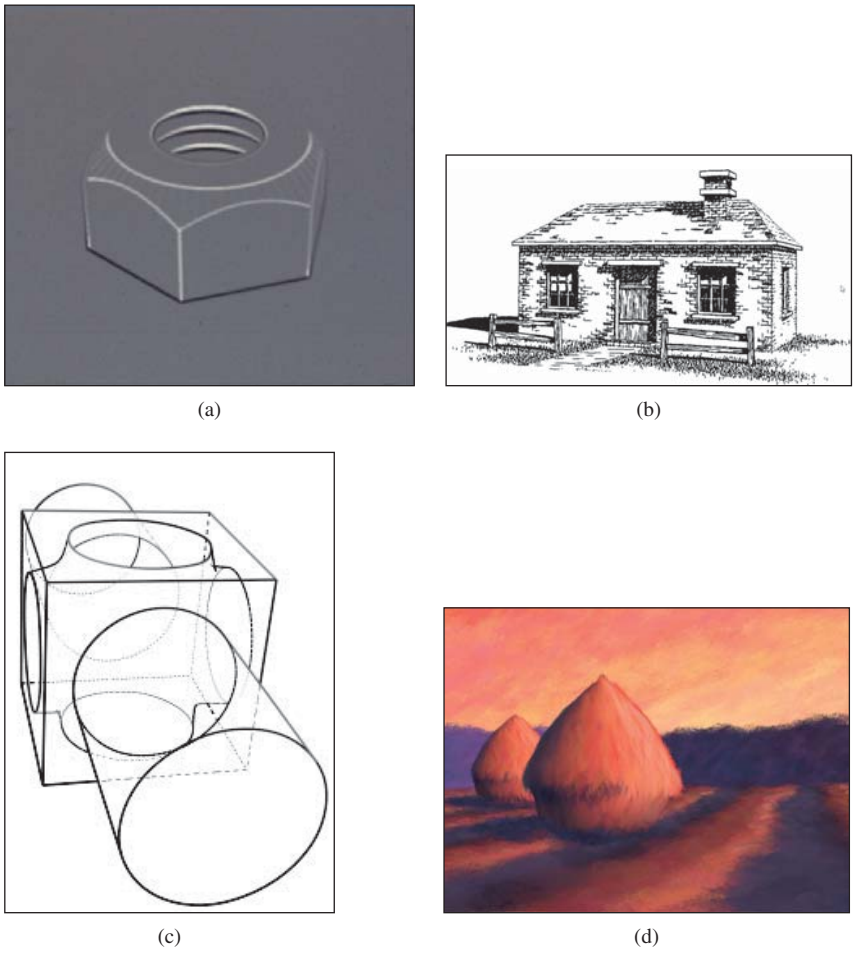


Figure 34.4: Four examples of expressive rendering. (a) Saito and Takahashi's depth-image approach is used to enhance contours and ridges with dark and light lines, respectively. (b) In Winkenbach and Salesin's [WS94] pen-and ink rendering, indication (the omission of repeated detail) is used to simplify the rendering of the roof and the brick walls. (c) In the work by Markosian et al. [MKG+97], contour curves are rapidly extracted and then assembled into longer strokes that can be stylized. (d) In Meier's painterly rendering work [Mei96], a back-to-front rendering of brush strokes, each attached to a point of some object in the scene, gives excellent temporal coherence as the viewpoint is altered. ((a) Courtesy of Takafumi Saito and Tokiichiro Takahashi, ©1990 ACM, Inc. Reprinted by permission. (b) Courtesy of David Salesin and Georges Winkenbach. ©1994 ACM, Inc. Reprinted by permission. (c) Courtesy of the Brown Graphics Group, ©1997 ACM, Inc. Reprinted by permission. (d) Courtesy of Barbara Meier, ©1996 ACM, Inc. Reprinted by permission.)

34.2 The Challenges of Expressive Rendering

While expressive rendering involves style, message, intent, and abstraction, the first of these is particularly ill-defined, which presents a serious problem. It's used to describe medium ("pen-and-ink style"), technique ("a stippled style"), mark-making action ("loose and sketchy style"), mark grouping or structure ("a textured style" or "a patterned style"), and broader notions like mood ("a film-noir style")

or even personality ("a lighthearted style"). Unfortunately, all of these things are loosely related. It's hard to imagine making an image that used, say, mosaic tiles, conveying a film-noir mood, but with a lighthearted style. A clear definition and characterization of style at all these levels remains elusive, but for problems like transferring style from one rendering to another, it's an essential ingredient.

At a more operational level, much expressive rendering work has concentrated on renderings of single objects, or scenes in which objects have similar sizes. Abstraction tends to operate on a scale of no more than an order of magnitude. Few expressive rendering systems have a broad enough range of application to be able to make an effective rendering of, say, Dorothy, from *The Wizard of Oz,* on the yellow brick road, surrounded by hilly fields, with the Emerald City in the distant background drawn with a few indicative strokes. Thus, *scale* remains an important challenge in expressive rendering.

Coherence is a general term for the relatedness of nearby items, whether in a single image (**spatial coherence**) or in a sequence of images (**temporal coherence**). Spatial coherence in expressive rendering arises in multiple contexts. For instance, if we decide to render object outlines using a wiggly line, we need to displace adjacent points of the outline by about the same amount, as in Figure 34.5. (If we displaced them by random amounts, the result would not be a line!) But as you can see in the figure, if we simply start making a wiggle at some point of an outline, when we return to that point the displacements may not match, and the *failure* to match manages to particularly attract the viewer's attention.

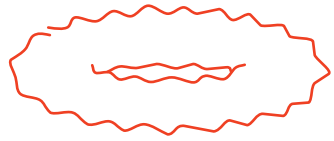


Figure 34.5: Mostly spatially coherent wiggles on the edges of a torus.

Temporal coherence is closely related. When we animate an expressive rendering, the strokes or other marks (e.g., tiles in a mosaic) vary over time. If the marks change rapidly from one frame to the next, the eye can be easily distracted. Proof of this can be seen by watching static on a broadcast (rather than cable) television. The average "frame" is a neutral gray, but as you watch, your eyes will detect patterns, notice things crawling or running across the screen, etc. If the marks in a rendering are something like stippling (a pattern of dots used to convey darkness or lightness) or a texture composed of short strokes, then even if the stipples or strokes have some temporal coherence (i.e., each stipple changes position slowly over time, or else disappears or appears, or each short stroke's endpoints move slowly over time), their motion can become a stronger perceptual cue than the marks themselves. For longer strokes, like long, thin pen-and-ink lines, the motion percept tends to be aligned perpendicular to the stroke; if the stroke corresponds to a contour, then such motion is consistent with contour motion, while motion *along* the stroke, as might appear when a small stroke that's part of a large contour shrinks before disappearing, is inconsistent with contour motion.

34.3 Marks and Strokes

Much expressive rendering is done with primitives that can be called **marks** or **strokes.** In stippling, for instance, each mark is a pen dot; in oil painting, each motion of the brush across the canvas is a stroke. In pen-and-ink rendering, a mixture of marks and strokes often serves to create texture, boundaries, etc. Not every form of expressive rendering uses marks and strokes (see Section 34.7), but many do. Why? First, many expressive rendering approaches mimic artistic techniques, and the use of strokes probably originated when some primitive human first picked up a stick and drew a shape in the dirt. So the simplest reason for marks and strokes is the ease with which we can create them. More important, though, is their power at triggering a response in the visual system. When we draw a

stick figure, or just a circle on a page, our minds can rapidly interpret this as the representation of a 3D shape—a human form, or a sphere. In fact, the tendency to see shape is almost overwhelming. Although when we look at a drawing we see pencil strokes on a piece of paper, when asked about it we always describe the thing depicted rather than saying, "I see a piece of paper with pencil strokes on it." This interpretation of strokes as representing shapes appears to be closely tied to our perceptual processes, in which edge detection is a first step. The strokes seem to manage to convey "edge-ness" directly, although each stroke, in principle, ought to convey *two* edges, one on either side of the stroke, as light transitions to dark and again as dark transitions to light.

Marks and strokes in expressive rendering systems are created by several approaches.

First there's the scanning/photography approach: An individual paint stroke on a canvas of contrasting color is photographed, and using the contrasting color, an α-value for each pixel near the boundary of the stroke is estimated. This is done for many example strokes, and then when it comes time to place strokes on the virtual canvas, the scanned strokes are recolored as needed and composited onto the canvas. The same approach has been used with charcoal and pencil marks, and can clearly be used with mosaic tiles, pastels, etc., as well. This approach may seem to be the simplest, but the actual scanning and processing of oil-paint strokes, for instance, turns out to be quite difficult.

Second, there's imitation of artistic technique, such as the pen-and-ink strokes used by Salisbury et al. [SWHS97], in which the core system determined the need for a curved stroke following some general path, and then the stroke-generating system created a spline path that approximated the general path, but had small perturbations in the normal direction to emulate the wiggliness of hand-drawn lines (the degree of wiggle was user-controllable), and which tapered to a point at each end, with the length of the taper also being user-controllable. Similar approaches were used by Northrup et al. [NM00] for a watercolor-like rendering. The idea of expanding or varying a stroke in the normal direction is a good one, but problems arise at **focal points,** where nearby normal lines cross (see Figure 34.6). These problems can be mostly addressed by adjusting the notion of "normal lines" to allow some bending and compression; Hsu et al. took this approach in their work on skeletal strokes [HLW93].

Finally, there's the physical simulation of media and tools. Physical simulation, of course, depends on a model of the thing being simulated, and such models may range from very accurate to somewhat informal. Curtis [CAS+97] used fluid simulation to model the flow of water and pigment in watercolors (see Figure 34.7); Strassmann [Str86] used a minimal physical model of a brush (a linear array of bristles of slightly varying lengths, each of which responds to both paper texture and pressure applied by the user), paper, and ink to allow a user to create sumi-e paintings like the one shown in Figure 34.8; Baxter and Lin [BL04] extended this model to handle far greater complexity, including interbristle coherence and physically based deformation of the brush and bristles.

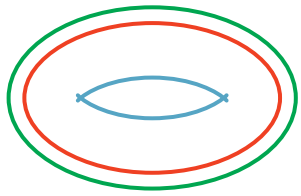


Figure 34.6: A small offset from the central red curve results in the smooth green outer curve; at the focal distance in the other direction, where nearby normals meet, we get a degeneracy—the sharp ends of the blue inner curve.

Figure 34.7: A watercolor produced with Curtis's system. (Courtesy of Cassidy Curtis. ©1997 ACM, Inc. Reprinted by permission.)

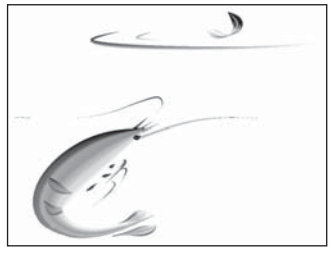


Figure 34.8: A sumi-e painting created with Strassmann's system. (Courtesy of Steve Strassmann. ©1986 ACM, Inc. Reprinted by permission.)

34.4 Perception and Salient Features

As we discussed in Chapter 5, the human visual system is sensitive to certain characteristics of arriving light, and not so sensitive to others. Those to which it's most sensitive are good candidates for inclusion in an expressive rendering.

Candidates for important features are silhouettes, contours on geometric models, apparent contours, suggestive contours, and places where the light field can be condensed to one line. All of these fit under the general category of **edges,** as the term is used in computer vision, that is, places where the brightness changes rapidly. Such edges can exist at multiple scales, in that something that presents a gradual change in brightness, seen up close, may represent a rapid transition when seen from farther away. There's some evidence [Eld99] that edges, considered at all scales, completely characterize an image. This idea, in reverse, is at the heart of recent work on gradient-based expressive rendering techniques, which we discuss in Section 34.7.

An alternative to reasoning about where lines *should* be drawn is to observe where they actually *are* drawn. Cole et al. [CGL+12] have performed a carefully constructed experiment to see where artists draw lines in single-object illustrations, given several views of the object to look at during the drawing process. They find that occluding contours and places with large image gradients are strongly favored, but these do not by any means account for all the lines that are drawn.

There are larger-scale issues in expressive rendering as well. In drawing a picture of two people standing on a bridge in Paris, we're likely to concentrate on the bridge and the people, sketch the general shapes of the buildings in the background, and perhaps include some added detail on the Eiffel Tower. These choices represent the features in the scene that are *salient* to us, but there's no way to algorithmically determine saliency from the image data without an understanding of the full scene; in larger-scale imagery, expressive rendering at present must rely on additional user input to determine the saliency of even details that may be strongly significant in terms of perception.

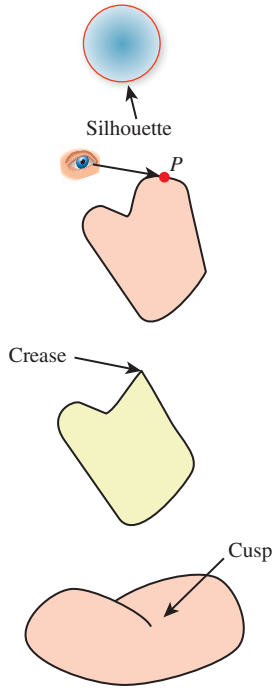


Figure 34.9: (Top) The silhouette separates foreground from background. (Top middle) P is on a contour if the ray from the eye to P is tangent to the surface at P. (Lower middle) A crease is a point at which nearby tangent planes converge to two different limits; the definition can be weakened to give a notion of a crease at a certain scale. (Bottom) A cusp is a point Q of a contour curve C at which the tangent line to C is the same as the line from the eye to Q.

34.5 Geometric Curve Extraction

Because geometric characteristics of objects, like their boundaries, arise in expressive rendering, and because these have also often been studied in geometry, there's a well-defined vocabulary in place; unfortunately, usages differ between mathematics and graphics. We'll adhere to the mathematical conventions.

First, when an object sits in front of a background, the **silhouette** is the boundary between the object's image and the background (see Figure 34.9). For a smooth object like a sphere, if S is a silhouette point, then the tangent plane at S contains the ray from the eye to S; points where the tangent contains the view direction are called **contour points**; the set of all contour points is called the **contour.** Thus, for a smooth object, every silhouette point is a contour point, or, equivalently, the silhouette is a subset of the contour. But there may be many other contour points as well, as seen in Figure 34.9 (bottom) where the contour extends into the interior of the surface.

The condition for a point P of a smooth surface to be a contour point, as viewed from an eyepoint C, is that

$$\mathbf{n}(P) \cdot (P - C) = 0, \tag{34.1}$$

that is, that the surface normal be orthogonal to the view vector.

Unfortunately, contour points have been called silhouettes in several graphics papers, blurring the distinction between the two notions. Note that a contour point S may be visible or not: All that's required is that the ray from the eye to S be

contained in the tangent plane at S. There are two other conventions: In the first, what we have called the **contour** is sometimes called the **contour generator,** and the term "contour" is reserved for what we would call the **visible contour;** in the second, the thing we call the contour is called the **fold set,** although the definition for the fold set is somewhat more general, extending to polyhedra as well as smooth surfaces [Ban74].

In the case of nonsmooth objects like polygonal meshes, notions like "tangent plane" and "normal vector" must be adjusted. One approach is to create a smoothly varying normal vector field on the surface, one that agrees with the facet normal at facet centers, for instance, but smoothly blends between them, away from the centers. This approach works decently, although with this approach it's easy to have a silhouette point (on the boundary between object and background) that is not a contour (view ray perpendicular to normal vector), which can lead to problems. Another approach is to say that at each edge between two facets there is a whole *set* of normal vectors, filling in between the normal vectors of the two facets. A simple version of this is shown in Figure 34.10: Looking end-on at an edge e, we treat the two adjacent facet normals $\mathbf{n}_1$ and $\mathbf{n}_2$ as points of the unit sphere; we then find a great-circle arc between them, and say that all vectors along this great-circle arc are normals to the edge e. The only time there's an ambiguity about which great-circle arc to pick is when the adjacent normals are opposites; in this case the polyhedral surface is degenerate (the interiors of adjacent facets intersect), and the approach fails for such surfaces (just as we cannot, in the smooth-surface case, handle normal vectors at nonsmooth points).

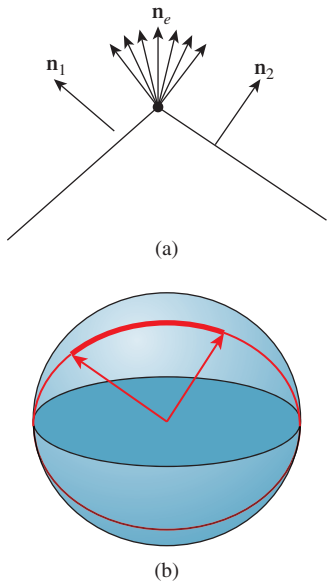


Figure 34.10: (a) The collection of normals at an edge e of a mesh interpolates between the normals $\mathbf{n}_1$ *and* $\mathbf{n}_2$ *of the adjacent facets. (b) Drawing* $\mathbf{n}_1$ *and* $\mathbf{n}_2$ *at the origin, so their tips are on the unit sphere, we use great-circle interpolation to create the set of normals for the edge e.*

This approach can be extended to define a set of normals at a vertex as well: The normal arcs for each edge adjacent to the vertex link together into a chain (on the unit sphere); we declare the normal at the vertex to be the interior of this loop. Once again, there are degenerate cases: Since any simple closed curve on the sphere is the boundary of two different regions, we must make a choice between these. If the polyhedral surface is nearly flat at the vertex, then all adjacent normals are near each other, and we can simply choose the region whose area is smaller. The degeneracy arises when the two areas are equal. In practice, in meshes derived from reasonably uniform and fine sampling of smooth surfaces, such vertices do not often arise.

Polygonal meshes are, generally speaking, a bad starting point for things like contour extraction, because almost every edge of a mesh represents a sharp change in the normal vector. In some cases, this results from the polygonalization of a smooth object. In others, the sharp edge is modeling a sharp edge on the original object (e.g., a cube). Without further information, it's impossible to tell which kind of edge is intended. Various researchers have experimented with various thresholds, but you need only consider a finely faceted diamond to realize that there's no obvious threshold that can work for all objects. It's probably best, as a practical matter, to allow the modeler of an object to mark certain edges as **crease edges,** that is, those across which the normal is supposed to change rapidly, and then treat all others as **smooth edges,** or those across which the normal is to be interpolated smoothly. This is an instance of the principle stated in the introduction, that you should understand the phenomena and goal of your effort, and only then choose a rich-enough abstraction and representation to capture the important phenomena. The "polygonal model" representation of shape was chosen before the advent of expressive rendering, and it lacks sufficient richness. It's also an instance of the Meaning principle: The "numbers" in the polygonal model don't have sufficient meaning attached to them.

With that in mind, Listing 34.1 is a simple algorithm for rendering the visible contours and crease edges of a smooth polygonal shape that represents a surface with no self-intersections so that each edge is either shared by two faces or on the boundary.

Listing 34.1: Drawing the visible contours, boundary, and crease edges of a polygonal shape from the point Eye.

```
Initialize z-buffer and projection matrix
Clear z-buffer to maximum depth
Clear color buffer to all white
Render all faces in white

edgeFaceTable = new empty hastable with edges as keys and faces as values
outputEdges = new empty list of edges

foreach face f in model:
  foreach edge e of f:
    if e is a crease edge:
      outputEdges.insert(e)
    else if e is not in edgeFaceTable:
      edgeFaceTable.insert(e, f)
    else:
      eyevec = e.firstVertex - Eye
      f1 = edgeFaceTable.get(e) // get other face adjacent to e
      if dot(f1, eyevec) * dot(f, eyevec) < 0:
        outputEdges.insert(e)
        edgeFaceTable.remove(e)

foreach edge in edgeFaceTable.keys():
  outputEdges.insert(edge)

Render all edges in outputEdges in black
```

The key ideas in this algorithm are that boundary edges are those that appear in only one polygon, and hence they are left in the table after all pairs have been processed, and that a pair of faces that are adjacent at some edge make a contour edge if one face normal points toward the eye and the other points away. Thus, the list of output edges consists of all contour, crease, and boundary edges. The rendering of the surface in white as an initialization prevents hidden output edges from being seen (i.e., it generates occlusions). In practice, we often draw each contour edge slightly displaced toward the eye so that it is not hidden by the faces it belongs to. This slight displacement can unfortunately let a very slightly hidden contour be revealed, but in practice the algorithm tends to work quite well. Chapter 33 gives a rather different approach to generating a contour rendering that does not suffer from this problem, but that does not handle boundary edges or crease edges.

Note that the preceding algorithm generates a list of edges to be drawn, but it does not try to draw a stroke along the contour; it merely renders each edge as a line segment. If you want to draw a long smooth curve (perhaps using the vertices of the edges as control points for a spline curve, or even making a slightly wiggly curve to convey a hand-drawn "feel") you need to assemble the edges into chains in which each edge is adjacent to its predecessor and successor in the list. For a smooth closed surface, such chains exist, and for a generic view, the chains form closed curves on the surface (i.e., cycles). (The crease edges may form noncyclic chains, however.) Unfortunately, for a polygonal approximation of a smooth surface, there's no such simple description of the contours. As Figure 34.11 shows,

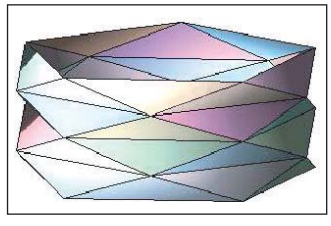


Figure 34.11: A triangulated cylinder in which every edge is a contour edge when viewed from directly overhead.

it's possible for almost every edge to be a contour from some points of view, in which case forming chains by the obvious greedy algorithm ("search for another edge that shares this vertex, and add it to my cycle") can fail badly, generating contour cycles that intersect transversely, for instance.

You may object that this example is contrived, but even a randomly triangulated cylinder, when viewed end-on, can have a great many contour edges (see Figure 34.12). Fortunately, all of these project to the same circle in the final image, but attempting to make coherent strokes on the contours remains problematic [NM00].

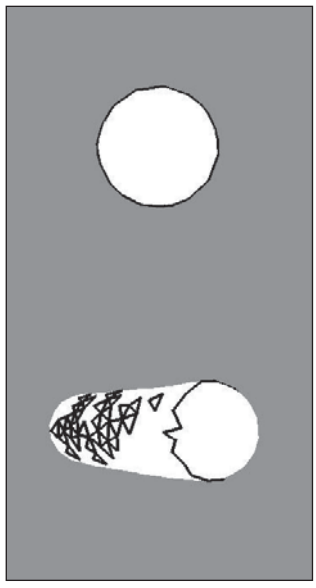


Figure 34.12: The contours of a lozenge shape, viewed end-on, appear to form a circle, but from a different view they are quite complex. (Courtesy of Lee Markosian, ©2000 ACM, Inc. Reprinted by permission.)

The problem of multiple contours, and contours that are not smooth, as well as other artifacts of polygonal contour extraction, are largely addressed by the work of Zorin and Hertzmann [HZ00], who observe that "no matter how fine the triangulation is, the topology of the silhouette of a polygonal approximation to the surface is likely to be significantly different from that of the smooth surface itself." They have the insight that the function

$$g(P) = \mathbf{n}(P) \cdot (P - C) \tag{34.2}$$

can be computed at each mesh vertex and interpolated across faces, and then the zero set of this interpolated approximation of g can be extracted and called the contour. By slightly adjusting the value of g at any vertex where it happens to be zero, they ensure that the contour curves so formed consist of disjoint polygonal cycles, which are ideal for stroke-based rendering. Figure 34.13 shows an example of such a contour rendering with "hatching" used to further convey the shape.

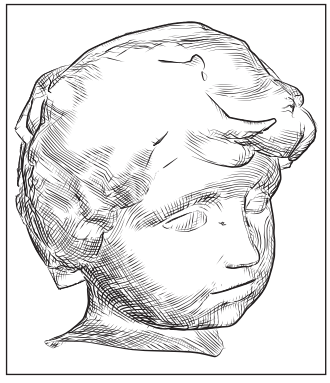


Figure 34.13: A shape whose contours are rendered via the Zorin-Hertzmann algorithm, with interior shading guided by curvature. (Courtesy of Denis Zorin, ©2000 ACM, Inc. Reprinted by permission.)

We now move on to suggestive contours, ridges, and apparent ridges. To discuss these features, we must discuss **curvature.** Recall from calculus that the curvature of the graph of $y = f(x)$ at the point (x, y) is given by

$$\kappa = \frac{f''(x)}{(1 + f'(x)^2)^{3/2}}. \tag{34.3}$$

In the case of a parametric curve $t \mapsto (x(t), y(t))$, the formula is

$$\kappa = \frac{x'(t)y''(t) - y'(t)x''(y)}{(x'(t)^2 + y'(t)^2)^{3/2}}. \tag{34.4}$$

For a polygonal curve, which is often what we have in practice, there are simple approximations to these formulas, although you may be better off fitting the polygonal curve with a spline and then computing the spline's curvature.

Inline Exercise 34.1: Confirm that if we take the graph $y = f(x)$ and make it into a parametric curve using $X(t) = t$ and $Y(t) = f(t)$, the two curvature formulas agree.

For a surface, curvature is slightly more complex. If you think of a point P on a cylinder of radius r, there are many possible directions in which to measure curvature (see Figure 34.14): In the direction parallel to the axis, the curvature is zero, while perpendicular to it, the curvature is $1/r$. To measure each of these, we intersect the cylinder with a plane through P containing the normal vector $\mathbf{n}$ and direction $\mathbf{u}$ in which we want to measure the curvature. The intersection is a curve in this plane, whose curvature at P we can measure using Equation 34.3 or Equation 34.4.

For the cylinder, the two curvatures we've described are in fact the maximum and minimum possible over all surface directions $\mathbf{u}$ at P. Because of this, they are called the **principal curvatures,** typically denoted κ_1 and κ_2, with the associated directions being called the **principal directions.** (One approach to expressive rendering for surfaces involves drawing strokes aligned with one or two principal directions [Int97].) The two principal directions are *orthogonal;* this turns out to be true at every point of every surface (except when the principal curvatures are the same, in which case the principal directions are undefined; such points are called **umbilic**). Furthermore, the principal directions $\mathbf{u}_1$ and $\mathbf{u}_2$ and their associated curvatures κ_1 and κ_2 completely determine the curvatures in every other direction; if

$$\mathbf{u} = \cos(\theta)\mathbf{u}_1 + \sin(\theta)\mathbf{u}_2, \tag{34.5}$$

then the curvature in the direction $\mathbf{u}$ (or **directional curvature in direction u**) is

$$\cos^2(\theta)\kappa_1 + \sin^2(\theta)\kappa_2. \tag{34.6}$$

Note that this formula does not depend on the orientation of $\mathbf{u}_1$ or $\mathbf{u}_2$: If we negate $\mathbf{u}_2$ (giving an equally valid "principal direction"), for instance, the sign of θ changes, which alters the sign of $\sin(\theta)$, but leaves $\sin^2(\theta)$ unchanged.

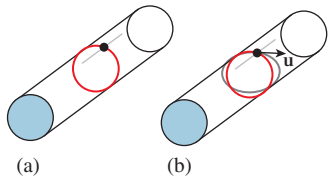


Figure 34.14: (a) The two principal curvatures on a cylinder: Along the axis of the cylinder, the curvature is zero; in the perpendicular direction, the curvature is $1/r$, where r is the cylinder radius. (b) To measure the curvature in some direction $\mathbf{u}$ at P, we intersect the surface with the plane through P containing $\mathbf{u}$ and $\mathbf{n}$, the normal to the surface. The result is a curve (gray) in a plane, whose curvature we can measure.

34.5.1 Ridges and Valleys

The principal direction $\mathbf{u}_1(P)$ corresponding to the maximum directional curvature at each point P is used to define the notion of a **ridge** or **valley:** The principal directions can be joined together into a curve called a **line of curvature**[1] (see Figure 34.15). As we traverse a line of curvature, the principal curvature κ_1

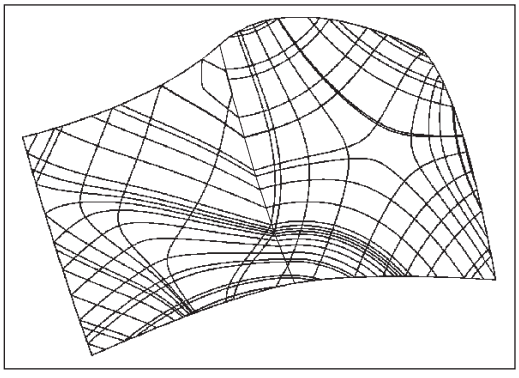


Figure 34.15: The curves in this diagram have tangents that are in the direction of either greatest or least curvature. The "bends" near the center occur because the surface is defined by two adjacent spline patches. (Courtesy of Nikola Guid and Borut Žalik. Reprinted from Computers & Graphics, *volume 19, issue 4, Nikola Guid, Črtomir Oblonšek, Borut Žalik, "Surface Interrogation Methods," pages 557–574, ©1995, with permission from Elsevier.)*

1. More explicitly: We can find a curve $t \mapsto \gamma(t)$ on the surface with the property that $\gamma'(t) = \mathbf{u}_1(\gamma(t))$ for every t, and $\gamma(0) = P$; this is the line of curvature through P.

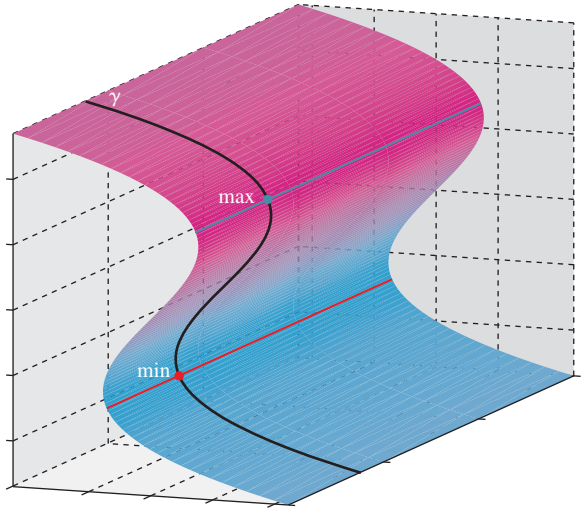


Figure 34.16: As we traverse the line of curvature γ, the directional curvature of the surface S in the direction of γ varies from point to point. We've marked a local maximum and minimum; these are ridge and valley points, respectively. Shown in red (valley) and blue (ridge) lines are the rest of the ridge and valley points of the surface.

changes from point to point. Local minima and maxima of the principal curvature along a line of curvature are called **ridges** and **valleys,** respectively (see Figure 34.16). These curves have been used to help communicate shape, but they suffer from two problems. The first is that, in practice, algorithms for computing ridges and valleys tend to be "noisy," that is, they tend to produce lots of short segments, which are distracting rather than informative. The second is that ridges and valleys often occur in pairs, so we end up with two lines where an artist would draw only one (see Figure 34.18 for an example).

34.5.2 Suggestive Contours

Instead of using the principal curvature directions on a surface to define curves along which to find local maxima and minima, as we did for ridges and valleys, we can use a different vector field, one that depends on our *view* of the surface rather than being intrinsic to the surface itself, independent of view, as are ridges and valleys. This is the approach taken by DeCarlo et al. [DFRS03]. We'll follow their development.

We let $\mathbf{v}(P) = E - P$ denote the view vector at each point of the surface. Notice that this vector points *from* P toward the eye E. And we let $\mathbf{w}(P)$ denote the projection of $\mathbf{v}(P)$ onto the tangent plane at P; omitting the argument P, $\mathbf{w}$ is defined by

$$\mathbf{w} = \mathbf{v} - (\mathbf{v} \cdot \mathbf{n})\mathbf{v}, \tag{34.7}$$

where $\mathbf{n} = \mathbf{n}(P)$ is the unit normal to the surface at the point P. The curvature in the direction $\mathbf{w}$ is called the **radial curvature** κ_r**.** Note that the radial curvature depends both on the point P *and* on the location of the eye: It's not an intrinsic property of the surface.

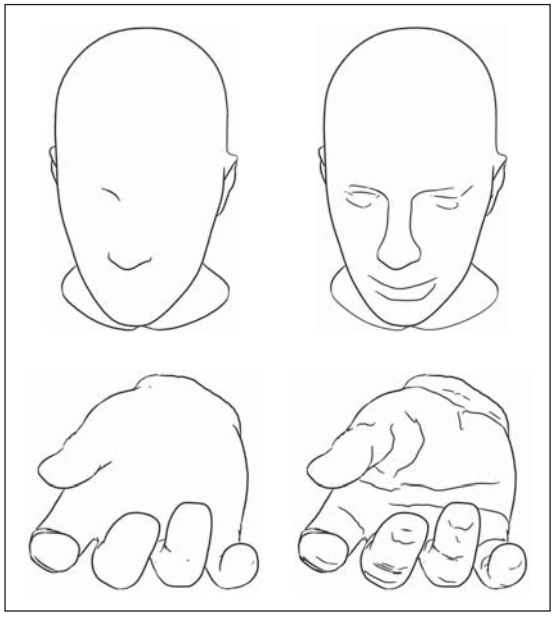


Figure 34.17: Contours of two shapes (left) together with suggestive contours (right). (Courtesy of Doug DeCarlo. ©2003 ACM, Inc. Reprinted by permission.)

Suggestive contours are the places where the radial curvature is zero, with the additional constraint that the derivative of κ_r, in the $\mathbf{w}$ direction, must be positive (i.e., if we move from P to $P + \epsilon\mathbf{w}(P)$, and then project the resultant point back to the surface and call it Q, then $\kappa_r(Q)$ must be positive for small enough values of ϵ). DeCarlo et al. use the term "suggestive contour generator" for these points, restricting "suggestive contours" to what we would call "visible suggestive contours." Suggestive contours can be characterized in two other ways. First, they are local minima of $\mathbf{n} \cdot \mathbf{v}$ in the $\mathbf{w}$ direction; if we consider ordinary contours as places where $\mathbf{n} \cdot \mathbf{v}$ is zero, then suggestive contours are the points where $\mathbf{n} \cdot \mathbf{v}$ got closest to zero before increasing again. This is closely related to the second characterization: If we move the eyepoint E, the contours appear to slide along the surface (think of the edge of night moving along the Earth as it rotates). But sometimes, in the course of such a motion, a new piece of contour appears; points of that new contour lie on the suggestive contour for the original eyepoint. As DeCarlo et al. describe it, suggestive contours are "those points that are contours in 'nearby' viewpoints, but do not have 'corresponding' contours in any closer views." In short, they might be characterized as places that are almost contours, or, if you prefer, as places that, if the surface were lit by a light source near the eyepoint, would be at the boundary between dark and light. As such, they are natural places to put lines to help indicate shape (see Figure 34.17).

34.5.3 Apparent Ridges

In the examples discussed so far, there are a range of characteristics: Contours are view-dependent, while ridges and valleys are view-independent (except that we only draw the visible ones, of course). Suggestive contours are view-dependent as well, while capturing some of the "where do we expect to see lines?" character: If

Figure 34.18: Shaded view, contours, suggestive contours, ridges and valleys, and apparent ridges for a single model. Courtesy of Tilke Judd and Frédo Durand, ©2007 ACM, Inc. Reprinted by permission.)

the surface is wiggly enough somewhere, then that point's likely to be on a suggestive contour. There is another kind of line—apparent ridges [JDA07]—that is also view-dependent (see Figure 34.18). In this case, however, the view dependence has an interesting character: It takes into account the projection of the surface to a particular view plane, and performs measurements in that view plane rather than on the surface itself. Since the projection of the object to the view plane captures what we can *see,* measurements made in this plane correspond to operations that our eyes could possibly perform. Thus, a slight curvature near a contour is drawn, while the same amount of curvature in a frontal region of an object is not.

34.5.4 Beyond Geometry

While geometric characteristics are important for determining which lines to draw, other characteristics may matter as well, such as texture: The lines between stripes on a plaid shirt should be drawn along with the contours of the shirt. This is essentially a reversion to the computer-vision notion of *edges,* discontinuities in brightness at some scale. This idea was implemented as an expressive rendering scheme by Lee et al. [LMLH07]. They first rendered a scene with traditional shading, and then used this preliminary rendering as a source for the final rendering. They searched in the preliminary rendering for brightness discontinuities, which were then rendered as lines in the final rendering, typically representing contours or texture changes like the stripes on a shirt. They also searched for thin, dark regions, which were rendered as dark lines, and thin, light regions, which were rendered as highlight lines (see Figure 34.19). The large-scale shading in the final rendering was done with **two-tone shading** [LMHB00], created by thresholding the preliminary image on brightness.

Figure 34.19: Abstracted shading with highlights. (Courtesy of Seungyong Lee, ©2007 ACM, Inc. Reprinted by permission.)

34.6 Abstraction

The representation of a shape by lines is a kind of abstraction. Of the three kinds of abstraction we mentioned (simplification, factorization, and schematization) this form of shape representation falls into the first or second category, involving the

elimination of detail, which in some cases may move the object depicted from the specific toward the general.

But even when we represent a shape by its contours or other lines, we may have far more detail than we want. A rough lump of granite has a great many contour edges, but we may want to draw just the outline, ignoring all the tiny interior contours provided by individual protrusions from the surface. Thus, there's a relation between scale and abstraction. One rule of thumb is that objects of equal importance in a scene should be represented with a number of strokes that is proportional to their projected size (or perhaps the square root of the projected size, since a circle of area A has an outline whose length is $2\sqrt{\pi A}$). Regardless, it's evident that there's a need to not only determine lines that represent a shape, but also to determine a line-based representation with a given budget for lines—to remove or simplify lines to give a less dense representation.

Two approaches immediately come to mind. The first is to simplify the object itself (e.g., if it's a subdivision surface, move up one level of subdivision to get a less-accurate but simpler representation) and then extract lines. The second is to extract the lines and then simplify them.

Inline Exercise 34.2: Think of shapes for which each method would give results that don't match your intuition. Which method was harder to "break"?

The second approach was carried out by Barla et al. [BTS05]. Their algorithm takes, as input, a set of lines (in the sense of line drawing, i.e., curves) in some vector representation, and produces a new set of lines using two criteria that are intended to create perceptually similar sets of lines.

1. New lines can be created only where input lines appear.
2. New lines must respect the shape and orientation of input lines.

The idea is then to form "clusters" of input lines that are perceptually similar at a specified scale and replace these with fewer lines, thus simplifying the drawing.

The second criterion is formulated to allow merging two nearly parallel curves into a single curve that's approximately their average, but *not* into a single curve with a hairpin bend at one end, for example. Figure 34.20 shows this. The hairpin bend in the pink line in the drawing on the left is a bad simplification of the input black lines, while the two pink lines in the drawing on the right are a better simplification.

The algorithm proceeds by first finding clusters of lines that are similar at the chosen scale, and then replacing each cluster with a single line. The replacement strategy may be as simple as selecting a single representative from the cluster, or as complex as creating some kind of "average" line from the lines in the cluster. Figure 34.21 shows the results using the second approach.

At a higher level, it makes sense to take a whole *scene* and abstract out those parts that are not important to the author or viewer. Determining what might be important is impossible without either an understanding of the full scene or knowledge of the intent (i.e., *a priori* markup of some kind). The latter approach can be used as part of an authoring tool to guide the scale of simplification in algorithms like that of Barla et al. DeCarlo and Santella [DS02] have taken the former approach, using a *human viewer* to implicitly provide scene understanding. Their system takes an image as input and transforms it to a line drawing with

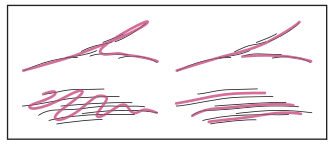


Figure 34.20: The thick pink lines at right are good simplifications of the thin black input lines; those at left are bad. (Courtesy of Pascal Barla. From "Rendering Techniques 2005" by Bala, Kavita. Copyright 2005. Reproduced with permission of Taylor & Francis Group LLC - BOOKS in the format Textbook via Copyright Clearance Center.)

Figure 34.21: A total of 357 input lines are simplified to 87 output lines. (Courtesy of Pascal Barla. From "Rendering Techniques 2005" by Bala, Kavita. Copyright 2005. Reproduced with permission of Taylor & Francis Group LLC - BOOKS in the format Textbook via Copyright Clearance Center.)

large regions of constant color and bold edges between regions, representing an abstraction of the scene according to the parts-and-structures kind of hierarchy present in many computer-vision algorithms. To effect this transformation, they use an eye-tracked human viewer. Broadly speaking, the eye tracking allows them to determine which elements of the image are most attention-grabbing, and thus deserve greater detail. Figure 34.22 shows a sample input and result.

It's also possible to consider a stroke-based rendering of a scene *over time,* and try to simplify its strokes in a way that's coherent in the time dimension; to do so effectively requires a strong understanding of the perception of motion (for strokes that vary in position or size) and change (for strokes that appear or disappear).

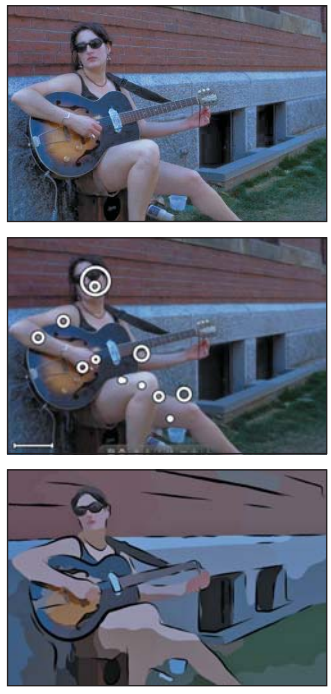


Figure 34.22: (Top to bottom) The input photograph, the eye-tracker fixation record, and the resultant image for DeCarlo and Santella's abstraction and simplification algorithm. (Courtesy of Doug DeCarlo and Anthony Santella, ©2002 ACM, Inc. Reprinted by permission.)

34.7 Discussion and Further Reading

The two abstraction techniques presented in this chapter fall into the simplification and factorization categories. Is it possible to also do schematization? Can we learn schematic representations from large image and drawing databases, for instance? This remains to be seen.

Much of what's been done in expressive rendering until now has emulated traditional media and tools. But the computer presents us with the potential to create new media and new tools, and thinking about these may be more productive than trying to imitate old media. Two examples of this are the **diffusion curves** of Orzan et al. [OBW$^+$08] and the **gradient-domain painting** of McCann and Pollard [MP08]. Each relies on the idea that with the support of computation, it's reasonable for a user's stroke to have a *global* effect on an image.

In the case of gradient-domain painting, the user edits the *gradient* of an image using a familiar digital painting tool interface. A typical stroke like a vertical line down the middle of a gray background will create a high gradient at the stroke so that the gray to the left of the stroke becomes darker and the gray to the right becomes lighter, and the stroke itself ends up being an edge between regions of differing values. (This is an "edge" in the sense of computer vision, by the way.) By adjusting the stroke width and the amount of gradient applied, the user can get varying effects. The user can also grab a part of an existing image's gradient and use *that* as a brush, allowing for further interesting effects. To be clear: The image the user is editing is not precisely the gradient of the final result; rather, an integration process is applied to the gradient to produce a final image with the property that its true gradient is as near to the user-sketched gradient as possible. Figure 34.23 shows an example of photo editing using gradient-domain painting with a brush whose gradient is taken from elsewhere in the image.

In diffusion curves, the user again has a familiar digital painting interface, but in this case each stroke draws boundary conditions for a diffusion equation: In the basic form, on one side of the stroke the image is constrained to have a certain color; on the other side it has a different color. The areas in between the strokes have colors determined from the stroke values by diffusion (i.e., each interior pixel is the average of its four closest neighbors). Again, a single stroke can drastically affect the whole image's appearance. But if we think instead about perceptually significant changes, we see the effects are quite local: In the nonstroke areas, the values change very smoothly so that there are no perceptually significant edges. Thus, the medium of diffusion curves allows the artist to work directly with perceptually significant strokes. Figure 34.24 shows an example of the results.

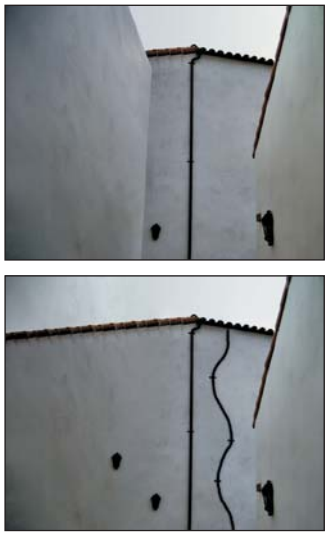


Figure 34.23: Photo editing with gradient-domain painting. The roof tiles and drain pipe have been altered, and the left wall eliminated, all in just a few strokes. (Photo courtesy of Christopher Tobias, ©2009; Courtesy of Nancy Pollard and James McCann, ©2008 ACM, Inc. Reprinted by permission.)

In both of these media, animation is quite natural. As strokes interpolate between specified positions at key frames, the global solutions for which they prescribe boundary conditions also change smoothly. Temporal coherence is almost automatic. The exception is in handling the appearance and disappearance of strokes, which still must be addressed. Nonetheless, the global effect of each stroke means that animation in one portion of an image can generate changes elsewhere, which may distract the viewer. By the way, temporal coherence in *stylized* strokes—things like wiggly lines in pen-and-ink renderings—remains a serious challenge as of 2013.

Video games are now often using deliberately nonphotorealistic techniques to establish mood or style, but maintaining a consistent feel throughout a game requires high-level art direction as well as an expressive rendering tool. There's a need for tools to assist in such art direction, and for authoring tools for scenes to be nonphotorealistically rendered so that modelers can indicate objects' relative importance (and how these change over time) as cues to simplification algorithms.

Tools like gradient-domain painting and diffusion curves let artists work with what might be called "perceptual primitives," but at the cost of global modifications of images, which may be inconvenient. It would be nice to find semilocal versions of these tools, ones whose range of influence can be conveniently bounded.

Figure 34.24: A figure drawn with just a few diffusion curves. (©L. Boissieux, INRIA) (Courtesy of Joelle Thollot.)

Chapter 35

Motion

35.1 Introduction

When you see a sequence of related images in rapid succession, they blend together and create the perception that objects in the images are moving. This need not involve a computer: Cartoons drawn in a flip-book and analog film projection (see Figure 35.1) both create the illusion of motion this way. The individual images are called **frames** and the entire sequence is called an **animation.** Beware that both of these terms have additional meanings in computer graphics; for example, a coordinate transform is a "reference frame" and an "animation" can refer to either the rendered images or the input data describing one object's motion.

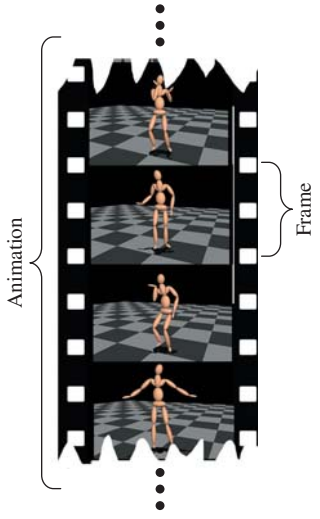


Figure 35.1: An animation is a sequence of frames.

This chapter presents some fundamental methods for describing the motion of objects in a 3D world over time. These mainly involve either interpolating between key positions (see Figure 35.2) or simulating dynamics according to the laws of physics (see Figure 35.3). Note that the laws of physics as in a virtual world need not be those of the real world.

Most character animation that you have observed was driven by key positions, with those positions created either by an artist or via motion capture of an actor (see Figure 35.4). Those processes are not particularly demanding from a computational perspective, but producing the animations is expensive and relatively slow because of the time and skill that they require from the artists in the process. In contrast, dynamics is computationally challenging but requires comparatively little input from an artist. This is a classic example of leveraging a computer to multiply a human's efforts dramatically. It is natural that as animation algorithms have become more sophisticated and computer hardware has become both more efficient and less expensive, the broad trend has been to increase the amount of animation produced by dynamics.

The artistry and algorithms of animations are subjects that have filled many texts, and thus even a survey would strain the bounds of a single chapter. This chapter focuses on the rendering and computational aspects of *physically based* animation. In particular, it emphasizes *concepts* in the interpolation and rendering sections and *mathematical detail* in the dynamics section. Dynamics is primarily

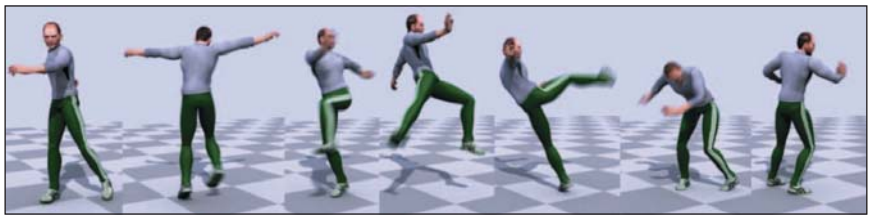


Figure 35.2: A series of key poses extracted from dense motion capture data [SYLH10] that have been visualized by rendering a virtual actor in those poses. (Courtesy of Moshe Mahler and Jessica K. Hodgins.)

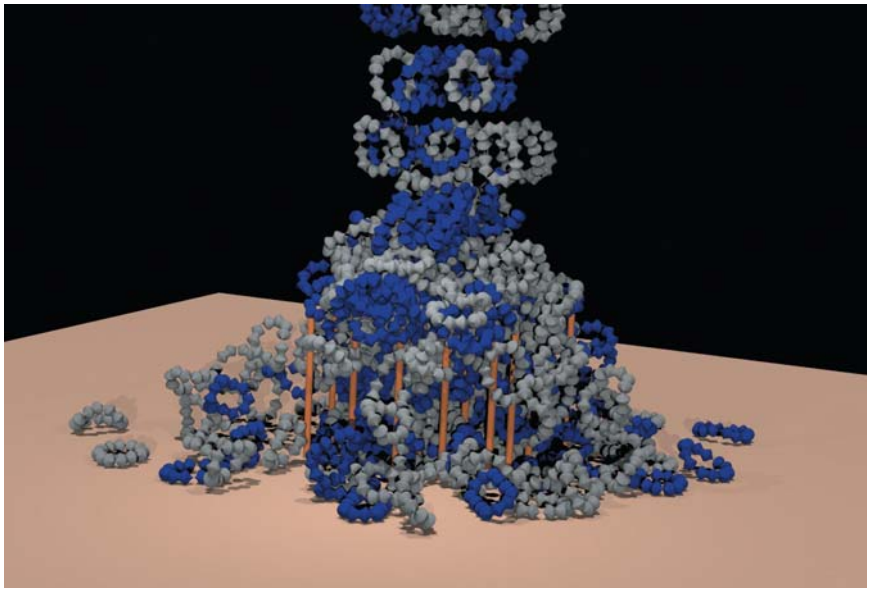


Figure 35.3: Hundreds of complex shapes fall into a pile in this rigid-body simulation of Newtonian mechanics [WTF06]. This kind of simulation is leveraged extensively to present "what if?" scenarios for both entertainment and engineering applications. The primary challenges are efficiency and numerical stability. (Courtesy of Ron Fedkiw and Rachel Weinstein Petterson, ©2005 ACM, Inc. Reprinted by permission.)

concerned with numerical integration of estimated derivatives; thinking deeply about the underlying calculus should help you navigate the notorious difficulty of achieving stability and accuracy in such a system. The techniques used are related to several other numerical problems beyond physical simulation. Notably, the integration methods that arise in dynamics serve as another example of the techniques applied to the integration of probability density and radiance functions in light transport.

There are many other ways to produce animations that are not discussed in this chapter. Two popular ones are filtering live-action video (e.g., as shown in Figure 35.5) and computing per-pixel finite automata simulation (e.g., Conway's Game of Life [Gar70], Minecraft, and various "falling sand" games). Although beyond the scope of this book, both filtering and finite automata make rewarding graphics projects that we recommend to the reader.

Figure 35.4: Motion capture systems, such as the InsightVCS system pictured, record the three-dimensional motion of a real actor and then apply those motions to avatars in the virtual scene. (Courtesy of OptiTrack.)

Figure 35.5: Video tooning creates an animation from live-action footage [WXSC04]. This kind of algorithm is an important open-research topic and a great project, but it is not discussed further in this chapter. (Courtesy of Jue Wang and Michael Cohen. Drawn and performed by Lean Joesch-Cohen. ©2004 ACM, Inc. Reprinted by permission.)

Finally, artists are essential! Even the best animation algorithms are ineffective without expressive input data, and the worst animation algorithms can succeed if controlled by a master animator. Those input data are created by artists employing animation tools that are themselves complex software. To build such tools one must appreciate the artists' goals and approach to animation. These tools shape the format of the data that then feed the runtime systems.

35.2 Motivating Examples

We begin with ad hoc methods for creating motion in some simple scenes. These illuminate the important issues of animation and suggest methods for generalizing to more formal methods.

35.2.1 A Walking Character (Key Poses)

Consider the case of creating an animation of a person walking. Let the person be modeled as a 3D mesh represented by a vertex array and an indexed triangle list as described in Chapter 14. Assume that an artist has already created several variations on this mesh that have identical index lists but potentially different vertex positions. These mesh variations represent different poses of the character during its walk. Each one is called a **key pose** or **key frame.** The terminology dates back to hand-animated cartoons, when a master animator would draw the key frames of animation and assistant animators performed the "tweening" process of computing the in-between frames. In 3D animation today, an **animator** is the artist who poses the mesh and an algorithm interpolates between them. Note that in many cases the animator is not the same person who initially created the mesh because those tasks require different skill sets. Later in the chapter we will address some of the methods that might be employed to create the poses efficiently. For now we'll assume that we have the data.

Although a real person might never strike exactly the same key pose twice, walking is a repetitive motion. We can make a common simplification and assume that there is a walk **cycle** that can be represented by repeating a finite number of discrete poses. For simplicity, assume that the key poses correspond to uniformly spaced times at $1/4$ second intervals, like the ones shown in Figure 35.6.

To play the animation we simply alter the mesh vertices according to the input data. Most displays refresh 60–85 times per second (Hz). Because the input is at 4 Hz, if we increment to the next key pose for every frame of animation, then the character will appear to be walking far too fast. For generality, let $p = 1/4$ s be the period between key poses. Let $\mathbf{x}(t) = \left[x(t), y(t), z(t)\right]^{\mathrm{T}}$ be the position of one vertex at time t. The input specifies this position only at $t = k/p$ for integer values

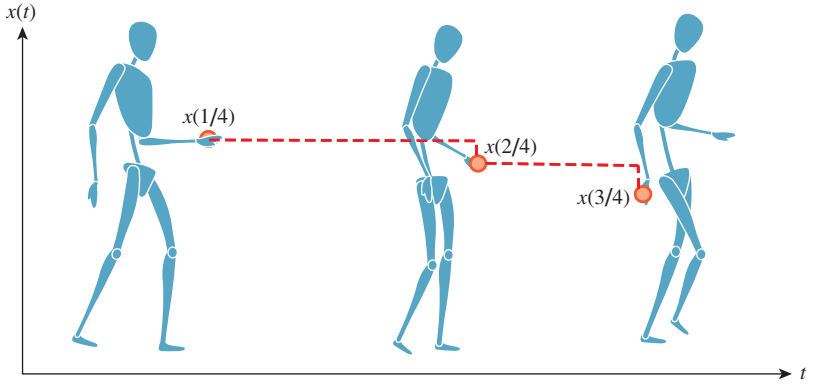


Figure 35.6: Sample-and-hold interpolation of position over time of a point on a character's hand.

of k; let those values be denoted $\mathbf{x}^*(t)$. Assume that the same processing will happen to all vertices; we'll return to ways of accomplishing that in a moment.

If we choose a **sample-and-hold** strategy for the intermediate frames:

$$\text{Let } t_0 = \lfloor t/p \rfloor p \tag{35.1}$$

$$\mathbf{x}(t) = \mathbf{x}^*(t_0), \tag{35.2}$$

then the animation will play back at the correct rate. The expression for t_0 in Equation 35.1 is a common idiom. It rounds t down to the nearest integer multiple of p.

Inline Exercise 35.1: Closely related to sample-and-hold is **nearest-neighbor,** where we round t to the nearest integer multiple of p.
(a) Write an expression for the nearest-neighbor strategy.
(b) Explain why sample-and-hold, given values at integer multiples of p, is the same as nearest-neighbor applied to the same values, each shifted by $p/2$. Because of this close relation, the terms are often informally treated as synonyms.

As shown in Figure 35.6, the result of sample-and-hold interpolation will not be smooth. At 60 Hz playback, we'll see a single pose hold for 15 frames and then the character will instantaneously jump to the next key pose. We can improve this by linearly interpolating between frames:

$$\text{Let } t_0 = \lfloor t/p \rfloor p; \;\; t_1 = t_0 + p; \;\; \alpha = (t - t0)/p \text{ and} \tag{35.3}$$

$$\mathbf{x}(t) = (1 - \alpha)\mathbf{x}^*(t_0) + \alpha\mathbf{x}^*(t_1). \tag{35.4}$$

The linear interpolation avoids the jumps between poses so that positions appear to change smoothly, as shown in Figure 35.7.

This discussion was for a single vertex. There are several methods for extending the derivation to multiple vertices; here are three. A straightforward extension is to apply equivalent processing to each element of an array of vertices. That is, to let $\mathbf{x}_i(t)$ be the position of the vertex with index i and then let

$$\mathbf{x}_i(t) = \begin{bmatrix} x_i(t) & y_i(t) & z_i(t) \end{bmatrix}^{\mathrm{T}}. \tag{35.5}$$

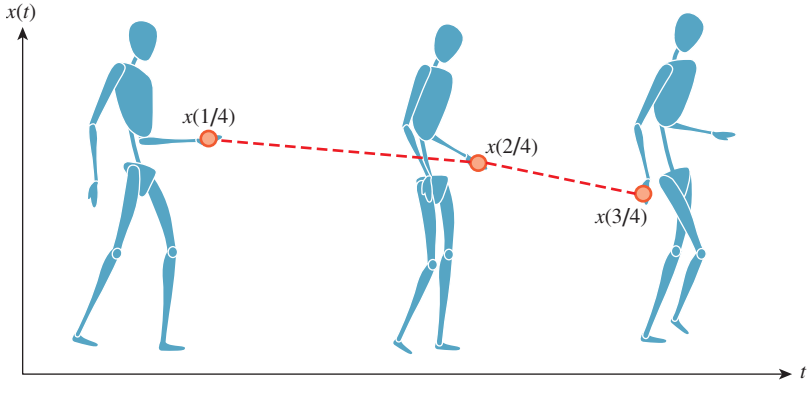


Figure 35.7: Linear interpolation of position over time of a point on a character's hand.

This array representation for the motion equations matches both the indexed trimesh representation and many real-time rendering APIs, so it is a natural way to approach mesh animation.

One alternative is to leave the interpolation equation (Equation 35.4) unmodified and instead redefine the input and output. For example, we could define a function $\mathbf{X}(t)$ describing the state of the system as a very long column comprising the positions of all vertices, instead of a 3-vector-valued function defining a single position:

$$\mathbf{X}(t) = \begin{bmatrix} x_0(t) & y_0(t) & z_0(t) & \dots & x_{n-1}(t) & y_{n-1}(t) & z_{n-1}(t) \end{bmatrix}^{\mathrm{T}}. \tag{35.6}$$

Note that nothing in our derivation depends on how components are arranged within the vector. Therefore, any two state vectors obeying the same convention can be linearly combined and the components will correspond along the appropriate axes. This representation works well when extending the linear interpolation to splines and, as shown later in this chapter in numerical integration schemes. The chosen interpolation algorithm will simply treat its input and output as a single very high-dimensional point, even though we consider it to be a concatenated series of mesh vertices.

Another alternative is to redefine the position function as a matrix,

$$\mathbf{X}(t) = \begin{bmatrix} x_0(t) & x_1(t) & \dots & x_{n-1} \\ y_0(t) & y_1(t) & \dots & y_{n-1} \\ z_0(t) & z_1(t) & \dots & z_{n-1} \\ 1 & 1 & \dots & 1 \end{bmatrix}. \tag{35.7}$$

This representation works well with the matrix representation of coordinate transformations because we can compute transformations of the form $\mathbf{M} \cdot \mathbf{X}(t)$, where $\mathbf{M}$ is a 4×4 matrix.

Each of these representations could be implemented with exactly the same layout in memory. The difference in choice of representation affects the theoretical tools we can bring to bear on animation problems and the practical interface to the software implementation of our algorithms. In practice, representations analogous to each of these have specific applications in animation for different tasks.

In closing, consider the state of our evolving interpolation algorithm. Although the vertex motion is continuous under linear interpolation, there are several remaining problems with this simple key pose interpolation strategy.

- The vertex motion has has C^0 continuity. This means that although positions change continuously, the magnitude of acceleration of the vertices is zero at times between poses, and infinite at the time of the key poses. Using a spline with C^1 or higher continuity can improve this (see Figure 35.8).
- The animation does not preserve volume. Consider key poses of a character and the character rotated 180° about an axis. Linear, or even spline, interpolation will cause the character to flatten to a line and then expand into the new pose, rather than turning.
- The walk cycle doesn't adapt to the underlying surface on which the character is walking. If the character walks up a hill or stairs, then the feet will either float or penetrate the ground.
- Animations are smooth within themselves, but the transitions between different animations will still be abrupt.

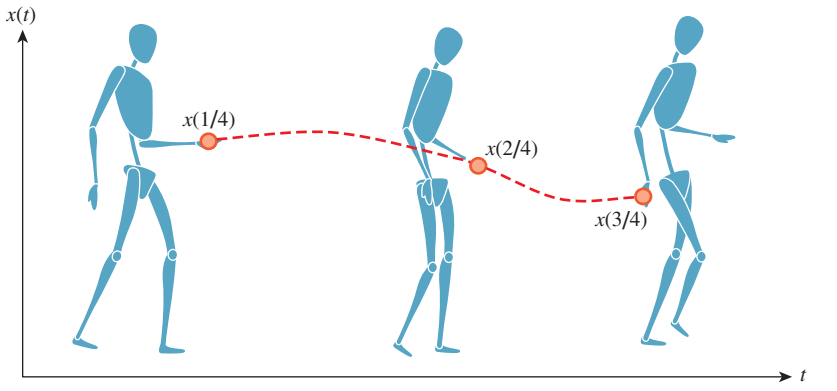


Figure 35.8: Piecewise-cubic interpolation of position over time of a point on a character's hand using a spline.

- We can't control different parts of the character independently. For example, we might want to make the arms swing while the legs walk.
- The animation scheme doesn't provide for interaction or high-level control. There is no notion of an arm or a leg, just a flat array of vertices.
- We still require a strategy for creating animation data, either by hand or as measurements of real-world examples.
- We've only considered animation that is tied to a specific mesh. Creating animation proves to be time-consuming by current practices, so it is desirable to transfer animation of one mesh to a new mesh representing a different character—doing so means abstracting the animation away from the vertices to higher-level primitives like limbs. (The mesh deformation transfer described in Section 25.6.1 is one technique for this.)

Inline Exercise 35.2: Create a simple animation data format and playback program to explore these ideas. Instead of a 3D character, limit yourself to a 2D stick figure. Use only four frames of animation for the walk cycle, and manually enter the key pose vertex positions in a text file. Making even four frames of animation will probably be challenging. Why? What kind of tool could you build to simplify the process? What aspects of the linear interpolation are dissatisfying during playback?

35.2.2 Firing a Cannon (Simulation)

Let's render an animation of a sailing ship firing a cannon as shown in Figure 35.9. The cannonball will be rendered as a black sphere, so we can ignore its orientation and focus only on the motion of its center of mass/local coordinate frame origin—the **root motion.** Neglecting the effects of drag due to wind and the slight variance in gravitational acceleration, the cannonball experiences only constant acceleration due to gravity after it is fired. A physics textbook gives an equation for the motion of an object under constant acceleration as

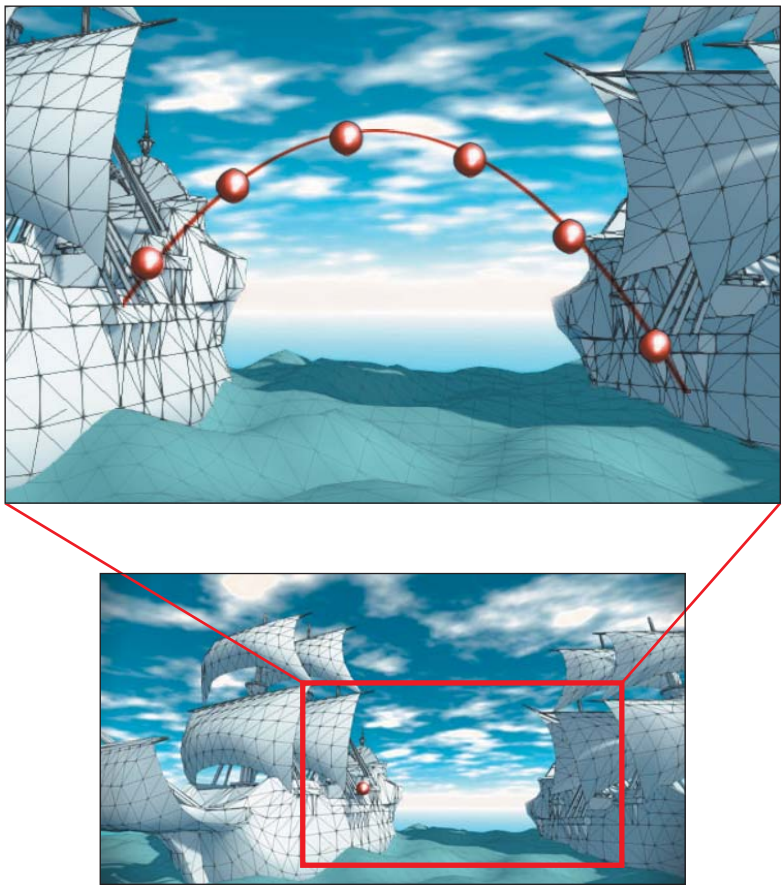


Figure 35.9: One frame (bottom) and a superimposed detail of a sequence of frames (top) of animation depicting the flight of a cannonball as computed by procedural physics.

$$\mathbf{x}(t) = \mathbf{x}_0 + \mathbf{v}_0 t + \frac{1}{2}\mathbf{a}_0 t^2, \tag{35.8}$$

where $\mathbf{x}(t)$ is the 3D position, t is time, $\mathbf{x}_0 = \mathbf{x}(0)$ is the initial position of the object, $\mathbf{v}_0$ is the initial velocity of the object, and $\mathbf{a}_0$ is the constant acceleration factor. In m-kg-s SI units, position is measured in meters, time in seconds, velocity in meters per second, and acceleration in meters per second squared. The same textbook gives the acceleration from the Earth's gravitational field as 9.81 m/s^2 (downward). We'll see where these equations and constants came from later in this chapter; for now, let's just trust the physics textbook. An 18th-century ship's cannon with an 11 kg ball has a 520 m/s muzzle velocity. Assuming that y is up and we are firing along the x-axis at an elevation of $\pi/4$ from the horizontal, the initial conditions are

$$\mathbf{v}_0 = (520\cos(\pi/4), 520\sin(\pi/4), 0) \text{ m/s}; \tag{35.9}$$

$$\mathbf{a}_0 = (0, -9.81, 0) \text{ m/s}^2 \tag{35.10}$$

To actually render an animation, we must produce many individual images separated by small steps in t (see Figure 35.9). When these are quickly viewed

consecutively, the viewer will no longer perceive the individual images and instead will see the cannonball moving smoothly through the air. The code to actually render a T-second animation of N individual frames looks something like Listing 35.1.

Listing 35.1: Immediate-mode rendering for a cannonball's flight.

```
float speed = 520.0f;
float angle = 0.7854f;

Vector3 x0 = ball.position();
Vector3 v0(cos(angle) * speed, sin(angle) * speed, 0.0f);
Vector3 a0(0, -9.8f, 0);

for (int i = 0; i < N; ++i) {
     float t = T * i / (N - 1.0f);
     ball.setPosition(x0 + v0 * t + 0.5 * a0 * t * t);

     clearScreen();
     render();
     swapBuffers();
}
```

The `swapBuffers` call is important. If we simply cleared the screen and drew the scene repeatedly the viewer would see a flickering as the image was built up with each frame. So we instead maintain two framebuffers: a static **front buffer** that displays the current frame to the viewer and a **back buffer** on which we are drawing the next frame. The `swapBuffers` call tells the rendering API that we are done rendering the next frame so that it can swap the contents of the buffers and show the frame to the viewer. This is called **double-buffered** rendering.

This was a simple example of procedural motion based on real-world dynamics. The steps here produce a satisfying animation but raise questions that we'll need to address in a more general framework for procedural motion and dynamics.

- Our equations don't take into account skipping the cannonball off the surface of the water, sinking it into the water, or crashing it into the targeted ship. How can we detect and respond to collisions and changing circumstances?
- We can solve for the flight time T based on the intersection of the parabolic arc and the other ship (or the water plane). But how should we choose the number of frames N to render in that time?
- What about objects that don't have constant acceleration from a single force such as gravity? For example, how do we move the ship itself through the water?
- For a featureless ball, we could ignore orientation. What are the equations of motion for an arbitrary tumbling object?
- Imagine procedural motion for an object not moving ballistically (e.g., a person dancing [PG96]). Deriving the equations of motion from first principles of physics would be really hard, and it would also be hard to ask an animator to specify the motion with explicit equations. What can we do?

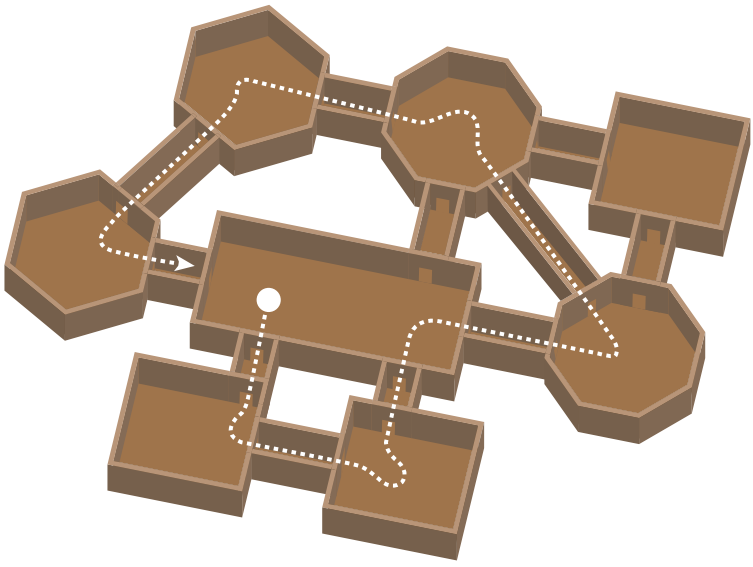


Figure 35.10: Selecting the key poses to navigate a character through these corridors is a problem at the interface between computer graphics and artificial intelligence (based on figure from [AVF04], which discusses AI-based methods for motion planning.).

35.2.3 Navigating Corridors (Motion Planning)

Consider the motion of a hovering robotic drone patrolling the interior of a building (Figure 35.10). We chose a hovering robot to avoid issues of rotating wheels or limbs, and a deforming mesh. The entire motion of the robot can be expressed as a change of the reference frame in which its rigid mesh is defined. This reference frame is called the **root frame** of the robot and the motion is called **root frame animation.** (This is a case where context determines the meaning of overloaded technical terminology for motion. The word "frame" in the previous sentence refers to a coordinate transformation, not an image; and "animation" refers to the true 3D motion, not a sequence of images.)

If we assume that the robot is a uniform sphere in order to ignore the problem of representing its rotational frame, the robot's animation is simply a formula for the translation of its root frame, $\mathbf{x}(t) = \ldots$. We could approach this problem as either key pose interpolation or a simulation. A hybrid strategy is often best for problems like this: Given an ideal path based on key poses, simulate the actual motion of the robot close to that path based on forces like gravity, the hover mechanism, and drag. This captures both the high-level motion and the character of real physics.

What is the source of the key poses? For a character's walk cycle, we were able to assume that an animator used some tool to create the key poses. To accurately model an autonomous robot, or to create an interactive application, we can't rely on an artist. The robot must choose its own key poses dynamically based on a goal, such as navigating to a specific room. This is an Artificial Intelligence (AI) problem. Real-world robots and video games with nonplayer characters solve it, generally using some form of path finding algorithm. Path finding has been long studied in computer science, primarily in the context of abstract graphs. For a 3D virtual world we must solve not only the graph problem but also the local problems that arise from actual room geometry and multiple interacting characters.

A common approach is to apply a traditional AI path finding algorithm like A* to create a root motion spline, and then use another greedy algorithm to look ahead a small time interval and avoid small-scale collisions.

So far, we have considered only translational root motion for navigation. The problem of synthesizing dynamic character motion becomes even more challenging when we must solve for the motion of limbs, coordinate multiple characters, or handle deformation. This general problem is called **motion planning,** and it is an active area of research in not only computer graphics but also AI and robotics. Most solutions draw on the principles and algorithms described in this chapter. However, they also tend to leverage search and machine learning strategies that require more AI background to describe. Having motivated it, we now leave the motion planning aside. The remainder of this chapter overviews basic primitives up to motion in the *absence* of AI, with emphasis on the motion of rigid primitives.

35.2.4 Notation

Variables in animation algorithms are qualified in many ways. Reference frames have three translational dimensions and three rotational dimensions, all quantities are functions of time, and most quantities are actually arrays to accommodate multiple objects or vertices. The algorithms we use also typically involve first and second time derivatives, and often consider the instantaneous value before and after an event such as a collision.

Animation-specific notations address these qualifications, but they differ from the notations predominant in rendering that are used elsewhere in this book. The following notation, which is common in the animation literature, applies only within this chapter (see Table 35.1).

Table 35.1: Formatting conventions for this chapter.

Symbol	Interpretation
$f(t)$	Value of a scalar function f of time at time t
$\mathbf{x}(t)$	Vector value of a function $\mathbf{x}$ at time t
$\dot{\mathbf{x}}(t)$	$\frac{d\mathbf{x}(t)}{dt}$
$\ddot{\mathbf{x}}(t)$	$\frac{d^2\mathbf{x}(t)}{dt^2}$
x_i	Element i of an ordered set of homogeneous elements
$\mathbf{x}[1]$	First element of a vector
$\mathbf{x}_i$	Element i of an array of vectors
$\dot{\mathbf{x}}^-$	Quantity immediately before an event (here, a single-sided derivative)
$\dot{\mathbf{x}}^+$	Quantity immediately after an event
$\mathbf{X}(t)$	Ideal state function for the entire system
$\mathbf{Y}_i \approx \mathbf{X}(i\Delta t + t_0)$	Actual state of the system at frame i resulting from a numerical integration scheme

Vectors are in boldface (e.g., $\mathbf{x}$), to leave space for other hat decorations. A dot over the $\mathbf{x}$ denotes that $\dot{\mathbf{x}}$ is the first derivative of $\mathbf{x}$ with respect to time. This is

a common notation in physics. In animation, time appears in all equations and is always denoted by t (except when we need temporary variables for integration), so all derivatives are with respect to t (e.g., $\dot{\mathbf{x}}(t) = \frac{d\mathbf{x}}{dt}(t)$). Multiple dots indicate higher-order derivatives (e.g., $\ddot{\mathbf{x}}(t) = \frac{d^2\mathbf{x}}{dt^2}(t)$).

There are two hazards in this dot notation. The first is that "$\dot{\mathbf{x}}$" is such a compact form that it is easy to drop the "(t)" and treat the value of a velocity expression as a variable instead of as the evaluation of a function. When taking derivatives of complex expressions such as momentum that are based on velocity, forgetting that velocity is a function of time can lead to errors if you forget to apply the chain rule. For example, let's look at a function of two arguments defined by $\mathbf{p}(m, \mathbf{v}) = \frac{1}{2}m\mathbf{v}^2$ (which happens to be the momentum equation). We can compute the partial derivatives, $\frac{\partial \mathbf{p}}{\partial m}(m, \mathbf{v}) = \frac{1}{2}\mathbf{v}^2$; $\frac{\partial \mathbf{p}}{\partial \mathbf{v}}(m, \mathbf{v}) = m\mathbf{v}$. What you might see in an animation paper is a description of this function where the author writes, "$\mathbf{p}(m, \dot{\mathbf{x}}) = \frac{1}{2}m\dot{\mathbf{x}}^2$." And you'll even see things like "$\frac{\partial \mathbf{p}}{\partial \dot{\mathbf{x}}}(m, \dot{\mathbf{x}}) = m\dot{\mathbf{x}}$." Here, $\dot{\mathbf{x}}$ is being treated as a variable, just like $\mathbf{v}$, and it is almost reasonable to do so thus far. It is also common to see "$\frac{d}{dt}\mathbf{p}(m, \dot{\mathbf{x}}) = m\dot{\mathbf{x}}\ddot{\mathbf{x}}$." This is a little strange because t isn't even one of the arguments of function $\mathbf{p}$, and $\dot{\mathbf{x}}$ has changed from its role as a variable whose symbol happens to have a dot hat to representing a function of time whose time derivative is denoted $\ddot{\mathbf{x}}$. Thus, for clarity, in this chapter when we want such a function, we write either $\mathbf{p}(m, t) = \frac{1}{2}m\dot{\mathbf{x}}(t)^2$, or more verbosely,

$$\mathbf{p}^*(m, \mathbf{v}) = \frac{1}{2}m\mathbf{v}^2, \tag{35.11}$$

$$\mathbf{p}(m, t) = \mathbf{p}^*(m, \dot{\mathbf{x}}(t)) = \frac{1}{2}m\dot{\mathbf{x}}(t)^2, \text{ and} \tag{35.12}$$

$$\frac{\partial \mathbf{p}(m, t)}{\partial t} = m\dot{\mathbf{x}}(t)\ddot{\mathbf{x}}(t). \tag{35.13}$$

The second notational hazard is that the implementation of a dynamics system often uses higher-order functions. That is, it contains functions that take other functions as their arguments. In programming, the argument functions are called first-class functions or function pointers. There is a real distinction between the vector-valued function $\mathbf{x}$ (i.e., `Vector3 position(float time) ...`) and the vector value of that function at time t, $\mathbf{x}(t)$ (i.e., `Vector3 currentPosition;`). Passing the wrong one as an argument will lead to programming errors. We therefore always keep the derivatives in function notation for this chapter. However, be warned that in the animation literature it is commonplace to move between the variable and function notation.

Here's one critical example of this notation. When discussing numerical integration schemes that dominate the dynamics portion of this chapter, we distinguish three fundamental representations. The position of an object (which may represent *only* position, or may be extended with other information) is $\mathbf{x}(t)$, which is a vector-valued function. The elements of the vector may be, for example, x-, y-, and z-coordinates, or those coordinates and rotational (and other pose) information. Since there may be many objects in a system, or many points on a single object, we consider a set of functions. When evaluated at t, their values are denoted $\mathbf{x}_1(t)$, $\mathbf{x}_2(t)$, etc. The state function of the entire system, which comprises the entire set of position functions and their derivatives, is written $\mathbf{X}(t)$ when evaluated at time t. The state function is defined for continuous t.

When we approximate the state function with a numerical integrator that takes discrete steps, we refer to the values of the state function at given step indices.

This is $\mathbf{Y}_i \approx \mathbf{X}(i\Delta t + t_0)$. Note that $\mathbf{Y}_i$ for a given *i is not a function*—it is a value, which is typically represented as an array of (3D) vectors in a program. One could alternatively think of a function on a discrete time domain whose value at step *i* is $\mathbf{Y}[i]$. However, we do not use that notation for two reasons. First, we reserve the bracket notation for referencing the elements of a vector value. Second, a typical implementation only contains one $\mathbf{Y}$ value at a time. It would be misleading to think of it as a discrete function or array of values because only a single one is present in memory at a time. A good mental model for $\mathbf{Y}_i$ is the *i*th element of a sequence that the program is iterating through, which is what that notation is intended to suggest.

35.2.4.1 Notation in the Big Picture

For what it is worth, the first, and perhaps one of the most significant, challenges in learning the mathematics of animation is simply grappling with the notation. We've tried to choose a relatively simple and consistent notation for this chapter, but we acknowledge that it is still a ridiculously large new language to learn for something as seemingly simple as expressing Newton's laws.

The notation is complex because it bears the burden of expressing the many shades of "position" and values derived from it in a formal computational system. The gist of those shades and derivations embodies the fundamental rules, and thus the power, of animation. The gist is what is important because in a virtual world, the specifics of the laws of physics are arbitrary and mutable. On the computational side, there are a large number of ways to integrate and interpolate values. There is no single best solution (although there are some solutions that are always inferior).

As a concrete example, understanding the inputs and outputs of an integrator is actually more important than understanding the integration algorithm itself. There are many integration algorithms, but they all fit into the same integration systems that dictate the data flow. So the notation really is telling you something important. It therefore is worth taking the time to ensure that you understand the distinction between each x and $\mathbf{x}$ in an equation.

35.3 Considerations for Rendering

We now consider several ways in which animation and rendering are interrelated, ranging from display techniques like double and triple buffering to make animation appear smoother, all the way to approaches for generating motion blur to help smooth the appearance of moving objects.

35.3.1 Double Buffering

When a display can refresh faster than the processor can render a frame, drawing directly to the display buffer would reveal the incomplete scene. Because this is generally undesirable, **double-buffered** rendering draws to an off-screen **back buffer** while the **front buffer** containing the previous frame is displayed to the viewer. When the back buffer is complete, the buffers are "swapped," either by copying the contents of the back to the front or by moving the display's pointer between the buffers. The cannonball code in Listing 35.1 showed an example of how an explicit call to swap buffers can manage double-buffered

rendering. Double-buffered rendering of course doubles the size of the framebuffer in memory.

Analog vector scope (oscilloscope) displays have no buffer—a beam driven by analog deflection traces true lines on the display surface. So double-buffered rendering is impossible for such displays and they inherently reveal the scene as it is rendered. Vector scopes are rarely used today, although some special-purpose laser-projector displays operate on the same principle and have the same drawbacks.

For displays driven by a digital framebuffer, the swap operation must be handled carefully. A CRT traces through the rasters with a single beam. Pixels in the framebuffer that are written to will not be updated until the beam sweeps back across the corresponding display pixel. LCD, plasma, and other modern flat-screen technologies are capable of updating all pixels simultaneously, although to save cost, a specific display may not contain independent control signals for each pixel. Regardless, the display is typically fed by a serial signal obtained by scanning across the framebuffer in much the same way as a CRT. The result is that for most modern displays the buffers must be swapped between scanning passes. Otherwise, the top and bottom of the screen will show different frames, leading to an artifact called **screen tearing.** The raster at which the screen is divided will scroll upward or downward depending on the ratio of refresh rate to animation rate.

The solution to tearing is **vertical synchronization,** which simply means waiting for the refresh to swap buffers. The drawback of this is that it may stall the rendering processor for up to the display refresh period. Two common solutions are disabling vertical synchronization (which entails simply accepting the resultant tearing artifacts) and triple buffering. Under **triple buffering,** three framebuffers are maintained as a circular queue. This allows the renderer to advance to the next frame at a time independent of the display refresh. Of course, the renderer must still be updating at about the same rate as the display or the queue will fill or become empty, stalling either the display or the renderer. The queue may be implemented in a straightforward manner as an array of framebuffers that are each used in sequence, or as two double-buffer pairs that share a front buffer. The drawbacks of triple-buffered rendering are further increased framebuffer storage cost and an additional frame of latency between user input and display.

35.3.2 Motion Perception

Motion perception is an amazing property of the human visual system that enables all computer graphics animation. Current biological models of the visual system indicate that there is no equivalent of a refresh rate or uniform shutter in the eye or brain. Yet we perceive the objects in sequential frames shown at appropriate rates as moving smoothly, rather than warping between discrete locations or as separate objects that appear and disappear.

Motion phenomena are believed to occur mostly within the brain. However, the retina does exhibit a tendency to maintain positive afterimages for a few milliseconds and this may interact with the brain's processing of motion. These are distinct from the negative afterimages that occur when staring at a strong stimulus for a long period of time.

One effect of positive afterimages is that the human visual system is only strongly sensitive to flickering due to shuttering or image changes up to about

25 Hz, and is completely insensitive to flickering at frequencies higher than 80 Hz. This is partly why the 50 Hz (Europe) or 60 Hz (U.S.) flickering of a fluorescent light is just barely perceptible, and is why most computer displays refresh at 60 to 85 Hz.

The **Beta phenomenon** [Wer61], sometimes casually referred to by the overloaded term "persistence of vision," is the phenomenon that allows the brain to perceive object movement. At around 10 Hz, the threshold of motion perception is crossed and overlapping objects in sequential frames may appear to be a single object that is in motion. This is only the *minimum* rate for motion perception. If the 2D shapes are not suitably overlapped they will still appear as separate objects. This means that fast-moving objects in screen space require higher frame rates for adequate presentation.

The combination of fast-moving objects and the limitations of afterimages to conceal flickering cause a phenomenon called **strobing.** Here, motion perception breaks down even at frame rates higher than 30 Hz. A classic example of strobing is a filmed or rendered roller-coaster ride from a first-person perspective. Because points on the coaster track can traverse a significant portion of the screen, even at high frame rates the individual frames may appear as actual separate flashing images and not blend into perceived motion of an object. This can be a disturbing artifact that causes nausea or headache if prolonged. Above 80 Hz, afterimages appear to completely conceal the strobing effect and the motion becomes apparent, although the actual images may be blurred by the visual system.

The human perception of motion and flickering creates a natural range for viable animation rates, from about 10 Hz to about 80 Hz. Table 35.2 shows that various solutions are currently in use throughout that viable range.

Table 35.2: Common image display rates.

Frequency	Phenomenon or Technology
10 Hz	Approximate threshold of human motion perception
24 Hz	U.S. film
25 Hz	Approximate point where afterimages begin to conceal flickering
25 Hz	PAL film (Europe)
25 Hz	PAL television and video (progressive rate)
29.92 Hz	NTSC television and video (progressive rate)
50 Hz	PAL television and video (interleaved rate)
59.94 Hz	NTSC television and video (interleaved rate)
60 Hz	U.S. power duty cycle; fluorescent light flicker rate
65 Hz	Typical LCD monitor refresh rate
72 Hz	U.S. film projector refresh (24 Hz with each frame shown three times)
80 Hz	Approximate limit of human flickering perception; strobing ceases
85 Hz	Typical CRT monitor refresh rate
120 Hz	Stereo-vision LCD monitor refresh (both views)

We note that for dynamics, the rendering rate may be independent of the simulation rate. Simulating at low rates and then interpolating between simulation steps when rendering amortizes the cost of a simulation step. Simulating at high rates and then subsampling simulation steps when rendering can increase accuracy and stability. We return to these issues later in this chapter.

Most LCD monitors refresh at around 60–65 Hz, although some displays refresh at 120 or 240 Hz for shuttered stereo viewing of 60 Hz or 120 Hz images.

Note that there are two standards for film: 24 Hz in the United States and Japan and 25 Hz for the PAL/SECAM standard used in Europe and the rest of Asia. Interestingly, projected films are typically not adapted to the display's frame rate when moving between PAL and NTSC standards. Thus, 25 Hz European films are screened in the United States at 24 Hz. The audio makes the corresponding 4% speed change to remain in synchrony. The net result is that European films appear slow and low-pitched while American films appear fast and high-pitched, when viewed in the opposite projection mode.

35.3.3 Interlacing

Many television broadcast and storage formats are **interlaced.** In an interlaced format, each frame contains the full horizontal resolution of the final image but only half the vertical resolution. These half-resolution frames are called **fields.** Even and odd fields are offset by one pixel (or historically, one scan line). To display a complete image, two sequential fields must be combined by interlacing their **rasters** (rows of pixels).

Because each image merges pixels from different time slices, no single image is consistent. However, fast motion can be represented at comparatively low bandwidth. The artifacts of interlacing were historically hidden by the decay time of CRT phosphors, which took longer to change intensity than the frame period. Some contemporary displays can change images rapidly and thus reveal the interlacing pattern. This can be observed when pausing playback on an LCD monitor, although some displays attempt to interpolate between adjacent frames to reconstruct a progressive signal from an interlaced one.

At the time of this writing, most broadcast television and archived television shows remain in interlaced formats. With the advent of high-definition digital displays, new content is increasingly moving to **progressive** formats. Progressive is what you would expect: Each frame contains a complete image.

The progressive rate for PAL/SECAM television is 25 Hz and the interlaced rate is 50 Hz. This means that normal European television broadcasts send one field every $1/50$ s such that every $1/25$ s a complete image has been transmitted.

Console games displayed on televisions may elect to render in either progressive or interlaced formats. The advantage of an interlaced format is that there are half as many pixels to render per frame, yet the viewer rarely perceives a 50% reduction in quality.

35.3.3.1 Telecine

The PAL/SECAM formats allow European films to be broadcast unmodified on European television because they both are driven at 25 Hz. To interlace such a film, simply drop half the rasters each frame.

In the United States, the process is not as simple. NTSC television requires approximately 30 Hz, but U.S. films are at 24 Hz. These only align once every six

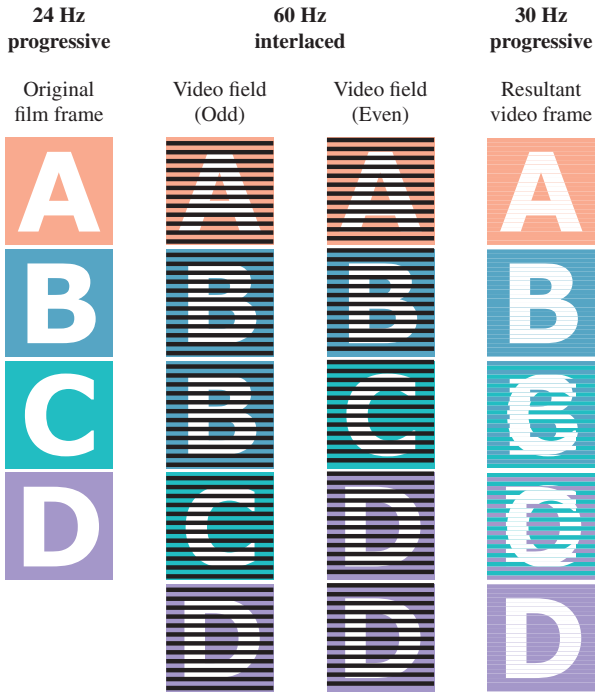


Figure 35.11: Schematic of how interlacing is exploited to adapt 24 Hz film frames for 60 Hz interlaced broadcast to NTSC televisions under the 3:2 pulldown algorithm. In the center columns, the odd and even source film frames have been repeated three and two times, respectively. (Created by Eric Lee.)

frames, and interpolating the remaining frames from adjacent ones would significantly blur the images. The **telecine** or **pulldown** employed in practice is a clever alternative that exploits interlacing.

The commonly employed algorithm is called 3:2 pulldown. We begin with a simplified example to understand the intuition behind it. Instead of resampling from 24 Hz progressive to 30 Hz progressive, consider the problem of resampling from 24 Hz progressive to 60 Hz interlaced. We can approach this by replicating each source frame "2.5" times, that is, by repeating frame i twice, blending frames i and $i + 1$, and then proceeding to process frame $i + 1$. This blends only one out of every three frames, which is substantially better than blending five out of every six frames. The output will be progressive, so to create an interlaced format, drop half the rasters from every frame.

What the actual **3:2 pulldown** algorithm does is perform the blending by choosing the source rasters more selectively. This avoids blending pixels from separate frames and directly produces interlaced output. This is similar to stochastic blending methods like dithering: The blending is spatial and is integrated by the eye. Figure 35.11 shows the process. Given four original film frames A, B, C, and D sampled at 24 Hz (left column), the algorithm will produce $4 \cdot 2.5 = 10$ interlaced frames at $\approx$60 Hz (center columns), that correspond to five progressive frames at $\approx$30 Hz (right column). Note that the interlaced frames are broadcast in alternating raster order, so the top frame of the left-center column will be

broadcast first, followed by the top frame of the right-center column, followed by the second-to-top row of the left-center column, etc.

The interlaced frames are chosen by repeating even frames from the original film twice and odd frames from the film three times; that is, odd to even appear in the ratio 3:2. Notice how in the center columns the even source frames *A* and *C* appear twice and the odd source frames *B* and *D* appear three times.

35.3.4 Temporal Aliasing and Motion Blur

Rendering a frame from a single instant in time is convenient because all geometry can be considered static for the duration of the frame. Each pixel value in an image represents an integral over a small amount of the image plane in space and a small amount of time called the **shutter time** or **exposure time.** Film cameras contained a physical shutter that flipped or irised open for the exposure time. Digital cameras typically have an electronic shutter. For a static scene, the measured energy will be proportional to the exposure time. A virtual camera with zero exposure can be thought of as computing the limit of the image as the exposure time approaches zero.

There are reasons to favor both long and short exposure times in real cameras. In a real camera, short exposure times lead to noise. For moderately short exposure times (say, $1/100$ s) under indoor lighting, background noise on the sensor may become significant compared to the measured signal. For extremely short exposure times (say, $1/10{,}000$ s), there also may not be enough photons incident on each pixel to smooth out the result. Nature itself uses discrete sampling because photons are quantized. In computer graphics we typically consider the "steady state" of a system under large numbers of photons, but this model breaks down for very short measurement intervals. A long exposure avoids these noise problems but leads to blur. For a dynamic scene or camera, the incident radiance function is not constant on the image plane during the exposure time. The resultant image integrates the varying radiance values, which manifest as objects blurring proportional to their image space velocity. Small camera rotations due to a shaky hand-held camera result in an entirely blurry image, which is undesirable. Likewise, if the screen space velocity of the subject is nonzero, the subject will appear blurry. This **motion blur** can be a desirable effect, however. It conveys speed. A very short exposure of a moving car is identical to that of a still car, so the observer cannot judge the car's velocity. For a long exposure, the blur of the car indicates its velocity. If the car is the subject of the image, the photographer might choose to rotate the camera to limit the car to zero screen-space velocity. This blurs the background but keeps the car sharp, thus maintaining both a sharp subject and the velocity cue.

For rendering, our photons are virtual and there is no background noise, so a short exposure does not produce the same problems as with a real camera. However, just as taking only one spatial sample per pixel results in aliasing, so does taking only one temporal sample. The top row of Figure 35.12 shows two images of a very thin row of bars, as on a cage. If we take only one spatial sample per pixel, say, at the pixel center, then for some subpixel camera offsets the bars are visible and for others they are invisible. Note that as the spatial sampling density increases, the bars can be resolved at any position. The bottom row shows the result of the equivalent experiment performed for temporal samples. A fast-moving car is driving past the camera in the scene depicted. For a single temporal

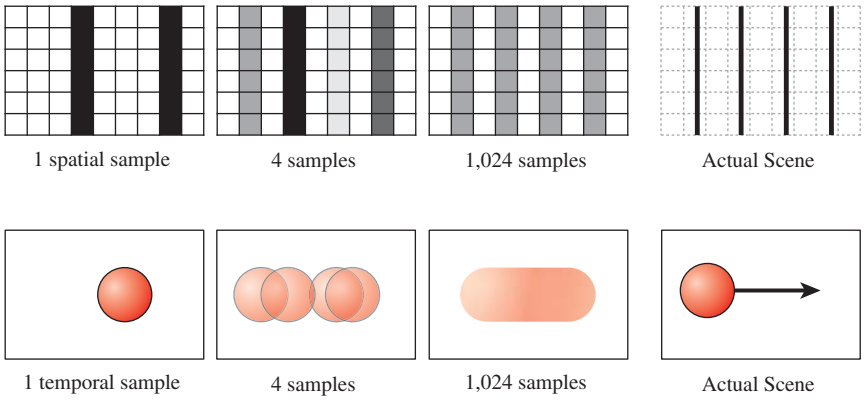


Figure 35.12: Top row: a fence made of black posts on a white background imaged with increasing spatial resolution. Bottom row: a moving sphere imaged with increasing temporal resolution. Increasing the number of samples better captures the underlying scene, in space or time. Here a regular sampling pattern is used for each.

sample, the car may be either present or absent. Increasing the temporal sampling rate increases the ability to resolve the car. For a high temporal sampling rate the image approaches that which would be captured by a real camera, where the car is blurred across the entire frame.

One method for ameliorating temporal aliasing is to use a high-refresh rate display and render once per refresh. Although not common today, there are production 240 Hz displays. Simply rendering at 240 Hz provides four times the temporal sampling rate of the common 60 Hz rendering rate. This does not solve the temporal aliasing problem. It merely reduces its impact. A sufficiently fast car, for example, will still flash into the center of the screen and disappear.

A more common alternative to a high refresh rate solves the flashing problem and does not require a special display. One can explicitly integrate many temporal samples, producing rendered motion blur. Distribution[1] ray tracing [CPC84] pioneered this approach, which has since been extended to rasterization. Here, software is performing the integration that was performed by the eye under a high-refresh display.

Integration over temporal samples does not necessarily give the same perception as observing a high refresh display or the real world, however. The reason is that the eye is not a camera. In the absence of temporal integration, the observer's eye can track the motion of an object in the scene at a high rate. For example, the eye can rotate to keep an object moving across the display's field of view at the same location in the eye's field of view. The resultant perception is that the moving object is sharp and the background is blurred. If the same scene is shown at a low frame rate that has been integrated over multiple temporal samples, the moving object will be blurred and the background will be sharp. This might lead one to the conclusion that motion blur cannot be rendered effectively without eye

1. Cook et al. originally called their technique "distributed" ray tracing because it distributes samples across the sampling domain, including time. Today it is commonly called "distribution" ray tracing to distinguish it from processing distributed across multiple computers. It is also called "stochastic" ray tracing since it is often implemented using stochastic sampling, although technically, the decision to distribute samples (especially eye-ray samples) is separate from the choice of sampling pattern.

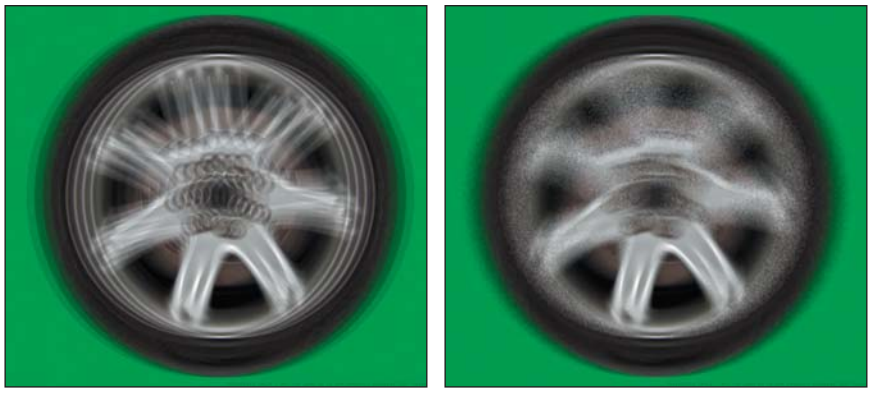


Figure 35.13: Undersampling in time with regular (a.k.a. uniform, statistically dependent) samples within each pixel (left) produces ghosting. Undersampling with stochastic (a.k.a. independent, random) samples produces noise (right) [AMMH07]. (Courtesy of Jacob Munkberg and Tomas Akenine-Möller)

tracking. However, all live-action film faces exactly this problem and rarely do viewers experience disorientation at the fact that the images have been preintegrated over time for their eyes. If the director does a good job of directing the viewer's attention, the camera will be tracking the object of primary interest in the same way that the eye would. Presumably, a poorly directed film fails at this and creates some disorientation, although we are not aware of a specific scientific study of this effect. Similar problems arise with defocus due to limited depth of field and with stereoscopic 3D. For interactive 3D rendering all three effects present a larger challenge because it is hard to control or predict attention in an interactive world. Yet in recent years, games in particular have begun to experiment with these effects and achieve some success even in the absence of eye tracking.

Antialiasing, motion blur, and defocus are all cases of integrating over a larger sampling area than a single point to produce synthetic images that more closely resemble those captured by a real camera. Many rendering algorithms combine these into a "5D" renderer, where the five dimensions of integration are subpixel x, y, time, and lens u, v. Cook et al.'s original distribution and stochastic ray tracing schemes [CPC84, Coo86] can be extended to statistically dependent temporal samples per pixel by simply rendering multiple frames and then averaging the results. Because all samples in each frame are at the same time, for large motions this produces discrete "ghosts" for fast-moving objects instead of noisy ghosts, as shown in Figure 35.13. Neither of these is ideal—the image has been undersampled in time and each is a form of aliasing. The advantage of averaging multiple single-time frames is that any renderer, including a rasterization renderer, can be trivially extended to simulate motion blur in this method.

The Reyes micropolygon rendering algorithm [CCC87] that has been heavily used for film rendering is a kind of stochastic rasterizer. It takes multiple temporal samples during rasterization to produce effects like motion blur, avoiding the problem of dependent time samples. Akenine-Möller et al. [AMMH07] introduced explicit temporal stochastic rasterization for triangles, and Fatahalian

et al. [FLB+09] combined that idea with micropolygons for full 5D micropolygon rasterization. Fatahalian et al. framed rasterization as a five-dimensional point-in-polyhedron problem and solved it in a data-parallel fashion for efficient execution on dedicated graphics hardware.

Because integration over multiple samples is expensive, a variety of tricks that generate phenomena similar to motion blur have been proposed. These have historically been favored by the game industry because they are fast, if sometimes poor-quality, approximations. See Sung et al. [SPW02] for a good survey of these. The major methods employed are adding translucent geometry that stretches an object along its screen-space velocity vector, artificially increasing the MIP level chosen from textures to blur within an object, and screen-space blurring based on per-pixel velocity as a post-process [Vla08].

Renderers paradoxically spend more time producing blurry phenomena such as motion blur than they do in imaging sharp objects. This is because blurring requires either multiple samples or additional post-processing. Since it is harder for viewers to notice artifacts in blurry areas of an image, it would be ideal to somehow extrapolate the blurry result from fewer samples. This is currently an active area of research [RS09, SSD+09, ETH+09].

35.3.5 Exploiting Temporal Coherence

An animation contains multiple frames, so rendering animation is necessarily more computationally intense than rendering a single image. However, the cost of rendering an animation is not necessarily proportional to its length. Sequential frames often depict similar geometry and lighting viewed through similar cameras. This property is referred to as **frame coherence** or **temporal coherence.** It may hold for the underlying scene state, the rendered image, both, or neither.

One advantage of frame coherence is that one can often reuse intermediate results from rendering one frame for the subsequent frame. Thus, the first frame of animation is likely as expensive to render as a single image, but subsequent frames may be comparatively inexpensive to render.

For example, it is common practice in modern rasterization renderers to only recompute the shadow map associated with a luminaire only when both the volume illuminated by that luminaire intersects the view frustum and something within that volume moved since the previous computation. Historically, 2D renderers were not fast enough to update the entire screen when drawing user interfaces. They exploited frame coherence to provide a responsive interface. Such systems maintained a persistent image of the screen and a list of 2D bounding boxes for areas that required updating within that image. These bounding boxes were called **dirty rectangles.** Although modern graphics processors are fast enough to render the entire screen every frame, the notion of dirty rectangles and its generalization to **dirty bit flags** remains a core one for incremental updates to computer graphics data structures.

The process of storing intermediate results for later reuse is generally called **memoization;** it is also a main component of the dynamic programming technique. If we allow only a fixed-size buffer for storing previous results and have a replacement strategy when that buffer is full, the process is called **caching.** Reuse necessarily requires some small overhead to see if the desired result has already

been computed. When the desired result is not available (or is out of date, as in the dirty rectangle case) the algorithm has already invested the time for checking the data structure and must now pay the additional time cost of computing the desired result and storing it. Storing results of course increases the space cost of an algorithm. Thus, reuse strategies can actually increase the total cost of rendering when an animation fails to exhibit frame coherence.

When reuse produces a net time savings, it is reducing the *amortized* cost of rendering each frame. The worst-case time may still be very high. This is problematic in interactive applications, where inconsistent frame intervals can break the sense of immersion and generally hampers continuous interaction. One solution is to simply terminate rendering a frame early when this occurs. For example, the set of materials (i.e., textures) used in a scene typically changes very little between frames, so the cost of loading them is amortized over many frames. Occasionally, the material set radically changes. This happens, for example, when the camera crests a hill and a valley is revealed. One can design a renderer that simply renders parts of the scene for which materials are not yet available with some default or low-resolution material. This will later cause a visual pop (violating coherence of the final image) when these materials are replaced with the correct ones, but it maintains the frame rate. The worst case is often the first frame of animation, for which there is no previous frame with which to exhibit coherence.

35.3.6 The Problem of the First Frame

The first frame of animation is typically the most expensive. It may cost orders of magnitude more time to render than subsequent frames because the system needs to load all of the geometry and material data, which may be on disk or across a network connection. Shaders have not yet been dynamically compiled by the GPU driver, and all of the hardware caches are empty. Most significantly, the initial "steady state" lighting and physics solution for a scene is often very expensive to compute compared to later incremental updates.

Today's video games render most frames at $1/30$ s or $1/60$ s intervals. Yet the *first* frame might take about one minute to render. This time is often concealed by loading data on a separate thread in the background while displaying a loading screen, prerendered cinematic, or menus. Some applications continuously stream data from disk.

If we consider the cost of precomputed lighting, some games take *hours* to render the first frame. This is because computing the global illumination solution is very expensive. The result is stored on disk and global illumination is then approximated for subsequent frames by the simple strategy of assuming it did not change significantly. Although games increasingly use some form of dynamic global illumination approximation, this kind of precomputation for priming the memoization structure is a common technique throughout computer graphics.

Offline rendering for film is typically limited by exactly these "first frame" problems. Render farms for films typically assign individual frames to different computers, which breaks coherence on each computer. Unlike interactive applications, at render time films have prescripted motion, so the "working set" for a frame contains only elements that directly affect that frame. This means that even within a single shot, the working set may exhibit much less coherence than for an interactive application. Because of this system architecture and pipeline, a film

renderer is in effect always rendering the first frame, and most film rendering is limited by the cost of fetching assets across a network and computing intermediate results that likely vary little from those computed on adjacent nodes.

35.3.7 The Burden of Temporal Coherence

When rendering an animation where the frames should exhibit temporal coherence, an algorithm has the burden of maintaining that coherence. This burden is unique to animation and arises from human perception.

The human visual system is very sensitive to change. This applies not only to spatial changes such as edges, but also to temporal changes as flicker or motion. Artifacts that contribute little perceptual error to a single image can create large perceptual error in an animation if they create a perception of false motion. Four examples are "popping" at level-of-detail changes for geometry and texture, "swimming jaggies" at polygon edges, dynamic or screen-door high-frequency noise, and distracting motion of brushstrokes in nonphotorealistic rendering,

Popping occurs when a surface transitions between detail levels. Because immediately before and immediately after a level-of-detail change either level would produce a reasonable image, the still frames can look good individually but may break temporal coherence when viewed sequentially. For geometry, blending between the detail levels by screen-space compositing, subdivision surface methods (see Chapter 23), or vertex animation can help to conceal the transition. Blending the final image ensures that the final result is actually coherent, whereas even smoothly blending geometry can cause lighting and shadows to still change too rapidly. However, blending geometry guarantees the existence of a true surface at every frame. Image compositing results in an ambiguous depth buffer or surface for global illumination purposes. For materials, trilinear interpolation (see Chapter 20) is the standard approach. This generates continuous transitions and allows tuning for either aliasing (blurring) or noise. A drawback of trilinear interpolation is that it is not appropriate for many expressions, for example, unit surface normals.

Sampling a single ray per pixel produces staircase "jaggies" along the edges of polygons. These are unattractive in a still image, but they are worse in animation where they lead to a false perception of motion along the edge. The solution here is simple: antialiasing, either by taking multiple samples per pixel or through an analytic measure of pixel coverage.

High-frequency, low-intensity noise is rarely objectionable in still images. This property underlies the success of half-toning and dithering approaches to increasing the precision of a fixed color gamut. However, if a static scene is rendered with noise patterns that change in each frame, the noise appears as static swimming over the surfaces in the scene and is highly objectionable.

The problem of dynamic noise patterns arises from any stochastic sampling algorithm. In addition to dithering, other common algorithms that are susceptible to problems here include jittered primary rays in a ray tracer and photons in a photon mapper. Three ways to avoid this kind of artifact are making the sampling pattern static, using a hash function, and slowly adjusting the previous frame's samples.

Supersampling techniques for antialiasing often rely on the static pattern approach. This can be accomplished by **stamping** a specific pattern in screen space. There has been significant research into which patterns to use [GS89,

Coo86, Cro77, Mit87, Mit96, KCODL06, Bri07, dGBOD12]. This work is closely tied to research on white noise random number generation [dGBOD12].

One drawback to using screen-space patterns is the **screen door effect** (Chapter 34). A static pseudorandom pattern in screen space is not perceptible in a single image, but when the pattern is held fixed in screen space and the view or scene is dynamic, that pattern becomes perceptible. It looks as if the scene were being viewed through a screen door. This is easy to see in the real world. Hold your head still and look through a window (or slightly dirty eyeglasses). The glass is largely invisible, but on moving your head imperfections in and dirt on the glass are accentuated. This is because your visual system is trying to enforce temporal coherence on the objects seen through the glass. Their appearance is changing in time because of the imperfections in the glass in front of them, so you are able to perceive those imperfections.

Fortunately, when the sampling pattern resolution falls below the resolution of visual acuity, the perception of the screen-door effect is minimal. This is exploited by supersampling and alpha-to-coverage transparency, which operate below the pixel scale and are therefore inherently close to the smallest discernible feature size. Dithering works well when the image is static or the pixels are so small as to be invisible, but it produces a screen-door effect for animations rendered to a display with large pixels.

It is challenging to stamp patterns in continuous spaces; for example, a ray tracer or photon mapper's global illumination scattering samples. Here, replacing pseudorandom sampling with sampling based on a hash of the sample location is more appropriate. By their very nature, hash functions tend to map nearby inputs to disparate outputs, so this only maintains coherence for static scenes with dynamic cameras. For dynamic scenes, a spatial noise function is preferable [Per85] because it is itself spatially coherent, yet pseudorandom.

Another approach to increasing temporal coherence of sample points is to begin with an arbitrary sample set and then move the samples forward in time, adding and removing samples as necessary. This approach is employed frequently for nonphotorealistic rendering. The Dynamic Canvas [CTP+03] algorithm (Figure 35.14) renders the background paper texture for 3D animations rendered in the style of natural media. A still frame under this algorithm appears to be, for example, a hand-drawn 3D sketch on drawing paper. As the viewer moves forward, the paper texture scales away from the center to avoid the screen-door effect and give a sense of 3D motion. As the viewer rotates, the paper texture translates. The algorithm overlays multiple frequencies of the same texture to allow for infinite zoom and solves an optimization problem for the best 2D transformation to mimic arbitrary 3D motion. The initial 2D transformation is arbitrary, and at any point in the animation the transformation is determined by the history of the viewer's motion, not the absolute position of the viewer in the scene.

Another example of moving samples is brushstroke coherence, of the style originally introduced for **graftals** [MMK+00] (see Figure 35.15). Graftals are scene-graph elements corresponding to strokes or collections of strokes for small-detail objects, such as tree leaves or brick outlines. A scene is initially rendered with some random sampling of graftals; for example, leaves at the silhouettes of trees. Subsequent frames reuse the same graftal set. When a graftal has moved too far from the desired distribution due to viewer or object motion, it is replaced with a newly sampled graftal. For example, as the viewer orbits a tree, graftals moving toward the center of the tree in the image are replaced with new graftals

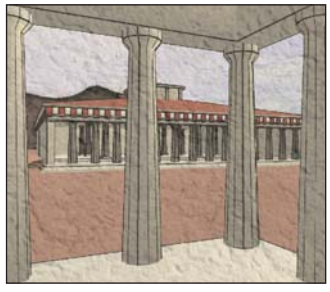


Figure 35.14: The Dynamic Canvas algorithm [CTP+03] produces background-paper detail at multiple scales that transform evocatively under 3D camera motion. (Courtesy of Joelle Thollot, "Dynamic Canvas for Non-Photorealistic Walkthroughs," by Matthieu Cunzi, Joelle Thollot, Sylvain Paris, Gilles Debunne, Jean-Dominique Gascuel and Fredo Durand, Proceedings of Graphics Interface 2003.)

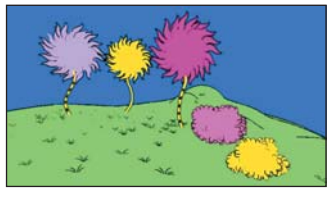


Figure 35.15: The view-dependent tufts on the trees, grass, and bushes are rendered with graftals that move coherently between adjacent frames of camera animation. (Courtesy of the Brown Graphics Group, ©2000 ACM, Inc. Reprinted by permission.)

at the silhouette. Of course, this only amortizes the incoherence, because there is still a pop when the graftal is replaced. Previously discussed strategies such as 2D composition can then reduce the incoherence of the pop.

An open question in expressive rendering is the significance of temporal coherence for large objects like strokes. On the one hand, we know that their motion is visually distracting. On the other hand, films have been made with hand-drawn cartoons and live-action stop motion in the past. There, the incoherence can be considered part of the style and not an artifact. Classic stop-motion animation involves taking still images of models that are then manually posed for the next frame. When the stills are shown in sequence, the models appear to move of their own volition because the intermediate time in which the animator appeared in the scene to manipulate it is not captured on film.

35.4 Representations

We now talk about animation methods, the naming of parts of animatable models, and alternatives among which one might choose to express the parameters and computational model of animation.

The **state** of an animated object or scene is all of the information needed to uniquely specify its pose. For animation, a scene representation must encompass both the state and a parameterization scheme for controlling it. For example, how do we encode the shape and location of an apple and the force of gravity on it?

As is the case in rendering, one generally wants the simplest representation that can support plausible simulation of an object. For rendering, interaction with light is significant, so the surface geometry and its reflectance properties must be fairly detailed. For animation, interaction with other objects is significant, so properties like mass and elasticity are important. Animation geometry may be coarse, and different from that used for rendering. A variety of animation representations have been designed for different applications. This chapter references many and explores particles and fluid boundaries in depth as case studies.

We categorize schemes for parameterizing, and thus controlling, state into key poses created by an artist, dynamics simulation by the laws of physics, and explicit procedures created by an artist-programmer. Many systems are hybrids. These leverage different control schemes for different aspects of the scene to accommodate varying simulation level of detail or artistic control.

35.4.1 Objects

The notion of an object is a defining one for an animation system. For example, by calling an automobile an "object" one assumes a complex simulation model that abstracts individual systems. If one instead considers an individual gear as an "object," then the simulation system for an automobile is simple but has many parts. This can be pushed to extremes: Why not consider finite elements of the gears themselves, or molecules, or progress the other way and consider all traffic on a highway to be one "object"?

The choice of object definition controls not only the complexity of the underlying simulation rules, but also what behaviors will emerge naturally versus requiring explicit implementation. For example, finite-element objects might naturally simulate breaking and deformation of gears and bricks, whereas atomic gear

or brick objects cannot break without explicit simulation rules for creating new objects from their pieces.

How much complexity do we need to abstract the behaviors of scenes that we might want to simulate? A tumbling crate retains a rigid shape relative to its own reference frame, but that frame moves through space. A walking person exhibits underlying articulated skeleton of rigid bones connected at joints that are then covered by deforming muscle and skin. The water in a stream lacks any rigid substructure. It deforms around obstacles and conforms to the shape of the stream bed under forces including gravity, pressure, and drag.

In each of these scenarios, the objects involved have varying amounts of state needed to describe their poses and motion. The algorithms for computing changes of that state vary accordingly.

Some object representations commonly employed in computer graphics (with examples) are

1. Particles (smoke, bullets, people in a crowd)
2. Rigid body (metal crate, space ship)
3. Soft rigid body (beach ball)
4. Articulated rigid body (robot)
5. Mass-spring system (cloth, rope)
6. Skinned skeleton (human)
7. Fluid (mud, water, air)

These are listed in approximate order of complexity and algorithmic state. For example, the dynamic state of a particle consists of its position and velocity. A rigid body adds a 3D orientation to the particle representation.

Why are so many different representations employed? As an alternative, a single unified representation would be much more theoretically appealing, be easier from a software engineering perspective, and automatically handle the tricky interactions between objects of different representations.

One seemingly attractive alternative is to choose the simplest representation as the universal one and sample very finely. Specifically, all objects in the real world are composed of atoms, for which a particle system is an appropriate representation. Although it is possible to simulate everything at the particle level [vB95], in practice this is usually considered awkward for an artist and overwhelming for a physical simulation algorithm.

35.4.2 Limiting Degrees of Freedom

The authoring method and representation do not always match what is perceived by the viewer. For example, many films and video games move the root frame of seemingly complex characters as if they were simple rigid bodies under physical simulation. In both cases, the individual characters are also animated by key pose animation of skinned skeletons relative to their root frames. Thus, at different scales of motion the objects have different specifications and representations. This avoids the complexity of computing true interactions between characters while retaining most of the realism.

This is analogous to level-of-detail modeling tricks for rendering. For example, a building may be represented by boxlike geometry, a bump map that

describes individual bricks and window casings, and bidirectional scattering distribution functions (BSDFs) that describe the microscopic roughness that makes the brick appear matte and the flower boxes appear shiny.

Switching to a less complex object representation is a way to reduce the number of independent (scalar) state variables in a physical system, also known as the number of **degrees of freedom** of a system. For example, a dot on a piece of paper has two degrees of freedom—its x- and y-positions. A square drawn on the paper has four degrees of freedom—the position of the center along horizontal and vertical axes, the length of the side, and the angle to the edge of the page. A 3D rigid body has trillions of degrees of freedom if the underlying atoms are considered, but only six degrees of freedom (3D position and orientation) if taken as a whole. Simulating the root positions of the characters in a crowd is a reduction of the number of degrees of freedom from simulating the muscles of every individual character.

Furthermore, an object may be modeled for rendering purposes with much higher detail than is present for simulation. For example, a space ship can be modeled as a cylinder for inertia and collision purposes but rendered with fins, a cockpit, and rotating radar dishes without the viewer perceiving the difference.

Separating the rendering representation, motion control scheme, and object representation introduces error into the simulation of a virtual world. This may or may not be perceptually significant. From a system design perspective, error is not always bad. In fact, acceptable error can be your friend: It provides room to tweak and choose where to put simulation (and therefore development) effort.

35.4.3 Key Poses

In a **key pose** animation scheme (a.k.a. key frame, interpolation-based animation), an animation artist (**animator**) specifies the poses to hit at specific times, and an algorithm computes the intermediate poses, usually in the absence of full physics.

The challenges in key pose animation are creating suitable authoring environments for the animators and performing interpolation that conserves important properties, such as momentum or volume. Because an animator's creation is expressive and not necessarily realistic or algorithmic, perfect key pose animation is ultimately an artificial intelligence problem: Guess the intermediate pose a human animator would have chosen. Nonetheless, this is the most popular control scheme for character performances, and for sufficiently dense key poses it is considered a solved problem with many suitable algorithms.

35.4.4 Dynamics

In a **dynamics** (a.k.a. physically based animation, simulation) scheme, objects are represented by positions and velocities and physical laws are applied to advance this state between frames. The laws need not be those of real-world physics.

The laws of mechanics from physics are well understood, but generally admit only numerical solutions. Two challenges in dynamics are stability and artistic control. It is hard to make numerical methods efficient while preserving stability, that is, conserving energy, or at least not increasing energy and "exploding." It is also hard to make realistic physics act the way that an art director might want (e.g., a film explosion blowing a door directly into the camera or a video game car that skids around corners without spinning out).

35.4.5 Procedural Animation

In a **procedural animation,** the artist, who is usually a programmer, specifies an explicit equation for the pose at all times. In a sense, all computer animation is procedural. After all, to execute an animation on a computer a *procedure* must be computing the new object positions. In the case of key pose animation that procedure performs interpolation and in the case of dynamics it evaluates physical forces. However, it is useful to think of a separate case where we specify an explicit, and typically not physically based, equation for the motion. Our cannonball example straddled the line between dynamics and procedural animation. A good example of complex procedural motion is Perlin's [Per95] dancer, whose limbs move according to a noise function.

Procedural animations today are primarily used for very simple demonstrations, like a planet orbiting a star, and for particle system effects in games. The lack of general use is probably because of the challenges of encoding artist-specified motion as an explicit equation and making such motion interact with other objects.

35.4.6 Hybrid Control Schemes

The current state of the art is to combine poses with dynamics. This combines the expression of an actor's performance or artist's hand with the efficiency and realism of physically based simulation. There are many ways to approach hybrid control schemes. We outline only a few of the key ideas here.

An active research topic is adjusting or authoring poses using physics or physically inspired methods. For example, given a model of a chair and a human, a system autonomously solves for the most stable and lowest-energy position for a sitting person. A classic method in this category is **inverse kinematics** (IK). An IK solver is given an initial pose, a set of constraints, and a goal. It then solves for the intermediate poses that best satisfy these (see Figure 35.16), and the final pose if it was underdetermined. For example, the pose may be a person standing near a bookshelf. The goal may be to place the person's hand on a book that is on the top of the shelf, above the character's head. The constraints may be that the person must remain balanced, that all joints remain connected, and that no joint exceeds a physical angular limit. IK systems are used extensively for small modifications, such as ensuring that feet are properly planted when walking on uneven terrain, and reaching for nearby objects. They must be supported by more complex systems when the constraints are nontrivial.

It is also often desirable to insert a previously authored animation into a novel scene with minor adjustment; for example, to adapt a walk animation recorded on a flat floor to a character that is ascending a flight of stairs without the feet penetrating the ground or the character losing balance. This is especially the case for video games, where the character's motion is a combination of user input, external forces, and preauthored content [MZS09, AFO05, AdSP07, WZ10].

As we said earlier, motion planning is an AI problem. But it is closely coupled with dynamics. Getting dressed in the morning is more complex than reaching for a book—it cannot be satisfied by a single pose. Presented with a dresser full of clothing, a virtual human would have to not only find the series of poses required to step into a pair of pants while remaining balanced, but also realize that the

Figure 35.17: Four images of animated characters autonomously performing complex tasks requiring motion planning. (Courtesy of the Graphics Lab at Carnegie Mellon University. ©2004 ACM, Inc. Reprinted by permission.)

Figure 35.19: A complex traversal requiring both pathfinding and general motion planning [SH07]. (Courtesy of Jessica Hodgins and Alla Safonova ©2007 ACM, Inc. Reprinted by permission.)

drawers must be opened before the clothes could be taken out. Figure 35.17 shows some examples of "simple" daily tasks that represent complex planning challenges for virtual characters.

A special case of motion planning is the task of navigating through a virtual world. At a high level, this is simply pathfinding for a single character. When there are enough characters to form a crowd, the multiple-character planning problem is more challenging. Creating the phenomenon of real crowds (or herds, or flocks . . .) is surprisingly similar to simulating a fluid at the particle level, with global behavior emerging from local rules as shown in Figure 35.18.

When the space to be traversed does not admit a character's default method of locomotion, the pathfinding problem is of course much harder. For example, the character in Figure 35.19 must not only find a path, but also decide when to crouch and when to jump to avoid obstacles on the desired path while satisfying physical constraints on the body.

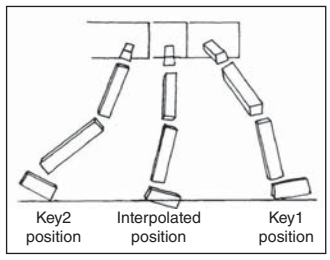


Figure 35.16: An interpolated leg position between key poses found by one of the earliest inverse kinematics algorithms. (Courtesy of A.A. Maciejewski, ©1985 ACM, Inc. Reprinted by permission.)

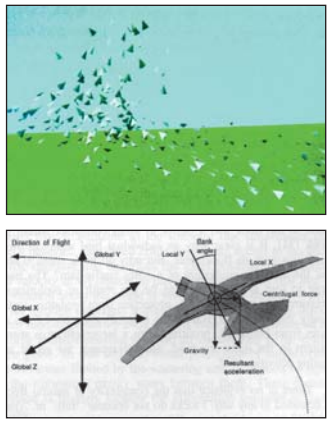


Figure 35.18: Complex global flocking behaviors (top) emerge in Reynolds's seminal "boids" animation system [Rey87] from simple, local rules for each virtual bird (bottom). (©1987 ACM, Inc. Included here by permission.) (Courtesy of Craig Reynolds, ©1987 ACM, Inc. Reprinted by permission.)

Secondary motion is the motion of small parts of a figure relative to its root motion; for example, the flowing of cloth and hair and the jiggling of muscle and fat. Secondary motion is important to our perception of motion and performance, but it is often inefficient to simulate as part of the entire system or explicitly pose. Animators often develop special-case secondary motion simulators that create the character of the motion detail without the full cost [PH06, BBO+09, JP02]. This is analogous to modeling small-scale visual detail in texture instead of geometry.

It is often desirable to transfer an animation performance that was authored or captured on one body to another body [BVGP09, SP04, BCWG09, BLB+08]. There are some cases where there is little choice but to transfer motion between bodies. For example, to animate a centaur with motion capture, we must transfer the motion from a human and a horse onto a single virtual creature. In other cases, it is a matter of cost. Rather than recording the performance of actors of many sizes wearing many costumes, with transfer we could record a single actor and adapt the performance to multiple characters in a crowd [LBJK09].

Plausible animation is the problem of working backward to compute the *start* state of a system, given the end state [BHW96, KKA05, YRPF09, CF00, TJ07, MTPS04]. For example, a film shot may require a player to roll two sixes on a pair of dice. We would like to start with the sixes facing up at the end of the shot, and solve backward for a physically viable series of bounces as the dice roll that leads them to that position. Since the problem may be overconstrained, we are willing to accept any "plausible" solution in which the laws of physics are bent only in ways that are imperceptible to the average observer. It is plausible for the momentum of a die to be exaggerated by 5% to produce one extra tumble, but not for the die to bounce three meters in the air off the initial throw.

35.5 Pose Interpolation

35.5.1 Vertex Animation

The most straightforward way to represent poses is to specify a separate mesh for each key frame. The mesh topology is usually constant over the animation so that every key frame contains the same number of vertices. Only their positions change, not their adjacency and ordering. To simplify the discussion, assume that key poses $\mathbf{k}[t]$ are defined at the ends of integers at time intervals, and that we want to form a continuous expression for the pose at the fractional times t in between these poses.

Sample-and-hold interpolation,

$$\mathbf{x}(t) = \mathbf{k}[\lfloor t \rfloor], \tag{35.14}$$

produces substantial temporal aliasing. The character simply holds its position until the end of the time period and then instantaneously snaps to the next pose. This also produces infinite velocity at the frame intervals because $\mathbf{x}(t)$ is discontinuous.

We'd like to make smoother transitions with finite velocities. As we said in the introduction, one solution is to linearly interpolate between key poses, which produces continuous positions but discontinuous velocity. That is, we've ensured C^0 continuity but still exhibit infinite acceleration, which is unnatural for character motion. A higher-order interpolation scheme can produce smoother

interpolation. This is a progression that we've seen before, when discussing splines in Chapter 22.

Splines are in fact a common solution to the problem of interpolating between key poses. Fitting one Catmull-Rom spline per vertex produces globally smooth animation. Vertex-position splines do not address all of the key pose problems, however. They still require storage proportional to the product of the vertex count and the number of key poses and require a number of interpolation operations linear in the vertex count at runtime. Vertex splines require an artist to pose individual vertices instead of working with higher-level primitives like limbs. Furthermore, the results can be hard to control.

Because each vertex is animated independently and the vertices are only on the surface of the character, there are no constraints to preserve volume or surface area during animation.

35.5.2 Root Frame Motion

Object geometry and vertex animation is typically expressed relative to the root frame of the object. This means that a walking character will stay at the origin while its feet swing below the center of mass. To make the character move around the world we could create an animation in which the vertices all travel away from that starting position; however, this would require a tremendous amount of redundant animation data. Instead, one typically models the transformation of the object root. Root transformations are useful for more than just visible objects in the virtual world. Lights, cameras, 3D widgets, and pointers all define their own frames which may be subject to motion. Throughout this book, we've constantly worked with transformations between reference frames, and we've seen several alternative representations of a 3D reference frame in Chapters 6, 11, and 12:

- 4×4 matrix
- 3×3 rotation matrix and translation vector
- Euler angles: roll, yaw, pitch, and translation vector
- Rotation axis, rotation angle, and translation vector
- Unit quaternion and translation vector

In rendering, the 4×4 matrix representation is often convenient and is what most APIs have adopted. Because conversions exist between all of these representations, the animation system need not use the same representation as the rendering system. Although the matrix representation is sometimes convenient, the quaternion-plus-vector representation is often preferred for simulation during animation and the Euler frame for motion specification of animation.

When choosing a representation, one typically seeks to minimize error and simplify computations—both for performance and for readability. In animation, we'll interpolate, differentiate, and integrate expressions containing the reference frames. For a rotating body, performing those operations on the inherently linear matrix structure will typically send points along the tangent to the desired sphere of motion. For small steps, we can correct that error by projecting back onto the sphere of motion, but small steps are not always possible or efficient. Although not without their own problems, the quaternion and angle formulations are better suited to expressing operations on the surface of a sphere. Among these, the axis-angle representation presents something of the same problem as the matrix.

The axis itself must rotate to describe rotations in other planes, and the axis is a linear representation. Euler angles prefer the arbitrary axes chosen (i.e., the magnitude of the derivatives depends on the direction of rotation) and allow gimbal lock under successive 90° rotations. The uniformity of the quaternion therefore leaves it as the best choice in many cases, particularly for freely moving bodies.

Now consider the problem of specifying an object's motion or the forces that inspire that motion. We often want to choose the reference frame within which it is easiest to create this specification. For many objects, a suitably chosen Euler frame is ideal. For example, an automobile's front wheels rotate about exactly two axes relative to the car frame. An airplane's controls affect yaw, roll, and pitch in its own reference frame.

Once the root frame has been transformed, the rigid or dynamic body attached to it can also be moved by simply transforming all of the vertices from the root frame to the world frame. Note that although it is common to place the root frame's origin within the body (and often at the center of mass), this is just a reference frame, and its origin can lie *outside* the geometry itself.

35.5.3 Articulated Body

Many objects that we would like to model as a single scene element change their shape as well as their root position and orientation. Some of these objects can be modeled as a collection of bodies that are individually rigid but which transform relative to one another. For example, an automobile can be modeled as a rigid frame and four rigid wheels that rotate relative to the frame. Each vertex in the automobile is contained within exactly one of the bodies, and can therefore be expressed relative to one body's frame.

A natural way of organizing the bodies relative to one another is a small "scene graph" subtree (see Chapter 6). There is some root body, whose children are other bodies in its frame and the vertices defining the shape of the root body. The other bodies recursively have their own children. In the case of the automobile, it would be natural to choose the car's frame as the root body. However, a nice property of a tree is that any node can be chosen as the root and the result is still a tree. So we could choose the front-left wheel to be the root, with the frame as its only child body, and the frame would then have three other wheels as child bodies. That choice would probably make it awkward to express the forces exerted by the drive train on the wheels since the tree has little symmetry, but it is a mathematically valid model of the system.

We call this structure an **articulated rigid body** because the edges of the scene graph typically correspond to **joints** in the model. A joint is a constraint on the relative movement of two bodies. For the automobile, these are the axles. For an android they would be the knees, elbows, waist, etc.; for a building they would be the door hinges and grooves within which the windows slide. Geometry is often added to the model to visually depict the physical basis for the constraint. However, the animation joints need have no visual representation or physical analog. Typically one puts the root frame of a body at the joint where it is connected to its parent, since that is the frame in which it is most natural to express the joint's constraint and forces on the two bodies.

An advantage of the articulated rigid body is that complex dynamic objects, such as most machines, can be represented without vertex animation. This is more

efficient in both space and time and much simpler to implement. Except at joints, the representation automatically preserves volume.

The limitation of an articulated rigid body is that motion necessarily appears mechanical because the individual pieces remain rigid. Were we to animate a character in this way, the character would appear robotic instead of humanoid. A more general solution is the **articulated body.** This maintains a scene graph of reference frames, but performs some other kind of animation, such as keyframe vertex interpolation, on the data within those reference frames. The particular combination of rigid frames with vertex-interpolated key frames was very popular for real-time applications like games until fairly recently, when it was overtaken by the more general scheme of skeletal animation.

35.5.4 Skeletal Animation

The shape of the surface of a living creature under motion is dictated by the dynamic position of its skeleton and the constraints of its musculature and other internals. It is sensible to consider parameterizing virtual objects that model such creatures in a similar way. (This is an example of the Wise Modeling principle that says that you should decide what phenomena you want to model, and then be sure your model is rich enough to represent those phenomena.) This is known as a **skeletal animation** (a.k.a. **matrix skinning**) model.

A skeletal animation model defines a set of reference frames called **bones.** That is evocative but something of a misnomer, because reference frames correspond most closely to the *joints* of the character, not the bones between those joints. The bones may be in a common object space or arranged hierarchically into a tree with parent-child relationships.

We are used to expressing a point as a weighted combination of axis vectors in some reference frame. A single vertex in a skeletal animation mesh is a weighted combination of points in multiple reference frames. For example, a point near a human's elbow may be defined as halfway between a point defined in the shoulder/upper arm's reference frame and a point defined in the elbow/forearm's reference frame. This joint parameterization allows that vertex and its similarly defined neighbors to define a smoothly deforming mesh near the elbow as it bends. That is in contrast to the sharp intersection of surfaces defined in different reference frames observed under articulated rigid body animation.

We can think of a point $\mathbf{x}$ represented as a linear combination of specific points P_b transformed by corresponding bone transformation matrices $\mathbf{B}_b$ at given times under various scalar weights w_b:

$$\mathbf{x}(t) = \sum_{b \in \text{bones}} (\mathbf{B}_b(t) P_b) w_b, \tag{35.15}$$

where $\sum w_b = 1$. However, since P_b and w_b are both part of the parameterization that we will compute from an artist's input and all relevant operators are linear, in practice we need only store 3-vectors $P'_b = P_b w_b$.

One typically constructs the representation by having an artist place a skeleton inside a mesh that is in some standard pose. The artist then assigns bone weights to each vertex and the system computes P'_b for each vertex from them. A first approximation might be that each vertex has a single nonzero weight, which is for its position in the standard pose relative to the closest reference frame. The artist then manipulates the skeleton to create a different pose and the vertices transform

accordingly. Those near joints are likely to land in undesirable positions, so the artist moves them to the desired position. The system can then recompute a set of P'_b values that minimize the position representation error under both inputs. The artist then repeats the process for more poses. Since this quickly becomes an overconstrained optimization process, the resultant weights might not satisfy the visual goal sufficiently in any pose. The artist can reduce the constraints on the system by adding more bones. Often this process results in a majority of bones with no physical interpretation in the original creature—they are present merely to offer enough degrees of freedom for the optimizer to satisfy each of the desired pose deformations. Since the bones need not have a true skeletal correspondence, skeletal animation can also be applied to objects like trees or even water that has no proper skeleton but which exhibits smooth deforming animation.

Learning to intuitively introduce new "bones" to yield the desired result from a nonlinear optimizer is difficult. This is one of the reasons that skilled rigging animators are in great demand. Just as we can ease the difficulty of approximating arbitrary shapes with a mesh by increasing the tessellation, we can ease the challenge of rigging a skeleton by increasing the number of bones available. The drawbacks of doing so follow from that analogy as well. At animation creation time (versus model rigging time), each bone must be positioned for each key frame. The number of bones obviously increases the difficulty and time cost of animation. At runtime, the system must transform each vertex. The summation for a single vertex in Equation 35.15 need only be performed for each matrix corresponding to a nonzero weight. Yet even if there are as few as three bones affecting each vertex, that transformation will be three times as expensive as for a rigid body with a single transformation, and the vertices must be batched into small (inefficient) groups that share the same subset of bone matrices versus large uniform streams that are more amenable to efficient processing on parallel architectures.

35.6 Dynamics

35.6.1 Particle

In dynamics, a **particle** is an infinitesimal body. At human scale, a particle is a reasonable approximation for a molecule, or a grain of sand. At astronomical scale, planets and moons can be modeled as particles. We study particles for two reasons. First, particles and **particle systems** are frequently used in graphics to model either very small objects, such as bullets, or amorphous compressible objects, such as smoke, rain, and fire [Ree83]. The second reason that we study particles is that they provide a simple system in which to derive dynamics. After deriving the dynamics of a particle we will then generalize to more complex bodies.

Although the particles are infinitesimal, we might still choose to render them as with geometry or billboards. This improves their appearance (a zero-volume particle wouldn't be visible!) and gives the impression of more complexity than is actually being simulated.

Given the initial position of a particle, its initial velocity, and an expression for its acceleration over time, we want an expression for its position at a later time. Let the unknown position function be $t \mapsto \mathbf{x}(t)$, where the known initial position is $\mathbf{x}(0)$. Recall that $\mathbf{x}(0)$ denotes a vector in 3-space. Particles are modeled as points, so we can ignore rotation. This simplifies the physics. Furthermore, for translation, we can treat each axis independently, with two exceptions related to

contact: For collisions, objects must overlap along all axes (e.g., at noon on the equator you and the sun have the same xz-position in your local reference frame, but are separated by a huge vertical distance, so you aren't inside the sun); and for contact force, we need the normal components from all axes to compute the force along each one. For the moment we ignore contact.

Velocity is change in position divided by change in time. When we say "velocity" we typically are referring to **instantaneous velocity** $\dot{\mathbf{x}}(t)$, which is the limit of velocity as the time duration approaches zero:

$$\text{velocity} = \dot{\mathbf{x}}(t) = \lim_{\Delta t \to 0} \frac{\mathbf{x}(t + \Delta t/2) - \mathbf{x}(t - \Delta t/2)}{\Delta t}. \tag{35.16}$$

Acceleration has the same relationship to velocity as velocity has to position. Instantaneous acceleration can therefore be expressed as the second time derivative of position:

$$\text{acceleration} = \ddot{\mathbf{x}}(t) = \lim_{\Delta t \to 0} \frac{\dot{\mathbf{x}}(t + \Delta t/2) - \dot{\mathbf{x}}(t - \Delta t/2)}{\Delta t}. \tag{35.17}$$

In this notation, our problem is thus to derive an expression for $\mathbf{x}(t)$ given $\mathbf{x}(0)$, $\dot{\mathbf{x}}(0)$, and $\ddot{\mathbf{x}}(t)$. By the second fundamental theorem of calculus, if $\mathbf{x}(t)$ is differentiable, then

$$\mathbf{x}(t) = \mathbf{x}(0) + \int_0^t \dot{\mathbf{x}}(s)\, ds. \tag{35.18}$$

Since we have the first and second derivatives of $x(t)$, we will assume that it is in fact a differentiable function.

We cannot apply Equation 35.18 directly because we have no explicit expression for $\dot{\mathbf{x}}(t)$ in our initial state. Therefore, we apply the second fundamental theorem again to obtain an expression in terms of only known values:

$$\dot{\mathbf{x}}(s) = \dot{\mathbf{x}}(0) + \int_0^s \ddot{\mathbf{x}}(r)\, dr \tag{35.19}$$

$$\mathbf{x}(t) = \mathbf{x}(0) + \int_0^t \left(\dot{\mathbf{x}}(0) + \int_0^s \ddot{\mathbf{x}}(r)\, dr \right) ds \tag{35.20}$$

$$\mathbf{x}(t) = \mathbf{x}(0) + \dot{\mathbf{x}}(0)t + \int_0^t \int_0^s \ddot{\mathbf{x}}(r)\, dr\, ds \tag{35.21}$$

One way to produce a dynamics animation is to evaluate this integral for $\mathbf{x}(t)$ analytically and then evaluate it at successive times $t = \{0,\ \Delta t,\ 2\Delta t,\ \ldots\}$. When an expression is available for $\ddot{\mathbf{x}}(t)$ and it is integrable in elementary (symbolic) terms, this approach is convenient and produces no incremental position error throughout the animation. The analytic approach is viable for simple scenarios, such as a body falling or sliding under constant linear acceleration or a satellite orbiting a planet under constant radial acceleration (both examples due to gravity). Introductory physics textbooks focus on such problems because most complicated scenarios cannot be solved analytically.

35.6.2 Differential Equation Formulation

Often we have the means to compute instantaneous acceleration given the position and velocity of an object, but no analytic solution for all time. For example, by Newton's second law ($F = m \cdot a$), the net acceleration $\ddot{\mathbf{x}}(t)$ experienced by a body

is the net force applied to it, divided by its mass. The force can often be computed from the current position and velocity of the particle, so given values $\mathbf{x}(t)$ and $\dot{\mathbf{x}}(t)$, we could compute

$$\ddot{\mathbf{x}}(t) = \frac{\mathbf{f}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t))}{m} \tag{35.22}$$

for a particle of known mass m and force function $\mathbf{f}$ at any possible time t. (Exceptions arise in situations where the force is controlled by user input, for example.)

Equation 35.22 is a (vector) **differential equation** because it relates $\mathbf{x}$ and its derivatives. It is specifically an **ordinary differential equation** (ODE) because $\mathbf{x}$ is a function of a single variable, t. Because $\mathbf{f}$ is arbitrary, this happens to be a nonlinear differential equation, which is a hard class to solve. In fact, there is no known analytic solution for general equations of this form.

Introducing an acceleration function does not immediately help us. That's because we started with the problem that we didn't have analytic solutions for $\mathbf{x}$ and $\dot{\mathbf{x}}$ for all time. Introducing a new function that depends on the two functions we don't know (and which can't be evaluated analytically) seems like we're going in the wrong direction. The key idea here is that *if* we had values of the functions at a single time, we could compute the forces and thus the acceleration that will help advance the simulation toward a future time without explicitly knowing exactly where it is headed. This will lead us to a numeric integration strategy. This is an important point that underlies most dynamics algorithms. Let us consider a concrete example.

Consider a 1D system of a single falling ball whose height is given by height $\mathbf{x}(t)$ relative to the ground. Following the notation introduced in Chapter 7, we can explicitly give the type of this function:

$$\mathbf{x} : \mathbf{R} \to \mathbf{R} : t \mapsto \mathbf{x}(t) \tag{35.23}$$

(which we might implement in C as `float x(float t);`, but again, this is a function that we'll assume we don't actually have an explicit implementation for).

Likewise, the force function $\mathbf{f}$ is

$$\mathbf{f} : \mathbf{R^3} \to \mathbf{R} : (t, y, v) \mapsto \mathbf{f}(t, y, v) \tag{35.24}$$

(which we might implement in C as `float f(float t, float y, float v);` and for which we *are* assuming the implementation exists).

The function $\mathbf{f}$ computes the force on a ball at time t if the ball currently has height y and velocity v. We might model this force as gravity on the Earth's surface:

$$\mathbf{f}(t, y, v) = -9.81 \text{ kg} \cdot \text{m/s}^2. \tag{35.25}$$

Note that under this model $\mathbf{f}$ is constant. It doesn't depend on time, position, or velocity.

A more realistic model accounts for the fact that as the ball moves faster, it experiences more friction with the surrounding air. This drag force inhibits the ball's motion. That is, it decelerates the ball along the velocity vector. Choosing an arbitrary drag coefficient, we might enhance our model to

$$\mathbf{f}(t, y, v) = -9.81 \text{ kg} \cdot \text{m/s}^2 - v \cdot 0.5 \text{ kg/s}. \tag{35.26}$$

Under this new force model, the force on a ball with velocity $v = 0$ m/s is -9.81 kg $\cdot$ m/s^2, while the force on a ball that is already falling with velocity $v = -2$ m/s has lower magnitude: -8.81 kg $\cdot$ m/s^2. This new force model still does not depend on time or position. But we could imagine a more sophisticated model that includes air currents, whose effects depend on the location in space time at which the current is experienced.

Now, consider the ball moving in three spatial dimensions. How do the types of the functions change? Positions and velocities are now 3-vectors:

$$\mathbf{x} : \mathbf{R} \to \mathbf{R^3} \tag{35.27}$$

$$\mathbf{f} : \mathbf{R^1} \times \mathbf{R^3} \times \mathbf{R^3} \to \mathbf{R^3} : (t, \mathbf{y}, \mathbf{v}) \mapsto \mathbf{f}(t, \mathbf{y}, \mathbf{v}) \tag{35.28}$$

$$: \mathbf{R}^7 \to \mathbf{R^3} : (t, \mathbf{y}, \mathbf{v}) \mapsto \mathbf{f}(t, \mathbf{y}, \mathbf{v}) \tag{35.29}$$

The force function still takes three arguments. Because it is always applied to position and velocity, people sometimes write $\mathbf{f}(t, \mathbf{x}, \dot{\mathbf{x}})$, treating the $\mathbf{x}$ and $\dot{\mathbf{x}}$ functions as if they were variables.[2] Recall the pitfalls of that notation identified earlier in the chapter. So, although you may see that as a convenience in animation notes and research papers, we will continue to be careful to distinguish the function and its value here.

35.6.3 Piecewise-Constant Approximation

Assume for the moment that the force function is constant with respect to time, and that acceleration is therefore also constant. In a physics textbook on Newtonian mechanics, the final position of a body experiencing constant acceleration would be expressed as

$$\mathbf{x}_1 = \mathbf{x}_0 + \mathbf{v}_0 t + \frac{1}{2}\mathbf{a}_0 t^2, \tag{35.30}$$

$$\mathbf{x}(t) = \mathbf{x}(0) + \dot{\mathbf{x}}(0)t + \frac{1}{2}\ddot{\mathbf{x}}(0)t^2, \text{ and} \tag{35.31}$$

$$= \mathbf{x}(0) + \dot{\mathbf{x}}(0)t + \frac{1}{2}\frac{\mathbf{f}(0, \mathbf{x}(0), \dot{\mathbf{x}}(0))}{m}t^2, \tag{35.32}$$

where the second version follows our notation. The right side is quadratic in t and all other factors are constant, so this describes a parabola. That is the arc of a thrown ball, which experiences essentially constant acceleration due to gravity, so this matches our intuition for the position of an object as a function of time. These equations arise from integrating Equation 35.21 under the constant acceleration assumption:

2. We could make this notation meaningful by redefining force as a higher-order function, $\mathbf{f} : \mathbf{R} \times (\mathbf{R} \to \mathbf{R^3}) \times (\mathbf{R} \to \mathbf{R^3}) \to \mathbf{R^3}$, but contorting ourselves to make the notation consistent is not useful because in practice we will apply force function to points and velocities from the previous frame, not functions over all time.

$$\mathbf{x}(t) = \mathbf{x}(0) + \int_0^t \left(\dot{\mathbf{x}}(0) + \int_0^s \ddot{\mathbf{x}}(0)\, dr \right) ds \tag{35.33}$$

$$= \mathbf{x}(0) + \int_0^t (\dot{\mathbf{x}}(0) + \ddot{\mathbf{x}}(0)s)\ ds \tag{35.34}$$

$$= \mathbf{x}(0) + \dot{\mathbf{x}}(0)t + \frac{1}{2}\ddot{\mathbf{x}}(0)t^2 \tag{35.35}$$

$$= \mathbf{x}(0) + \dot{\mathbf{x}}(0)t + \frac{1}{2}\frac{\mathbf{f}(0, \mathbf{x}(0), \dot{\mathbf{x}}(0))}{m}t^2 \tag{35.36}$$

Thus far, we haven't advanced over the original analytic solution, since we could always evaluate the integral when acceleration is constant. But we can now generalize this result. Assume that $\mathbf{f}$ is only constant for each time interval from t_i to t_{i+1} of duration Δt, but may change between intervals. Acceleration is now only piecewise constant over time, which is a much better approximation of the real world.

Under this assumption, we can rework Equation 35.36 to advance a known state described by $\mathbf{x}(t_1)$ and $\dot{\mathbf{x}}(t_1)$ at the beginning of the time interval to the state at the end of the time interval under constant acceleration:

$$\mathbf{x}(t_2) = \mathbf{x}(t_1) + \dot{\mathbf{x}}(t_1)\Delta t + \frac{1}{2}\frac{\mathbf{f}(t_1, \mathbf{x}(t_1), \dot{\mathbf{x}}(t_1))}{m}\Delta t^2 \tag{35.37}$$

$$\dot{\mathbf{x}}(t_2) = \dot{\mathbf{x}}(t_1) + \frac{\mathbf{f}(t_1, \mathbf{x}(t_1), \dot{\mathbf{x}}(t_1))}{m}\Delta t \tag{35.38}$$

(for constant force on the interval). This is **Heun-Euler integration,** also known as **Heun integration** or **Improved Euler integration** (it is distinct from just "Euler" integration, described in Section 35.6.7.1).

Nothing in these equations assumes a specific number of spatial dimensions, so they hold equally well for the common cases of 1D, 2D, or 3D particles. In fact, we can generalize even further. Rather than limiting $\mathbf{x}$ to describing the motion of a single particle, we can pack the positions of multiple particles into $\mathbf{x}(t)$ by simply treating them as separate dimensions. We'll later generalize this even further and encode orientation for bodies with volume in the same vector.

This integration scheme is by no means perfect, but it is often good enough. If the animation doesn't look right or is unstable, you can often reduce the time interval and the quality will improve. This is because in the limit as $\Delta t \to 0$ s the Heun-Euler's estimate approaches the true integral of any arbitrary (integrable) force function. For systems with large or high-frequency forces, you may have to make Δt so small to ensure stability that the simulation becomes quite computationally intense. Section 35.6.6 discusses integration schemes that are more efficient than Heun-Euler for such scenarios.

35.6.4 Models of Common Forces

There are fundamental and derived forces. Fundamental forces such as gravity and electrical force are the elements of the standard physics model and cannot be simplified to the interaction of other forces under that model. Derived forces such as buoyancy are a method for abstracting many microscopic forces into a simple high-level model for macroscopic behavior.

Recall that the net force on a system is written $\mathbf{f}(t, \mathbf{y}, \mathbf{v})$. If we were considering a single body, then it would have its center of mass be at point $\mathbf{y}$ and linear velocity $\mathbf{v}$. In terms of our position function $\mathbf{x}$, we can and often will apply the force function as $\mathbf{f}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t))$. As previously discussed, we make the position and velocity explicit arguments distinct from $\mathbf{x}$ because that is the form of the function when incorporated into a dynamics solving system.

Forces always arise between a pair of objects. By Newton's third law of motion, both objects in the pair experience the force with the same magnitude but in opposing directions. Thus, it is sufficient to describe a model of the force on one object in the pair. Likewise, one need only explicitly compute the force on one object in the pair when implementing a simulator. In a system with multiple bodies, the $\mathbf{y}$ and $\mathbf{v}$ arguments are in arrays of vectors, and the net force function $\mathbf{f}$ computes the force on all objects, due to all objects.

For n bodies, there are $O(n^2)$ pairs to consider. Forces combine by superposition, so the net force on object 1 is the sum of the forces from all other objects on object 1. This means that the general force implementation of the force function looks like Listing 35.2, where the pairwise force $\mathcal{F}$`(t, y, v, i, j)` function computes the force on the object with index i due to the object with index j. There will be many kinds of forces, such as gravity and friction, so we consider an array of `numForces` instances of $\mathcal{F}$.

Listing 35.2: Naive implementation of the net force function.

```
// Net force on all objects
Vector3[n] f(float t, Vector3[n] y, Vector3[n] v) {
   Vector3[n] net;

   // for each pair of objects
   for (int i = 0; i < n; ++i) {
      for (int j = i + 1; j < n; ++j) {

         // for each kind of force
         for (int k = 0; k < numForces; ++k) {
            Vector3 fi = F[k](t, y, v, i, j);
            Vector3 fj = -fi;
            net[i] += fi;
            net[j] += fj;
         }

      }
   }

   return net;
}
```

One would rarely implement the net and pairwise force functions this generally. In practice many forces can be trivially determined to be zero for a pair. For example, the spring force between two objects *that are not connected by a spring* is zero, gravity near the Earth's surface can be modeled without an explicit ball model of the Earth, and gravity is negligible between most objects. Other forces only apply when objects are near each other, which can be determined efficiently using a spatial data structure (see Chapter 37). Many of the pairwise force functions don't require all of their parameters. Taking these ideas into account, an efficient implementation of the net force function for a system with a known set of kinds of forces would look something like Listing 35.3.

Listing 35.3: Specialized net force function.

```
// Net force on all objects
Vector3[n] f(float t, Vector3[n] y, Vector3[n] v) {
   Vector3[n] net;

   // for each object
   for (int i = 0; i < n; ++i) {
      net[i] += F_gravity(i) + F_buoyancy(t, y, i);

      for (int j in objectsNeariWithHigherIndex(i, y)) {
         Vector3 fi = F_friction(t, y, v, i, j) + F_normal(y, i, j);
         Vector3 fj = -fi;
         net[i] += fi;
         net[j] += fj;
      }
   }

   // for each pair connected by a spring
   for (int s = 0; s < numSprings; ++s) {
      int i = spring[s].index[0];
      int j = spring[s].index[1];
      Vector3 fi = F_spring(y, v, i, j);
      Vector3 fj = -fi;
      net[i] += fi;
      net[j] += fj;
   }

   return net;
}
```

Note that we use subscripts to denote the *kind* of force pair. For example, the force of gravity is denoted $\mathcal{F}_{\text{gravity}}$. This is a standard notation in physics, although it is unusual in math. It follows our convention that subscripts denoting meaning (like "gravity" or the "i" and "o" for "incoming" and "outgoing") are typeset in roman font, while subscripts indicating indexing are treated as variables and typeset in italic. The "gravity" is part of the name of the function, not a variable. In equations we'll shorten this to just $\mathcal{F}_{\text{g}}$.

Most force functions depend on additional constants that are not explicit arguments. This means that in an actual implementation they would have access to additional information about each object (maybe through applying the provided indices to a global scene array). This is reminiscent of the typical design for a BSDF implementation, where the canonical incoming and outgoing light direction vector arguments are augmented by member variables such as reflectivity and index of refraction.

35.6.4.1 Gravity

Assume that the object with index i is a point mass and that the object with index j is a sphere (or a second point mass). By Newton's law of universal gravitation, the gravitational force experienced by object i (at $\mathbf{y}_i$) due to object j (at $\mathbf{y}_j$) is

$$\mathcal{F}_{\text{g}}(\mathbf{y}, i, j) = G\frac{m_i m_j}{||\mathbf{y}_j - \mathbf{y}_i||^2}\frac{\mathbf{y}_j - \mathbf{y}_i}{||\mathbf{y}_j - \mathbf{y}_i||}. \tag{35.39}$$

This remains a good approximation if the radius of the bounding sphere around each object is small compared to the distance between them (e.g., for computing the forces that planets and stars exert on one another).

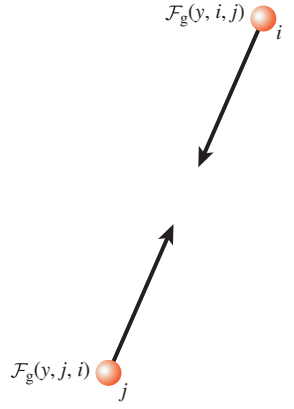


Figure 35.20: Forces due to gravity between two bodies.

When $m_j >> m_i$ and the radius of the bounding sphere of object i is small compared to $||\mathbf{y}_j - \mathbf{y}_i||$, the gravitational acceleration experienced by object i is approximately constant and the gravitational acceleration experienced by object j is negligible. This is the case, for example, when object j is a planet and object i is a human-scale object on the surface of the planet. In this case (Figure 35.21), we can simply ignore j and define the function by

$$\mathcal{F}_g(i) \approx \mathbf{g}m_i, \tag{35.40}$$

which is a good approximation of gravitational attraction, where $\mathbf{g}$ is the acceleration vector. On the surface of the Earth, for example, $||\mathbf{g}|| \approx 9.81\text{m/s}^2$ is a reasonable approximation of the magnitude. The direction is "toward the center of the Earth." Because it is common to work in a local tangent reference frame on the surface of the Earth that neglects planetary curvature, one often calls that direction "down" and assumes that gravity acts along the constant "vertical" axis.

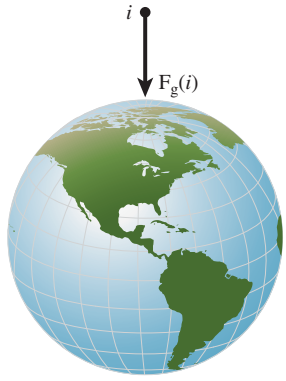


Figure 35.21: Force of gravity on a small object near a planet.

Note that although the gravitational *force* on an object is proportional to its mass, the *acceleration* is independent of its mass. This gives rise to the observation that all objects fall at the same pace, neglecting air resistance and other forces. Specifically, the centers of mass of two bodies that experience only gravitational force will fall at the same pace. Individual points on those objects may experience different acceleration and velocity curves. For example, one end of a tumbling stick may actually have a net upward velocity even while the center of the stick is falling.

Gravity is surprisingly weak compared to other forces. A coin-sized refrigerator magnet is able to resist the gravitational attraction of the entire Earth. A human being can temporarily overcome gravity entirely just by jumping. That gravity would be so weak is a puzzling clue as to the nature of the universe, and there is not yet consensus on why it should be this way. For simulation, the weak-gravity observation yields two practical insights. First, we can neglect gravity except in cases of a tremendous size difference between objects or the absence of almost all other forces. For many applications, this generally means we only care about gravitational attraction to the Earth. Second, we can expect most of the forces in a virtual world to have magnitude on the scale of gravity, because they oppose it and keep objects at rest, or to have magnitude substantially stronger than gravity because they move objects despite gravity. This means that tuning the integrator's constants to be stable for a pile of tumbling blocks is a bad strategy. The force experienced by a rifle bullet when fired or by a car when accelerating can easily be an order of magnitude larger than gravity. *Stability* should be ensured under the largest anticipated force, which is seldom gravity, and then constants should be tuned for *sensitivity* so that gravity can still accelerate an unsupported body from rest.

✓ **THE STRUCTURE PRINCIPLE:** We can generalize this observation into a Structure principle: Treat surprising structural symmetries and asymmetries as both clues about underlying structure..., and warnings to check the robustness of your plan. For example, if otherwise-similar elements differ by orders of magnitude or demand different parameters, as is the case for the fundamental forces of nature, something interesting is going on that can either lead to insight if followed, or bite you if ignored.

35.6.4.2 Buoyancy

Buoyancy arises from pressure within a fluid medium, such as air or water. It is only observed when the medium and objects within it are exposed to a common external acceleration, such as that due to gravity.

Let v_i be[3] the volume (in m^3) of an object i, ρ the density (kg/m^3) of the medium, and $\mathbf{g}$ the net gravitational acceleration (m/s^2) at the location of the object. Assume that $\mathbf{g}$ is constant in the region of interest. Furthermore, assume that both the medium and the object are **incompressible,** meaning that their volumes do not change significantly under pressure. Water is incompressible, as are many of the things that one might expect to find floating in it, such as wood, buoys, boats, fish, and people.

If object i is not in the medium, the buoyancy force that it experiences is obviously $\mathcal{F}_{\text{b}}(\mathbf{x}, i) = 0$ N. If it is submerged (Figure 35.22), then the buoyancy force is

$$\mathcal{F}_{\text{b}}(\mathbf{x}, i) = -\rho v_i \mathbf{g}. \tag{35.41}$$

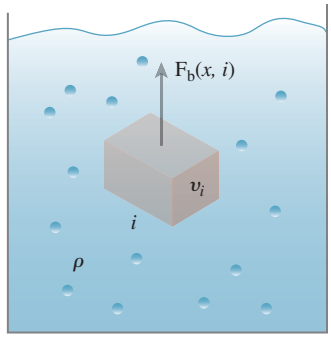


Figure 35.22: Buoyancy on an object with volume v_i submerged in a fluid with density ρ.

Note that the density of the object does not appear. A dense object will experience less upward acceleration against gravity because of the high force of gravity on the object, not because of a reduction in its buoyancy force. Thus, a "buoyant" object counterintuitively experiences no greater "buoyancy" force than a nonbuoyant one.

The medium of course experiences a force of equal magnitude and opposite direction. However, the medium is by definition a fluid composed of individual freely moving particles, so it is internally a very loosely coupled system. In practice, for animation one often deals with very large bodies of fluid and neglects the impact of buoyancy on them.

For a sense of scale, the density of liquid water is about 1,000 kg/m^3 at 4°C, and falls slightly as it is heated or cooled. That this number is exactly a power of ten is no accident—it is one of the constants around which the metric system was originally designed.

Note that the mass and density of the submerged object do not appear. So why do dense objects sink and less dense ones float? Because the *net* force also depends on gravity: $\mathbf{f}_i = \mathcal{F}_{\text{g}}(i) + \mathcal{F}_{\text{b}}$. If the object is dense, then $\mathcal{F}_{\text{g}}(i)$ is large compared to $\mathcal{F}_{\text{b}}$ and a net downward acceleration is observed.

When the fluid is not at rest, different parts of the fluid exert differing pressure on itself and the objects within it. This leads to wave and vortex phenomena, both at the surface and within the fluid.

35.6.4.3 Springs

Consider an ideal spring that has no mass of its own and that always returns to its rest length after being stretched or compressed and then released. Assume that this spring also only expands and contracts along its "length" axis and is perfectly rigid along axes orthogonal to the length.

Let the spring connect objects i and j (Figure 35.23). Let the spring have rest length r, which means that the spring exerts no force on its ends when

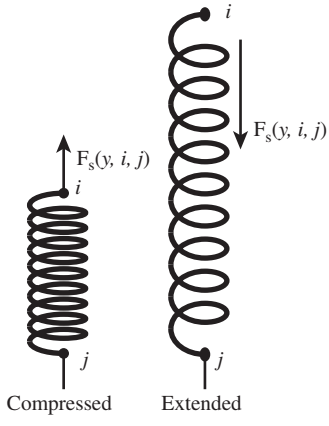


Figure 35.23: Spring force between two bodies.

3. Note that volume v_i is distinct from $\mathbf{v}$ and $\dot{\mathbf{x}}$, the variables used for velocity in this chapter.

$||\mathbf{y}_i - \mathbf{y}_j|| = r$. By Hooke's Spring Law, the restorative spring force experienced by object i is

$$\mathcal{F}_s(\mathbf{y}, i, j) = k_s \left(r - ||\mathbf{y}_i - \mathbf{y}_j||\right) \frac{\mathbf{y}_i - \mathbf{y}_j}{||\mathbf{y}_i - \mathbf{y}_j||}. \tag{35.42}$$

Some springs are stiffer than others. The **spring constant** k_s describes the stiffness of the spring in kg/s^2 (this is another case where subscript s is part of the name, not a proper index). Larger constants describe springs that exert higher restorative forces, which we call stiffer, and smaller constants exert smaller forces, which we call softer. When mass is attached to the spring, a stiffer spring will accelerate that mass faster.

Like a pendulum under gravity, a spring will overshoot the rest position and oscillate. Energy lost to friction within the spring or between the mass and air or a surface will cause it to gradually decelerate and come to rest. For some combination of initial conditions and forces the spring could be critically damped so that it exactly comes to rest without oscillating, but the oscillating behavior is more typical. This means that a stiffer spring will not necessarily resume its rest length sooner than a soft one if disturbed—it may merely oscillate with higher frequency, if the frictional forces are too small.

Oscillators under numerical simulation can increase in energy due to roundoff and integration errors, making them unstable. This is exacerbated by the fact that ropes are often simulated as chains of very stiff springs and cloth as networks of stiff springs. Those yield very large forces and high-frequency oscillations, which require very accurate integration to handle stably.

Most springs in the real world lose significant energy due to their own mass and material deformation. It is common to add an explicit damping term to springs to model that loss. This term has the form of a frictional force because it opposes velocity. The damping term has the same form as the original force term, but it applies to velocity instead of position. This means it acts like a "higher order" spring. This makes the problem of tuning springs for stability easier, but does not eliminate it. In the case of a numerical integrator with fixed time steps, an overly large damping constant can actually increase oscillation rather than reducing it when the integral of the acceleration due to damping exceeds the original velocity.

35.6.4.4 Normal Force

An apple on a table experiences a downward force from gravity and negligible downward buoyancy from the surrounding air. Yet the apple is at rest relative to the table, so there must be a force opposing gravity to prevent the apple from experiencing a net downward force. The source of the force is electrostatic repulsion between the molecules of the apple and the table, which keep their surfaces from interpenetrating. We call this a **normal force** because its axis is perpendicular to the interface between the surfaces. In this case, the interface is the horizontal tabletop, so the force vector experienced by the apple is directed vertically upward along the table's normal vector. When we consider the case of an object on a tilted surface (Figure 35.24), the normal force is directed away from the surface, but not directly opposing gravity, so it can create a net horizontal acceleration.

The force is trivial to describe but challenging to compute: A normal force is as large as it needs to be to prevent interpenetration of (rigid) objects. It has direction along the normal to the interface and magnitude that is equal to the magnitude of the net sum of all *other* forces, projected onto that axis.

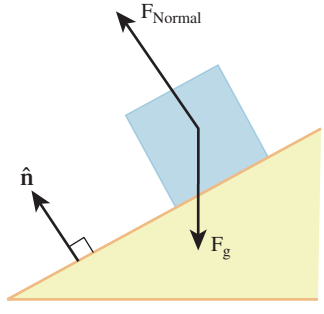


Figure 35.24: The normal force prevents penetration. It is in the direction of the adjacent surface's normal and has magnitude dependent on all other forces.

The normal force is very different from the other forces that we've considered. It depends not only on the state of the system, but also on *the other forces*. For example, when considering spring forces, we can put two masses near each other, attach some springs, and consider their change in velocity and position based solely on the state of the objects. But if we stack ten books on top of a table and ask what the contact forces are between them, then the question doesn't even make sense. The notion of "compute all of the other forces first" implies an acyclic ordering, which doesn't exist in this case. Each book within the stack presses both up and down. Therefore, the friction of the normal force breaks down and there are cases where we simply can't compute reasonable normal forces. Even this simple case of rigid bodies with stacking has proven remarkably challenging—stable, efficient algorithms for solving it were introduced only in the past few years [GBF03, WTF06]. When objects become heavily articulated or have complex contact constraints like the pieces of a completed jigsaw puzzle, simple models like "normal force" break down.

In other words, we've changed modes. We have to model something that is outside of our Newtonian model proper, so we're extending it with a hack. That will be a point where things go wrong. It is ironic that resting contact is surprisingly hard (even before we consider stacking) to simulate compared to the ballistics of a fast-moving bullet. Friction, which relies on normal force, has the same problems. As we discussed in Chapter 1, science depends on the art of modeling what you need for the level of accuracy desired. The simple, first-year Newtonian physics model of normal forces is efficient and accessible, but it is a poor model for complex interactions of many bodies. One can live within those limitations and avoid complex interactions, attempt to patch over the model to hide its failures, or incorporate a more sophisticated model that is probably more expensive to compute and to integrate.

As an example of developing a more sophisticated model, consider that the circular relationships of contact forces are very similar to those studied in light transport. What we need is a steady state solution to an integral equation, and as with light transport there are many mathematical models for numerically approximating that steady state. For this chapter, we will leave this particular problem at that analogy in the interest of returning to efficient simulation of simple systems.

35.6.4.5 Friction and Drag

Friction is a description of a set of forces that lead to the common phenomenon of deceleration. In general, we call any force frictional that is negatively proportional to velocity. This is why friction is described as a force that "opposes motion." However, "motion" depends on your reference frame.

For example, a car could not move or turn without friction. When a car accelerates linearly, the drive shaft rotates the axles, which then rotate the wheels. Friction between the tires covering the wheels and the road resists the rotation of the wheels. This causes the car to move forward relative to the road, or the road to move backward relative to the car, depending on the reference frame we choose. For ideal tires, this completely eliminates motion in the axis along the road between the point of contact of the tire and the road itself. Thus, friction is indeed opposing some motion. However, there's little friction perpendicular to the road, so the point of contact is still able to move outside the plane of the road. In this case, it moves upward a moment after it breaks contact. Ignoring deformation, every point on the tire has instantaneous velocity along a tangent to the tire, in the

rotational plane of the tire. The tire as a whole is driven by the wheel, the axle, and ultimately the drive train and engine to rotate. Because the point of contact resists motion in the plane of the road, that frictional force propagates backward through the system and produces a net linear acceleration of the car along the road. On ice or in mud there is little friction between a tire and the road. In this case, the points on the tire can move freely relative to the road, so the car does not move.

A similar situation applies to turning a car. At the point of contact, high friction along the axis of the axle resists motion along that axis, while lower friction perpendicular to the axis of the car's wheel allows motion. This creates a net rotation of the car. Thus, in the broader sense, friction can be essential for enabling motion.

Frictional forces arise from electrostatic repulsion and attraction. In the repulsion case, molecules of adjacent surfaces collide. As we saw when studying BSDFs, surfaces that are macroscopically planar may be microscopically rough. Thus, what appears to be two smooth surfaces sliding parallel to each other may actually more closely resemble the meshed teeth of two long linear gears. This is why rougher surfaces increase friction. It is also why the sole of a new dress shoe is slippery. That sole is a relatively smooth piece of leather. After the shoe has been worn, say, by walking on concrete sidewalks for a few days, the leather sole exhibits small bumps and cracks from uneven wear. These mesh with bumps on the ground and create greater friction.

In the attraction case, materials of different surfaces chemically bond with each other when in contact, and resist being pulled apart. The bonding is often strongest when the two materials are the same. This is why smooth glass slides easily on smooth wood but is hard to drag across another piece of smooth glass (try this with two juice glasses at breakfast!).

There are case-specific models of friction for commonly arising scenarios. We present two here. **Dry friction** occurs between solids moving parallel to the plane of contact, and **drag** occurs between a solid and a fluid.

In the dry friction model, the force is factored into **static** and **kinetic** (a.k.a. dynamic) terms. Let the two objects be numbered 1 and 2. We consider the force on object 1 in the reference frame the combined system. That is, the center of mass of both objects combined will always be at rest.

Static friction is zero when object 1 is moving, that is, when $\mathbf{x}(t) \neq \mathbf{0}$ in the system's frame. When object 1 is still relative to the reference frame of the system and is in contact with object 2 only along a surface, then we model static friction as able to resist acceleration due to net forces up to

$$k = \mu_{\mathrm{s}} ||\mathcal{F}_{\mathrm{n}}|| \tag{35.43}$$

parallel to the surface. That is, if the net force is $\mathbf{f}$, then the object will experience no acceleration in the plane of the surface if $||\mathbf{f}|| - |\mathbf{f} \cdot \hat{n}| < k$. Otherwise, it will experience some acceleration and kinetic friction as described shortly.

The coefficient of static friction μ_{s} is determined by the chemical composition of the two surfaces, the amount of surface area in contact, and the microgeometry of the surfaces. In other words, there are a lot of hidden parameters in the static friction equation. However, the coefficient is often approximated as a constant material property in simple simulations.

When two objects already have relative velocity with respect to each other in the plane of their contact, we model kinetic friction. Kinetic friction has lower magnitude than the static friction threshold. This is because objects that are

already in motion tend to be bouncing away from their plane of contact at a microscopic scale. Small bounces preclude many of the surface interactions, both chemical and mechanical, that cause friction.

We model kinetic friction (Figure 35.25) as a force

$$\mathcal{F}_{\mathrm{kf}}(t, \mathbf{y}, \mathbf{v}) = -\mu_{\mathrm{k}} ||\mathcal{F}_{\mathrm{n}}|| \frac{\mathbf{v}}{||\mathbf{v}||} \tag{35.44}$$

...but we must be careful in simulation to never reverse direction of velocity in a time step due to frictional forces. This constraint must be enforced by the integrator or at least with knowledge of the integrator's time steps.

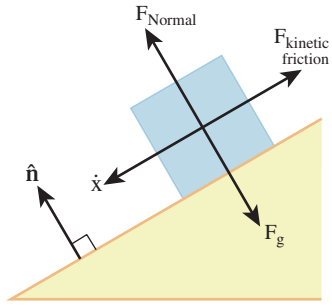


Figure 35.25: Kinetic friction has magnitude proportional to the normal force magnitude and direction opposite velocity (in the plane of the surface).

Drag is the frictional force on a moving object (Figure 35.26) due to the surrounding fluid medium, which is often water or air. Lord Raleigh's model of drag is a force,

$$\mathcal{F}_{\mathrm{d}}(t, \mathbf{y}, \mathbf{v}) = -\frac{1}{2}\rho ||\mathbf{v}|| a C_d \mathbf{v}, \tag{35.45}$$

where a is the surface area of the object presented along the direction of motion; C_d is the drag coefficient, which depends on the shape of the object, its roughness, and the chemical composition of the materials; and ρ is the density of the fluid. As with other frictional forces, we must be careful that the drag force does not reverse the direction of motion during a time step.

Drag force leads to two particularly interesting macroscopic phenomena. The first is **terminal velocity.** An object falling through a medium does not experience continual net acceleration. For example, a skydiver's velocity levels off during freefall. This occurs because at some point $\mathcal{F}_{\mathrm{g}} = -\mathcal{F}_{\mathrm{d}}$, since $\mathcal{F}_{\mathrm{d}}$ is proportional to velocity but $\mathcal{F}_{\mathrm{g}}$ is constant on the object.

Lift is another interesting phenomenon. An airfoil moving through a fluid experiences a net upward force, which allows airplanes and birds to rise in the absence of updrafts. Bernoulli's principle describes how lift arises from variations in drag force over the entire surface. As was the case with a car tire, friction is paradoxically *enabling* motion in this case—just not motion in the direction that the friction opposed.

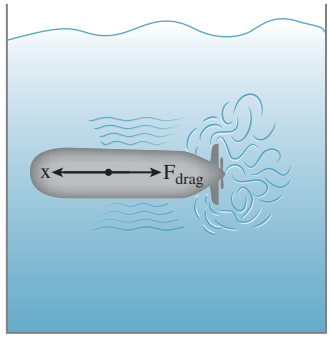


Figure 35.26: Drag forces are caused by friction between an object and the surrounding fluid, and by the pressure built up in the fluid by the object's relative motion and friction within the fluid. Drag forces are hard to model accurately and efficiently because the fluid's behavior is complex and highly dependent on the object's shape at all scales.

35.6.4.6 Other Forces

There are of course other fundamental real-world forces: for example, magnetic, strong and weak nuclear, and electrical. These can be adopted from a physics textbook in the same manner as the ones described here. There are also nonfundamental forces that are useful modeling techniques, such as tension and compression, that can be found in any mechanical engineering text.

Nonphysical forces, such as tractor and repulsor beams in a science fiction context, can be described using arbitrary functions. Any force desired can be inserted into the simulation framework, since Newton's first law of motion reduces it to an acceleration that can be inserted into the integrator.

35.6.5 Particle Collisions

35.6.5.1 Collision Detection

For particles with very small cross sections, collision detection is equivalent to ray tracing. We can ignore the collisions between particles because the probability of those collisions occurring is vanishingly small. The path of a particle over a

sufficiently small time interval is a line segment, so by tracing a ray from the particle's origin in the direction of its velocity we can detect collisions with other parts of the scene. All of the data structures and algorithms built for tracing light rays can thus be applied to dynamics simulation.

It is also possible to model collisions between particles with volume while simulating the movement of each particle as if it were a point mass. To detect a general intersection between two particles we choose the reference frame of one (so that it is not moving), and then extrude the other one along the relative velocity vector. For particles with spherical bounds the intersection test is relatively simple because it is equivalent to testing whether a point is inside a capsule (a cylinder with a hemispherical cap on each end). Collisions between other shapes are often more difficult to compute, and the best solution can be to use a spatial data structure to detect intersecting polygons (see Chapter 37).

35.6.5.2 Normal Forces through Transient Constraints

After a collision is detected, the system must respond to prevent particles from penetrating the scene. We've already seen one response—normal forces. When a collision occurs between objects with very low velocity toward each other, it is likely due to resting contact. In this case, reacting to the collision by reversing velocity might introduce energy into the system, since the "collision" itself is an artifact of limited numerical precision for times and positions: Two objects in resting contact should never have accelerated toward each other in the first place. We can restore stability by explicitly moving the objects out of contact or applying normal forces to prevent penetration before it occurs.

We've seen that normal forces can be tricky to apply because they must occur "after all other forces" . . . which leads to an awkward ordering constraint when normal forces from multiple surfaces are in effect. In systems with a sophisticated integration scheme, it is possible to apply normal forces implicitly rather than explicitly. In these cases, contact creates a temporary "joint" between two objects, requiring that the solution to the simulation during the time step must not move the point of contact on either object [Lö81].

35.6.5.3 Penalty Forces

A generalization of normal forces can elegantly resolve collisions between objects with significant velocities. A **penalty force** is something like a normal force. It applies to objects that are interpenetrating. The magnitude of a penalty force increases with the level of penetration, and the direction is that which will most quickly separate them. This is actually a reasonable model for microscopic electrostatic repulsion. Atoms never really contact one another. As they come close, they repulse each other with increasing strength until the relative velocities reverse and the proximity decreases. When the force is integrated over the time period from when the penalty force first becomes significant to when it again becomes insignificant, the net result is that of an elastic collision: The velocities of the objects have been reflected about the normal to the plane of their "collision."

Penalty forces are simple to compute and apply, but often difficult to tune. If the penalty force is too weak, then objects will interpenetrate too far before rebounding. This gives the appearance of overly rubbery materials. If the penalty force is too strong, then any inaccuracy in the integration process can destabilize the system by creating energy due to undersampling. When there are multiple potential points of contact, uneven application of penalty forces due to differing

depths of penetration can create a net rotation. With sufficiently large penalty forces this is a further potential for destabilization in the system as small translational error may lead to large rotational error.

35.6.5.4 Impulses

The problem with penalty forces is that they take time to apply, which makes their effectiveness sensitive to the numerical integration scheme of a dynamics simulator. **Impulses** are an alternative collision resolution strategy that avoids this by directly and instantaneously changing velocities outside of the numerical integration process. This is a good example of the principle of knowing the limits of a model and changing models, rather than patching, when those limits are exceeded.

Numerical integration of forces is stable for constant or slowly varying accelerations, such as those due to gravity, springs, and buoyancy. Numerical integration becomes more fragile for forces with dependency cycles or those that depend on velocity, such as normal and friction forces. For accelerations that vary over time scales shorter than the simulation interval, such as penalty forces, numerical methods generally fail. One approach is to try ever-shorter simulation intervals or more sophisticated integrators, but that is merely shifting the limitation of the model rather than removing it. By pausing numerical integration, directly manipulating velocities with momentum impulses, and then restarting numerical integration, we escape the inherent limitations of integrating derivatives. Note that there is an analogous situation in light transport. In that context, an "impulse" is a spike in a probability distribution function, such as created by a directional light source or a perfect specular reflection. Integrating those zero-area phenomena is just as unstable as integrating a zero-time force. So, stable renderers explicitly handle specular reflections and directional sources with absolute probabilities rather than attempting to make very small angular measurements of very high-magnitude probability distribution functions. In dynamics, we will directly apply velocity changes rather than trying to integrate large-magnitude derivatives of velocity over small-scale time intervals.

Concretely, an *instantaneous* change means that the integral of acceleration ($\ddot{\mathbf{x}}$) over a *zero*-second time interval is nonzero; therefore, $\ddot{\mathbf{x}}$ is infinite. So we're applying infinite force for zero time. Specifically, the force and acceleration functions contain mathematical impulses. We can't directly represent impulses within the integration framework because presenting an infinite acceleration to the numerical integrator will simply result in infinity $\times$ 0 = "not a number" under floating-point representations, and because any attempt to perform higher-order integration on such quantities will simply propagate the "not a number" value further. So we step outside the integrator. This of course is a source of instability and error. Anytime the integrator can't observe a value or control an operation, it becomes vulnerable to roundoff and approximation errors in that operation.

The impulse equations are derived from the physical concept of linear momentum, which is simply

$$\mathbf{p}_i(t) = m_i \dot{\mathbf{x}}(t). \tag{35.46}$$

The change in velocity at a collision will therefore be

$$\Delta \dot{\mathbf{x}}_i(t) = \frac{\Delta \mathbf{p}_i(t)}{m_i}. \tag{35.47}$$

The distinction between $\Delta\dot{\mathbf{x}}(t)$ and $\ddot{\mathbf{x}}(t)$ is important. We have a rigorous definition of the acceleration function as a derivative:

$$\ddot{\mathbf{x}}(t) = \lim_{\Delta t \to 0} \frac{\dot{\mathbf{x}}(t + \Delta t) - \dot{\mathbf{x}}(t)}{\Delta t} = \lim_{\Delta t \to 0} \frac{\dot{\mathbf{x}}(t) - \dot{\mathbf{x}}(t - \Delta t)}{\Delta t}. \tag{35.48}$$

In the presence of an impulse at time t_0, $\dot{\mathbf{x}}(t)$ is discontinuous at t_0. This means that the lower and upper limits are not equal, so $\dot{\mathbf{x}}(t)$ *is not differentiable at* t_0. In other words, $\ddot{\mathbf{x}}(t_0)$ is not defined (which makes sense, since it is "infinite," yet integrates to a finite quantity).

Physicists often employ a Dirac delta "function" $\delta(t)$ for which

$$\int_{-\infty}^{+\infty} \delta(t)dt = 1 \text{ and} \tag{35.49}$$

$$\delta(t) = 0 \; \forall \; t \neq 0 \tag{35.50}$$

as a notational tool for assigning a value to $\ddot{\mathbf{x}}(t_0)$ (e.g., $\ddot{\mathbf{x}}(t) = \delta(t - t_0)\Delta\dot{\mathbf{x}}(t)$). However, beware that $\delta(t)$ is not actually a proper function, and that this notation conceals the fact that differential and integral calculus cannot represent $\ddot{\mathbf{x}}(t)$.

Since the rest of our simulation is based on numerical integration of estimated derivatives, concealing the facts that $\dot{\mathbf{x}}$ is not differentiable at t_0 and $\ddot{\mathbf{x}}$ is not integrable at t_0 is a dangerous practice. We therefore accept that $\ddot{\mathbf{x}}(t_0)$ does not exist and define a closely related function that does generally exist:

$$\Delta\dot{\mathbf{x}}(t) = \dot{\mathbf{x}}(t)^+ - \dot{\mathbf{x}}(t)^-, \tag{35.51}$$

where the sign superscripts denote the single-sided limits

$$\dot{\mathbf{x}}(t)^- = \lim_{\Delta t \to 0} \dot{\mathbf{x}}(t - \Delta t) \tag{35.52}$$

$$= \lim_{\Delta t \to 0} \frac{\mathbf{x}(t) - \mathbf{x}(t - \Delta t)}{\Delta t} \tag{35.53}$$

and

$$\dot{\mathbf{x}}(t)^+ = \lim_{\Delta t \to 0} \dot{\mathbf{x}}(t + \Delta t) \tag{35.54}$$

$$= \lim_{\Delta t \to 0} \frac{\mathbf{x}(t + \Delta t) - \mathbf{x}(t)}{\Delta t}. \tag{35.55}$$

In plain language, a plus sign denotes a value immediately after a collision and a minus sign denotes a value immediately before a collision. The value at the time of the collision is irrelevant, since we're using the impulse to resolve the collision and jump from the before value to the after value.

The advantage of moving from velocity to momentum is that the **law of conservation linear momentum** in physics states that the momentum of a closed system is always constant. Therefore,

$$\sum_i \mathbf{p}_i^+(t) = \sum_i \mathbf{p}_i^-(t). \tag{35.56}$$

For a system with exactly two objects whose indices are i and j, the law expands to

$$\mathbf{p}_i^-(t) + \mathbf{p}_j^-(t) = \mathbf{p}_i^+(t) + \mathbf{p}_j^+(t_0), \tag{35.57}$$

$$\mathbf{p}_i^+(t) - \mathbf{p}_i^-(t) = -\left(\mathbf{p}_j^+(t) + \mathbf{p}_j^-(t)\right), \text{ and} \tag{35.58}$$

$$\Delta\mathbf{p}_i(t) = -\Delta\mathbf{p}_j(t), \tag{35.59}$$

which indicates that we only need to solve for the momentum change of one object explicitly.

Now consider a system with more objects. If we assume that all collisions are pairwise and that exactly one collision occurs at a given time t_0, then we can ignore the change in momentum of all objects except i and j and still apply Equation 35.59. This also requires that we be able to assign a strict ordering on multiple collisions. When a simulator encounters a multibody collision the assumption has been violated, and assigning an artificial ordering creates instability. For example, each internal object in a stack continuously collides with two other objects, so it is not surprising that impulse-based simulators often are unstable in the presence of stacked objects.

The linear momentum change in a collision is given by [BW97a]

$$\Delta\mathbf{p}_i(t_0) = \hat{n}\frac{(1+\varepsilon_{i,j})\hat{n}\cdot(\dot{\mathbf{x}}_i^-(t_0) - \dot{\mathbf{x}}_j^-(t_0))}{\frac{1}{m_i}+\frac{1}{m_j}} \quad \text{and} \quad \Delta\mathbf{p}_j(t_0) = -\Delta\mathbf{p}_i(t_0), \tag{35.60}$$

where $\varepsilon_{i,j}$ is the **coefficient of restitution** (a measure of how much they resist interpenetration) between objects i and j, $\hat{n}$ is the unit normal to the contact plane, and m is mass. ($\Delta\mathbf{p}_i(t_0)$ is often denoted $\mathbf{j}_i$.)

One nice property of this equation is that it is expressed in terms of inverse masses, rather than masses. This means that the limit of $\Delta\mathbf{p}$ as one of the masses goes to infinity exists and is finite. In practice, it is useful to pin certain objects so that they are not affected by collisions. For example, in a human-scale simulation, simulating the Earth, or even a building as having infinite mass introduces no significant error and typically leads to a simpler implementation of these immovable objects. Beware that assigning infinite mass to a moving object can lead to undesirable behavior, since that object's momentum is then also infinite. For example, if a train following a prescripted spline strikes a car simulated by dynamics, that car will receive infinite momentum in the collision.

A common model is $\varepsilon_{i,j} = \min(\varepsilon_i, \varepsilon_j)$, where the single-object coefficients are chosen to such that more deformable materials have lower ε. In a perfectly inelastic collision, the objects stick together after collision.

35.6.6 Dynamics as a Differential Equation

Digital computers have limited precision. Therefore, with every iteration of the Heun-Euler Equations 35.37 and 35.38, some error will be introduced. The longer we simulate, the less predictable, and potentially, the less accurate, our solution will be. Of course, unless the forces really were piecewise-constant, approximating them as piecewise-constant is another source of error. Smaller time intervals will lead to finer sampling of the forces and therefore improve accuracy. However, we need many more iterations to span a given time period with small intervals, so there will be more computation and more accumulated error. We now consider numerical methods for solving for $\mathbf{x}$ that relax the assumption of piecewise-constant force. These allow fairly large time intervals without undersampling force, which can mitigate the accumulated error problem without increasing the net amount of computation.

Let the function describing the state of the universe at time t be

$$\mathbf{X}(t) = \begin{bmatrix} \mathbf{x}(t) \\ \dot{\mathbf{x}}(t) \end{bmatrix}. \tag{35.61}$$

Although $\mathbf{X}$ will be a computationally convenient representation, there is also some physical motivation for it. Both position and velocity (as a proxy for momentum) are included in the state vector because they are properties of objects that appear in mechanical models of the real world. The laws of physics tell us how to compute forces, which are proportional to acceleration. The inputs to force functions are always position and velocity. Forces never turn out to be proportional to acceleration, so we don't explicitly store acceleration in the state. The exception is the "normal" force model, for which we have already described the problems arising from including acceleration as an input.

Note that $\mathbf{X}(t)$ describes the second and third parameters of the $\mathbf{f}$ and impulse functions; we can redefine them to take only two parameters and thus write

$$\dot{\mathbf{X}}(t) = \begin{bmatrix} \dot{\mathbf{x}}(t) \\ \ddot{\mathbf{x}}(t) \end{bmatrix} = \begin{bmatrix} \mathbf{X}(t)[n+1..2n] \\ \frac{\mathbf{f}(t,\mathbf{X}(t))}{m} + \frac{d\mathbf{j}(t,\mathbf{X}(t))}{dt} \end{bmatrix}. \tag{35.62}$$

When considering multiple particles with varying mass, we could replace $\mathbf{f}/m$ with $\mathbf{f} \cdot \mathbf{M}^{-1}$ for some diagonal matrix of masses $\mathbf{M}$. Note that differential Equation 35.62 is the system version of the single-particle relation previously introduced in Equation 35.22.

Chapter 29 framed light transport in the rendering equation, which was an integral equation. As was the case there, considering general-purpose numeric methods will lead us to broader computer science than needed for the specific dynamics problem at hand. This has two advantages. We can build on previous work that is not graphics-specific, including not just mathematics but also numerical ODE-solving software libraries. We can also take the algorithms that we develop and apply them to other problems beyond dynamics, both in computer graphics and in other fields.

35.6.6.1 Time-State Space

Let's look at a concrete example to gain some intuition for the $2n+1$-dimensional time-state space defined by $t, \mathbf{X}(t)$.

Figure 35.27 shows the path (described by its y-coordinate) of a cannonball that is modeled as a particle in a 1D universe. The upper-left plot shows the familiar time-space path plot of $\mathbf{x}(t)$ versus t, which is a parabola. The upper-right plot shows the time-velocity path plot of $\dot{\mathbf{x}}(t)$ versus t, which is linear because the ball experiences constant negative acceleration. This path crosses $\dot{\mathbf{x}}(t) = 0$ at the apex of the cannonball's flight. The lower-left image combines these to show the time-state path plot of $\mathbf{X}(t)$ versus t. Keep in mind that $\mathbf{X}$ describes the entire universe, not just one object. In this example, they happen to be the same because we're considering a universe with a single particle. Were there more particles, the plot would have many more dimensions.

The thick line in the lower-left subfigure shows one particular $\mathbf{X}$ of the many that could exist. The thin lines are its projection onto the $(t, \mathbf{x})$- and $(t, \dot{\mathbf{x}})$-planes as "shadows" for reference. Those shadows are exactly the upper-left and upper-right figures.

From the perspective of $t = 0$, the dark $(t, \mathbf{X}(t))$-curve describes the entire future fate of the universe. From the perspective of some point at the high end of the timeline, this is the history of the universe. But that curve is only one possible reality. Were there different initial conditions, $\mathbf{X}$ would have been a different function and traced a different curve.

The lower-right subfigures depicts three alternative curves that *could have been* solutions to $\mathbf{X}$, given different initial conditions. Here only the shadows on

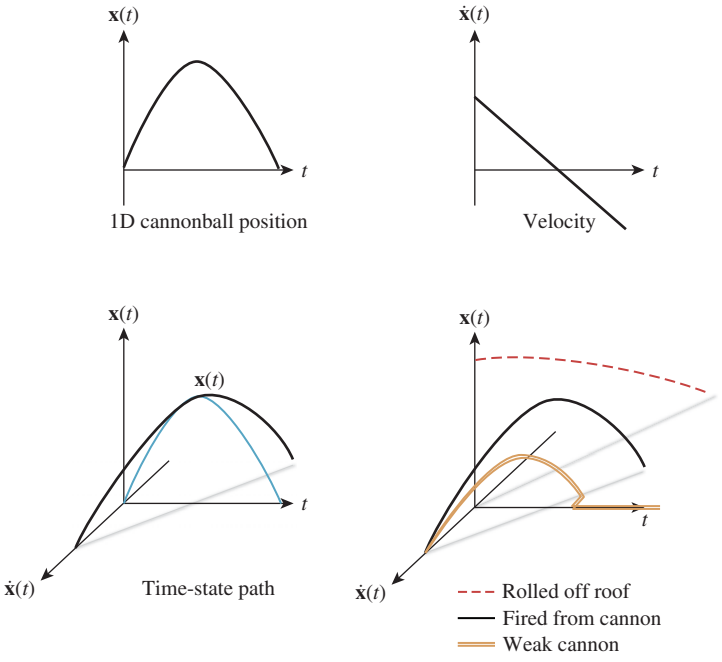


Figure 35.27: Cannonball path in time-state space.

the $(t, \dot{\mathbf{x}})$-plane are shown to keep the diagram simpler. One thick curve is our original cannonball, another is a cannonball rolling off a roof, where $\dot{\mathbf{x}}(0) = 0$ and $\mathbf{x}(t) > 0$; and the third is a cannonball fired from a weak cannon so that the initial velocity is small but nonzero. That cannonball strikes the ground early and has zero velocity and position after collision. There are infinitely many other solutions for $\mathbf{X}$ in Equation 35.62, depending on the initial state.

By definition, the initial state of the universe is entirely identified by the value that we choose for $\mathbf{X}(0)$. There is nothing special about $t = 0$. Ignoring forces from human interaction, for any particular t_i the entire future solution for $t > t_i$ is determined by $\mathbf{X}(t_i)$. In other words, if you give me an $\mathbf{X}(0)$, I can tell you $\mathbf{X}(t)$ for all subsequent t. My response is a curve in $(t, \mathbf{X})$-space. If you give me a different $\mathbf{X}(0)$, I'll give you a different curve. Every point lies in space on one of these many possible curves. Furthermore, the curves aren't merely geometry: They are parametric curves, with parameter t. That is why it makes sense to compute $\dot{\mathbf{X}}(t)$.

Inline Exercise 35.3: This exercise is mandatory. Don't progress until you've completed it.

We've just discussed the initial conditions that give rise to three different $(t, \mathbf{X})$ curves shown in the lower-right subfigure of Figure 35.27. Make a copy by hand of that lower-right subfigure. Think about how we drew the curves and the equations that govern them—don't just copy the shape.

Now, consider a new scenario. Someone takes the (ill-advised) action of leaning out a window halfway up the building with a cannon, points it straight upward, and fires off a cannonball. Write down the equation of motion as a function $\mathbf{X}(t)$ and draw the trajectory in time-state space superimposed on the previous curves.

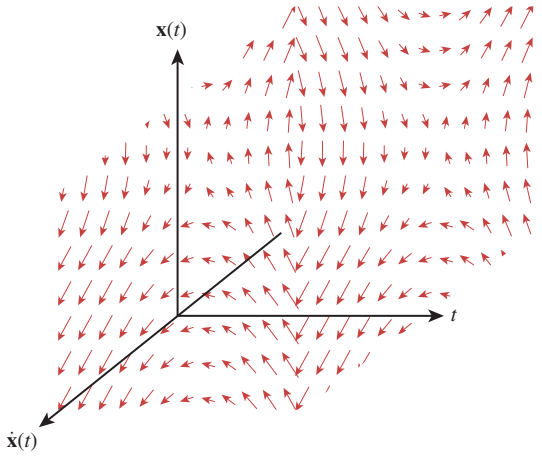


Figure 35.28: Visualization of **D** *for the state space of one 1D particle under some arbitrary forces.*

Consider the tangents to solution curves for t versus $\mathbf{X}$, like those in Figure 35.27. They define some vector field over the time-state space such that if the universe we are simulating has solution $\mathbf{X}$, the value of this vector field is $\dot{\mathbf{X}}(t)$ at $(t, \mathbf{X}(t))$. Formally, let $\mathbf{D}$ be the vector-valued function such that

$$\mathbf{D}(t, \mathbf{Y}) = \dot{\mathbf{X}}(t) \text{ if } \mathbf{Y} = \mathbf{X}(t), \forall \text{ physically possible } \mathbf{X}. \tag{35.63}$$

In the context of physical simulation under Equation 35.62, this means that

$$\mathbf{D}(t, \mathbf{Y}) = \begin{bmatrix} \mathbf{Y}[n+1..2n] \\ \frac{\mathbf{f}(t,\mathbf{Y})}{m} + \frac{\mathbf{j}(t,\mathbf{Y})}{\Delta t} \end{bmatrix}, \tag{35.64}$$

although in the following derivation we will make no assumptions about $\mathbf{D}$, $\mathbf{X}$, or $\dot{\mathbf{X}}$, so as to keep the discussion valid for all ordinary differential equations. From here on, these will simply be arbitrary functions, which may not even be vector-valued, and our goal is to trace a flow curve through the tangent field.

We chose the letter "D" because the tangent field function is reminiscent of the derivative $\dot{\mathbf{X}}$. It is not actually the derivative: $\mathbf{D}$ is a field over time-state space (as shown in Figure 35.28) and $\dot{\mathbf{X}}$ is the tangent function to one particular curve $\mathbf{X}$ through that space. That is, every solution $\mathbf{X}$ is a **flow curve** of the field $\mathbf{D}$.

If we were following a particular solution $\mathbf{X}$ and somehow stepped off that curve, the $\mathbf{D}$ field would guide us along some other flow curve that quite likely diverged from our original solution. Figure 35.29 depicts two instances of this situation for a simple tangent field. Although it is undesirable, this will prove to be unavoidable during our numerical evaluation, and the best that we can do is to try to minimize divergence and encourage transitions that are at least plausible.

35.6.6.2 Following the Tangent Field

Consider a series of **state vectors** $\{\mathbf{Y}_1, \mathbf{Y}_2, \ldots\}$ such that $\mathbf{Y}_i \approx \mathbf{X}(t_i)$, where the sampling period $t_{i+1} - t_i = \Delta t$ is constant. These are the regularly spaced samples of the state function $\mathbf{X}$ that we want the simulation system to provide.

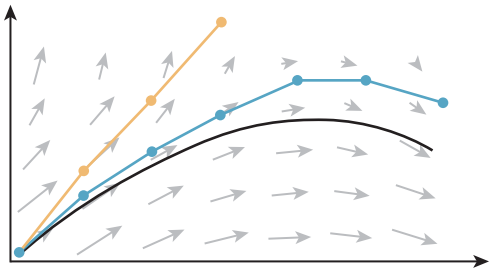


Figure 35.29: A flow curve through a tangent field, and two attempts to follow it in discrete steps.

To find these, we begin with an exact initial state $\mathbf{Y}_1 = \mathbf{X}(t_1)$. We want to follow the flow curve of $\mathbf{D}$ that passes through $(t_1, \mathbf{Y}_1)$ until it intersects the plane $t = t_2$. That intersection is at $\mathbf{Y}_2$. We then repeat the process for every other sample. This strategy evaluates $\mathbf{X}$ by numerical integration of the $\dot{\mathbf{X}}$ function implicit in $\mathbf{D}$, rather than solving for an explicit expression for $\mathbf{X}$ or $\dot{\mathbf{X}}$.

The process is a little like a detective tailing a suspect in a black sedan in city traffic. The detective doesn't want to be observed. So he stays a few cars behind on the road, out of the suspect's line of sight. But he also doesn't want to lose the suspect, so he has to periodically pull up closer to catch sight of the sedan. Here, the sedan's path is the solution $\mathbf{X}$ that we (and the detective) want to follow, and checking the location of the sedan is like evaluating $\mathbf{D}$. The detective's actual path is defined by the $\mathbf{Y}$ values, which we want to follow $\mathbf{X}$, but might diverge if he isn't careful. Unfortunately for our detective, there are a lot of other black sedans on the road that don't contain the suspect—and for our dynamics solver there are other solutions to $\mathbf{X}$ that don't match our initial conditions. If the detective waits too long between checking on his suspect, he might accidentally start following the wrong sedan. In the equivalent situation, our solver might start tracking the wrong solution.

The previously explored Heun-Euler solver in Equations 35.37 and 35.38 made the assumption that the $\mathbf{D}$ field described linear flows over the time intervals. This is like pausing exactly once per time interval, choosing a direction, and then following the parabolic curve that exits our current position in that direction. We could do worse; for example, following a line that exits our current position in that direction. That process is called Explicit Euler integration. We can also do much better than either Heun-Euler or Explicit Euler, in terms of estimator quality versus computation.

To see how, we stretch our car-chase analogy a little further. The detective has more choices than just driving straight toward where he last saw the sedan. He can watch the sedan for a while (i.e., evaluate $\mathbf{D}$ at multiple points), and then make an educated guess about where it is really going before dropping back out of sight. We'll cover strategies for this in Section 35.6.7. First let's look at how we will employ such a strategy in terms of a general dynamics solver.

35.6.6.3 A General ODE Solver

The framework for the numerical method for evaluating solutions to ODEs at specific time intervals is extremely simple. Function $\mathbf{D}$ is implemented as a first-class closure. The mechanism for this depends on the language: a class in C#,

C++, or Java, a function in Python, Scheme, or Matlab, and a function pointer and `void` pointer in C.

The integrator that advances $\mathbf{Y}_i$ to $\mathbf{Y}_{i+1}$ is shown in Listing 35.4.

Listing 35.4: ODE integrator using Explicit Euler steps.

```
vector<float> integrate
  (float t,
   vector<float> Y,
   float Δt,
   vector<float> D(float t, vector<float> Y) {

   // Insert your preferred integration scheme on the next line:
   ΔY = D(t, Y) · Δt

   return Y + ΔY
}
```

We will replace the center line with alternate expressions in what follows. The method shown is the least computationally expensive and least accurate reasonable method for evaluating ΔY.

35.6.7 Numerical Methods for ODEs

There is a tradeoff between the computational cost of a single iteration and the number of iterations required to maintain accuracy. In general, one is willing to perform significantly more work per iteration to take large time steps. However, for graphics applications the time step may be driven by the rendering or user input rate, and therefore there is a limit to how large it can be made. Furthermore, the **D** function may be very expensive to evaluate because it can include collision detection and a constraint solver. Thus, although for general applications the 4th-order Runge-Kutta scheme is the default choice, many dynamics simulators for real-time computer graphics still rely on simple Euler integration.

The basic idea underlying all of the integration schemes presented in this chapter is that any infinitely differentiable function f can be expressed as the Taylor polynomial,[4]

$$f(t) = \lim_{N\to\infty} \sum_{n=0}^{N} \frac{d^n f(t_0)}{dt^n} \frac{(t-t_0)^n}{n!} \tag{35.65}$$

$$= f(t_0) + \dot{f}(t_0)(t-t_0) + \ddot{f}(t_0)\frac{(t-t_0)^2}{2} + \dddot{f}(t_0)\frac{(t-t_0)^3}{6} + \ldots \tag{35.66}$$

A finite number of Taylor terms allows approximation of $f(t)$ for arbitrary t. To apply this, we must know the value of the function $f(t_0)$ at some specific time t_0, and some derivatives of f at t_0. For the dynamics application of approximating $\mathbf{Y} \approx \mathbf{X}(t)$, we know $\mathbf{X}(t_0)$ because that is the current state of the system. We also know $\dot{\mathbf{X}}(t_0)$ because that is the velocity and acceleration, which can be computed by the **D** function. We can therefore immediately make the two-term approximation:

4. This is only guaranteed to hold on some neighborhood of t_0. In practice, this neighborhood is usually the whole real line (for functions which mathematians call "analytic"), but there exist functions for which the neighborhood is very small. Impulses are one case where the neighborhood is *not* the entire real line, which gives more intuition as to why they do not interact well with the ODE framework for dynamics.

$$\mathbf{X}(t) \approx \mathbf{X}(t_0) + \dot{\mathbf{X}}(t_0)(t - t_0). \tag{35.67}$$

The highest-order term present in this approximation is the $n = 1$ term, so this is called a first-order approximation. This particular first-order approximation leads to the integration scheme just presented, and is known as **Forward Euler integration.**

The error in an approximation of a Taylor polynomial with a finite number of terms is the missing higher-order terms. If the magnitude of higher-order derivatives grows no faster than the factorial, higher-order terms have successively less impact. This is true for many functions encountered in practice, especially physical simulation without impulses. The error in a first-order approximation is therefore dominated by a second-order term. That term contains $(t - t_0)^2$, so for a time step $\Delta t = t - t_0$, it is often referred to as growing like $\mathrm{O}(\Delta t^2)$. This is a bit misleading. For large n, the factorial in the denominator decreases the magnitude of the higher-order terms faster than the exponent on Δt. Furthermore, for $\Delta t > 1$, Δt^n *increases* with order, and the point at which that occurs depends on the arbitrary choice of units for measuring time (although the derivative also changes to counteract this, as we'll see in a moment). So a better way to think about the accuracy of an approximation is that an "nth-order approximation" contains n terms of the Taylor expansion.

In a dynamics system we do not have an explicit function for higher-order derivatives of $\mathbf{X}$. However, by applying the definition of the derivative to $\mathbf{D}$, we can estimate higher-order derivatives for specific values of t. For example:

$$\ddot{\mathbf{X}}(t) = \lim_{t \to t_0} \frac{\mathbf{D}(t, \mathbf{X}(t)) - \mathbf{D}(t_0, \mathbf{X}(t_0))}{t - t_0}. \tag{35.68}$$

We can approximate the limit by choosing any $t > t_0$, with accuracy typically decreasing at larger time steps. Recursively applying the numerical derivative form allows estimation of arbitrarily high derivatives. Note that the nth derivative contains $(t - t_0)^n$ in its denominator, which will be canceled by the same coefficient in the numerator of the Taylor polynomial. Thus, many numerical integrators don't explicitly contain higher powers of the time step in their final form.

The challenge in estimating the numerical derivatives is that we can't apply the $\mathbf{D}$ function at t_0 unless we know $\mathbf{X}(t_0)$. Since we're trying to solve for $\mathbf{X}(t_0)$, this means we can't solve until we already know the solution. Fortunately, we can *estimate* $\mathbf{X}(t_0)$ with some $(n - 1)$th-order method. We can then use that value to numerically estimate an nth-order derivative. To prevent the error from these successive estimations from accumulating, many schemes then go back and reestimate $\mathbf{X}(t_0)$.

One can estimate arbitrarily high derivatives with arbitrary accuracy by this method. Increasing accuracy in the derivatives allows larger time steps, which allows fewer computations per unit simulation time. However, there is a computational cost for each $\mathbf{D}$ evaluation. If the net cost of computing an accurate derivative to take a large time step is higher than that of computing a less accurate derivative and making many small time steps, nothing has been gained. The choice of what order of integrator to apply therefore depends on the cost of $\mathbf{D}$, which depends on the complexity of the scene and forces. This is why there is no single "best" integration scheme for dynamics.

35.6.7.1 Explicit Forward Euler

We've already seen the Explicit (a.k.a. Forward) Euler method,

$$\Delta \mathbf{Y} = \mathbf{D}(t, \mathbf{Y}) \cdot \Delta t, \tag{35.69}$$

in which $\mathbf{Y}$ is the initial state of the system, $\mathbf{D}$ is the field for which $\mathbf{D}(t_0, \mathbf{Y}) \approx d\mathbf{Y}(t_0)/dt$, and Δt is the duration of the time interval.

This is perhaps the simplest and most intuitive integrator. The change in state of the system is our estimate of the current derivative scaled by the time step. For position, this corresponds to assuming that velocity is constant (i.e., acceleration is zero) and thus advancing by the current velocity. It obviously is a good estimator when those conditions are true and is a worse estimator when there is significant acceleration.

35.6.7.2 Semi-Implicit Euler

Explicit Euler integration only considers the derivative (e.g., velocity) at the beginning of a time step. Under large accelerations, that is a poor estimate for the derivative throughout the time step. An alternative is to use the derivative from the *end* of the time step. The challenge in doing so is that since we're trying to estimate the state of the system at the end of the time step, it is hard to use values from that time to reach it.

The Semi-Implicit Euler (confusingly also known as Semi-Explicit Euler) method performs a tentative Explicit Euler step to the end of a time interval and measures the derivative there. It then returns to the beginning of the interval and applies the discovered derivative to the original position. This is more stable than Explicit Euler and much less expensive than the full Implicit Euler method, which requires iterating on this process to find a fixed point [Par07].

Expressed in our state notation, the Semi-Implicit Euler integrator is

$$\text{Let } \mathbf{Z} = \mathbf{Y} + \mathbf{D}(t, \mathbf{Y}) \cdot \Delta t; \tag{35.70}$$

$$s = t + \Delta t; \tag{35.71}$$

$$\Delta \mathbf{Y} = \mathbf{D}(s, \mathbf{Z}) \cdot \Delta t. \tag{35.72}$$

35.6.7.3 Second-Order Runge-Kutta

Runge-Kutta[5] refers to a family of related iterative methods for approximating solutions to ODEs. They are described by a set of coefficients and the number of stages, where each stage requires one evaluation of $\mathbf{D}$ based on the result of previous evaluations. See [Pre95] for the general form of Runge-Kutta methods.

Explicit Euler is the only one-stage Runge-Kutta method. A commonly employed one of the many possible two-stage Runge-Kutta methods is

$$\Delta \mathbf{Y} = \mathbf{D}\left(t_i + \frac{\Delta t}{2}, \mathbf{Y}_i + \frac{\Delta t}{2}\mathbf{D}(t_i, \mathbf{Y}_i)\right) \cdot \Delta t. \tag{35.73}$$

35.6.7.4 Heun

Heun's method is an improved two-stage Runge-Kutta method compared to the one just presented. It is equivalent to averaging the steps from the Explicit and

5. Phonetically: "run-ga cut-a"

Semi-Implicit Euler methods, and to the scheme previously presented in Equations 35.37 and 35.38. Heun's method is also known as the **Modified Euler** and **Explicit Trapezoidal** methods.

$$\text{Let } \mathbf{Z} = \mathbf{Y} + \mathbf{D}(t, \mathbf{Y}) \cdot \Delta t \tag{35.74}$$

$$s = t + \Delta t \tag{35.75}$$

$$\Delta \mathbf{Y} = \frac{\mathbf{D}(t, \mathbf{Y}) + \mathbf{D}(s, \mathbf{Z})}{2} \cdot \Delta t \tag{35.76}$$

35.6.7.5 Explicit Fourth-Order Runge-Kutta

The classic fourth-order Runge-Kutta method is one of the most commonly used integrators for dynamics in computer graphics and is the most accurate integrator in this chapter. It is:

$$\mathbf{K_1} = \mathbf{D}(t, \mathbf{Y}) \tag{35.77}$$

$$\mathbf{K_2} = \mathbf{D}(t + \frac{\Delta t}{2}, \mathbf{Y} + \mathbf{K}_1 \cdot \frac{\Delta t}{2}) \tag{35.78}$$

$$\mathbf{K_3} = \mathbf{D}(t + \frac{\Delta t}{2}, \mathbf{Y} + \mathbf{K}_2 \cdot \frac{\Delta t}{2}) \tag{35.79}$$

$$\mathbf{K_4} = \mathbf{D}(t + \Delta t, \mathbf{Y} + \mathbf{K}_3 \cdot \Delta t) \tag{35.80}$$

$$\Delta \mathbf{Y} = \frac{1}{6} (\mathbf{K}_1 + 2\mathbf{K}_2 + 2\mathbf{K}_3 + \mathbf{K}_4) \cdot \Delta t \tag{35.81}$$

One might always use fourth-order Runge-Kutta, and in fact, that is the approach of many dynamics experts. However, for state configurations in which evaluating **D** is very expensive, one must consider the net cost of using a better integrator compared to using more and smaller time steps with a worse integrator. We generally care about the wall-clock time to achieve stable integration, and that might be better achieved through taking many small Euler steps. Again we see analogy to light transport: Many inexpensive samples may achieve faster convergence than few expensive samples. Thus, only with an end-to-end convergence, stability, and performance goal can we make a design decision for a particular system.

35.7 Remarks on Stability in Dynamics

While performance and scalability are important, dynamics is one of the few areas in computer graphics where numerical stability, and not performance, is the primary concern and focus of advancement. Numerical stability is related to precision and accuracy, but it is not the same as them. A simulation of a block tumbling down stairs is *accurate* if it produces results close to those of the corresponding real-world scenario. The simulation is *precise* if there are many bits in the output (regardless of whether they have the correct values!). The simulation is *stable* if the block reliably tumbles down, rather than gaining energy and launching into space or exploding, for example. The challenge is that stability is often opposed to accuracy. If the block merely remains sitting at the top of the stairs, even when unbalanced, the simulation is stable but so inaccurate that it is useless. Although stability is often driven by energy conservation, such events as missed collisions, infinite (conserving) spring oscillations, and infinite (conserving) micro-collisions may also be characterized as instability.

Why is stability so difficult to achieve in dynamics? As we've seen in this chapter, dynamics is deeply related to ordinary differential equations and numerical integration. A numerical integrator's precision and accuracy are governed by three elements: its representation precision (e.g., floating-point quaternions), its approximation method for derivatives, and its advancement (stepping and integration) method. Each of these is a complex source of error. When integrating forward in time, the feed-forward nature of the system creates positive feedback that tends to amplify errors. This is what often creates additional energy and is a major cause of instability. Solving via a noncausal process (i.e., looking both forward and backward in time) can counter the positive feedback loop with corresponding negative feedback, which may increase stability. Unfortunately, solving a system for all time generally precludes interactivity, in which future inputs are unknown.

In contrast to dynamics, physical simulation of light transport by various algorithms typically converges to a stable result. Energy in light differs from that in matter in three ways: Photons do not interact with one another (at macroscopic scales, at least), energy strictly decreases at interactions along a transport path through time and space (in everyday scenarios, at least), and the energy of light is independent of its position between interactions. The last point of comparison is the subtlest: Kinetic energy is explicit in a dynamics solver, but *potential* energy is largely hidden from the integrator and therefore a place in which error easily accumulates.

Thus, although algorithms like radiosity and photon mapping simulate light exclusively forward (or backward) in time, the feedback in the light transport integrator is not amplified by the underlying physics. In contrast, in dynamics, the laws of mechanics and the hidden potential energy repository conspire to amplify error and instability. Even worse, this error is often not proportional to precision, so one's first efforts at addressing it by increasing precision are often insufficient. For example, turning all 32-bit floating-point values into 64-bit ones or halving the integration time step often doubles simulation cost without "doubling" stability. As a result, many practical dynamics simulators are rife with scene-specific constants controlling bias, energy restitution, and constraint weights. Manual tuning of these constants is tricky and often unsatisfying, since a loss of accuracy is frequently the price of stability.

Thus, instability is an inherent problem in dynamics for interactive systems. Stability is a primary criterion for evaluating dynamics systems and algorithms, and much good work has been done on it in both industry and academia. Looking toward the future, we offer two speculations on stability.

The first speculation is that the *structure* of the integrator may be at least as important as the schemes it uses for derivatives and steps, and that this may be a fruitful area in which to seek improvements. This statement is motivated by Guendelman, Bridson, and Fedkiw's work on stability for stacks of rigid bodies [GBF03]. They showed that simply reordering steps in the inner loop of the integrator can dramatically increase stability for scenarios that have been traditionally challenging to simulate, and then demonstrated some additional sorting methods for enhancing stability further. This inspired others to experiment with the integration loop, and has led to various minor changes that produced significant increases in the stability of popular dynamics simulators. This area merits further study.

The second speculation is that the conventional wisdom, "fourth-order Runge-Kutta with fixed time steps is good enough" for dynamics in computer graphics,

has not been formally explored. Numerical integration is essential to other fields and is an area of constant scientific advance. We are not aware of experiments with various newer integration methods, particularly adaptive step-size ones, in dynamics simulation for computer graphics. Thus, it is an open question where the best tradeoff lies between per-step cost and steps per frame. Recall that the benefits of higher-order integration are only justified if they result in fewer steps and implicitly conserve precision. Taking more low-order steps, especially on massively parallel processors, could be preferable in some systems. Alternatively, taking very few high-order steps could be preferable. Thus, experimentation with the core numerical integration scheme is another area that may yield interesting results. Our understanding is that physical simulation for nongraphics engineering and science applications has been proceeding in this direction and we encourage its development within graphics as well.

35.8 Discussion

We've explained in this chapter how physically based animation can be described by differential equations whose solutions present various challenges, depending on context. The other kind of animation, which one might call "fine-art animation," and which involves human animators, is a separate discipline. Nonetheless, physically based animation and algorithmic animation are being used more and more, even within fine-art animation. This presents challenges and opportunities in designing comprehensible interfaces for artists who must interact with physics and try to control it to achieve certain artistic goals.

Chapter 36

Visibility Determination

36.1 Introduction

Determining the visible parts of surfaces is a fundamental graphics problem. It arises naturally in rendering because rendering objects that are unseen is both inefficient and incorrect. This problem is called either **visible surface determination** or **hidden surface removal,** depending on the direction from which it is approached.

The two distinct goals for visibility are algorithm correctness and efficiency. A visibility algorithm responsible for the correctness of rendering must exactly determine whether an unobstructed line of sight exists between two points, or equivalently, the set of all points to which one point has an unobstructed line of sight. The most intuitive application is **primary visibility:** Solve visibility exactly for the camera. Doing so will only allow the parts of the scene that are actually visible to color the image so that the correct result is produced. **Ray casting** and the **depth buffer** are by far the most popular methods for ensuring correct visibility today.

A **conservative visibility** algorithm is designed for efficiency. It will distinguish the parts of the scene that are likely visible from those that are definitely not visible, with respect to a point. Conservatively eliminating the nonvisible parts reduces the number of exact visibility tests required but does not guarantee correctness by itself. When a conservative result can be obtained much more quickly than an exact one, this speeds rendering if the conservative algorithm is used to prune the set that will be considered for exact visibility. For example, it is more efficient to identify that the sphere bounding a triangle mesh is behind the camera, and therefore invisible to the camera, than it is to test each triangle in the mesh individually.

Backface culling and **frustum culling** are two simple and effective methods for conservative visibility testing; **occlusion culling** is a more complex refinement of frustum culling that takes occlusion between objects in the scene into account. Sophisticated **spatial data structures** have been developed to decrease the cost of conservative visibility testing. Some of these, such as the Binary Space Partition

(BSP) tree and the hierarchical depth buffer, simultaneously address both efficient and exact visibility by incorporating conservative tests into their iteration mechanism. But often a good strategy is to combine a conservative visibility strategy for efficiency with a precise one for correctness.

> ✓ **THE CULLING PRINCIPLE:** It is often efficient to approach a problem with one or more fast and conservative solutions that narrow the space by culling obviously incorrect values, and a slow but exact solution that then needs only to consider the fewer remaining possibilities.

Primary visibility tells us which surfaces emit or scatter light toward a camera. They are the "last bounce" locations under light transport and are the only surfaces that directly affect the image. However, keep in mind that a global illumination renderer cannot completely eliminate the points that are invisible to the camera. This is because even though a surface may not *directly* scatter light toward the camera, it may still affect the image. Figure 36.1 shows an example in which removing a surface that is invisible to the camera changes the image, since that surface casts light onto surfaces that are visible to the camera. Another example is a shadow caster that is not visible, but casts a shadow on points that are visible to the camera. Removing the shadow caster from the entire rendering process would make the shadow disappear. So, primary visibility is an important subproblem that can be tackled with visibility determination algorithms, but it is not the only place where we will need to apply those algorithms.

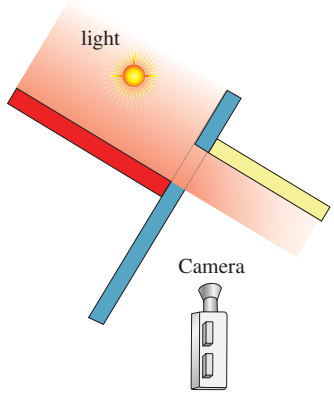


Figure 36.1: The yellow wall is illuminated only by light reflected from the hidden red polygon. Removing it will cause the yellow wall to be illuminated only by light from the blue surface.

The importance of indirect influence on the image due to points not visible to the camera is why we define exact visibility as a property that we can test for between *any* pair of points, not just between the camera and a scene point. A rendering algorithm incorporating global illumination must consider the visibility of each segment of a transport path from the source through the scene to the camera. Often the same algorithms and data structures can be applied to primary and indirect visibility. For example, the shadow map from Chapter 15 is equivalent to a depth buffer for a virtual camera placed at a light source.

There are of course nonrendering applications of algorithms originally introduced for visibility determination. Collision detection for the simulation of fast-moving particles like bullets and raindrops is often performed by tracing rays as if they were photons. Common modeling intersection operations such as cutting one shape out of another are closely related to classic visibility algorithms for subdividing surfaces along occlusion lines.

The motivating examples throughout this chapter emphasize primary visibility. That's because it is perhaps the most intuitive to consider, and because the camera's center of projection is often the single point that appears in the most visibility tests. For each example, consider how the same principles apply to general visibility tests. As you read about each data structure, think in particular about how many visibility tests at a point are required to amortize the overhead of building that data structure.

In this chapter, we first present a modern view of visibility following the light transport literature. We formally frame the visibility problem as an intersection query for a ray ("What does this ray hit first?") and as a visibility function on pairs of points ("Is Q visible from P?"). We then describe algorithms that can amortize that computation when it is performed conservatively over whole primitives for

efficient culling. This isn't the historical order of development. In fact, the topic developed in the opposite order.

Historically, the first notion of visibility was the question "Is any part of this triangle visible?" That grew more precise with "How much of this triangle is visible?" which was a critical question when all rendering involved drawing edges on a monochrome vector scope or rasterizing triangles on early displays and slow processors. With the rise of ray tracing and general light transport algorithms, a new visibility question was framed on points. That then gave a formal definition for the per-primitive questions, which expanded under notions of partial coverage to the framework encountered today. Of course, classic graphics work on primitives was performed with an understanding of the mathematics of intersection and precise visibility. The modern notion is just a redirection of the derivation: working up from points and rays with the rendering equation in mind, rather than down from surfaces under an ad hoc illumination and shading model.

36.1.1 The Visibility Function

Visible surface determination algorithms are grounded in a precise definition of visibility. We present this formally here in terms of geometry as the basis for the high-level algorithms. While it is essential for defining and understanding the algorithms, this direct form is rarely employed.

A performance reason that we can't directly apply the definition of visibility is that with large collections of surfaces in a scene, exhaustive visibility testing would be inefficient. So we'll quickly look for ways to amortize the cost across multiple surfaces or multiple point pairs.

A correctness concern with direct visibility is that under digital representations, the geometric tests involved in single tests are also very brittle. In general, it is impossible to represent most of the points on a line in any limited-precision format, so the answer to "Does this point occlude that line of sight?" must necessarily almost always be "no" on a digital computer. We can escape the numerical precision problem by working with spatial intervals—for example, line segments, polygons, and other curves—for which occlusion of a line of sight is actually representable, but we must always implicitly keep in mind the precision limitation at the boundaries of those intervals. Thus, the question of whether a ray passes through a triangle if it only intersects the edge is moot. In general, we can't even represent that intersection location in practice, so our classification is irrelevant. So, beware that everything in this chapter is only valid in practice when we are considering potential intersections that are "far away" from surface boundaries with respect to available precision, and the best that we can hope for near boundaries is a result that is spatially coherent rather than arbitrarily changing within the imprecise region.

Given points P and Q in the scene, let **visibility function** $V(P,Q) = 1$ if there is no intersection between the scene and the open-ended line segment between P and Q, and $V(P,Q) = 0$ otherwise. This is depicted in Figure 36.2. Sometimes it is convenient to work with the **occlusion function** $H(P,Q) = 1 - V(P,Q)$. The visibility function is necessarily symmetric, so $V(P,Q) = V(Q,P)$.

Note that the "visibility" in "visibility function" refers strictly to geometric line-of-sight visibility. If P and Q are separated by a pane of glass, $V(P,Q)$ is zero because a nonempty part of the scene (the glass pane) is intersected by the line segment between P and Q. Likewise, if an observer at Q has no direct line of sight

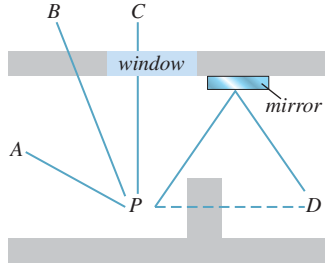


Figure 36.2: $V(P,A) = 1$ because there is no occluder. $V(P,B) = 0$ because a wall is in the way. $V(P,C) = 0$ because, even though P can see C through the window, the window is an occluder as far as mathematical "visibility" is concerned. Likewise, $V(P,D) = 0$, even though P sees a reflection of D in the mirror.

to P, but can see P through a mirror or see its shadow, we still say that there is no direct visibility: $V(P, Q) = 0$.

Let X be the first scene point encountered along a ray with origin P in direction $\hat{\omega} = \mathcal{S}(()Q - P)$. Point X partitions the ray into two visibility ranges. To see this, define f to be visibility as a function of distance from Q, that is, let $f(t) = V(P, P + \hat{\omega}t)$. Between X and P, $0 \leq t \leq |X - P|$, there is visibility, so $f(t) = 1$ as shown in Figure 36.3. Beyond X, $t > ||X - P||$. For that domain there is no visibility because X is an occluder, so $f(t) = 0$.

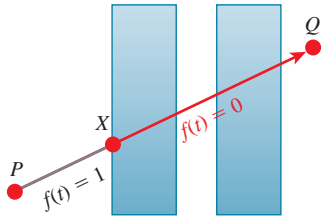


Figure 36.3: Visibility parameterized by distance along a ray.

Evaluating the visibility function is mathematically equivalent to finding that first occluding point X, given a starting point P and ray direction $\hat{\omega}$. The first occluding point along a ray is the solution to a ray **intersection query.** Chapter 37 presents data structures for efficiently solving ray intersection queries. It is not surprising that a common way to evaluate $V(P, Q)$ is to solve for X. If X exists and lies on the line segment PQ, then $V(P, Q) = 0$; otherwise, $V(P, Q) = 1$.

Some rendering algorithms explicitly evaluate the visibility function between pairs of points by testing for *any* intersection between a line segment and the surfaces in a scene. Examples include direct illumination shadow tests in a ray tracer and transport path testing in bidirectional path tracing [LW93, VG94], and Metropolis light transport [VG97].

Others solve for the first intersection along a ray instead, thus implicitly evaluating visibility by only generating the visible set of points. Examples include primary and recursive rays in a ray tracer and deferred-shading rasterizers. These are all algorithms with explicit visibility determination. They resolve visibility before computing transport (often called "shading" in the real-time rendering community) between points, thus avoiding the cost of scattering computations for points with no net transport. Simpler renderers compute transport first and rely on ordering to implicitly resolve visibility. For example, a naive ray tracer might shade *every* intersection encountered along a ray, but only retain the radiance computed at the one closest to the ray origin. This is equivalent to an (also naive) rasterization renderer that does not make a depth prepass. Obviously it is preferable to evaluate visibility before shading in cases where the cost of shading is relatively high, but whether to evaluate that visibility explicitly or implicitly greatly depends on the particular machine architecture and scene data structure. For example, rasterization renderers prefer a depth prepass today because the memory to store a depth buffer is now relatively inexpensive. Were the cost of a full-screen buffer very expensive compared to the cost of computation (as it once was, and might again become if resolution or manufacturing changes significantly), then a visibility prepass might be the naive choice and some kind of spatial data structure again dominate rasterization rendering.

Observe that following conventions from the literature, we defined the visibility function on the *open* line segment that does not include its endpoints. This means that if the ray from P to Q first meets scene geometry at a point X different from P, then $V(P, X) = 1$. This is a convenient definition given that the function is typically applied to points *on* surfaces. Were we to consider the closed line segment, then there would never be any visibility between the surfaces of a scene—they would all occlude themselves. We would have to consider points slightly offset from the surfaces in light transport equations.

In practice, the distinction between open and closed visibility only simplifies the notation of transport equations, not implementations in programs. That is because rounding operations implicitly occur after every operation when working

with finite-precision arithmetic, introducing small-magnitude errors (see Figure 36.5). So we must explicitly phrase all applications of the visibility function and all intersection queries with some small offset. This is often called **ray bumping** because it "bumps" the origin of the visibility test ray a small distance from the starting surface. Note that the bumping must happen on the other end as well. For example, to evaluate $V(Q, P)$, attempt to find a scene point $X = Q + \mathcal{S}(Q - P)t$ for $\epsilon < t < |P - Q| - \epsilon$. If and only if there is no scene point satisfying that constraint, then $V(Q, P) = 1$. Failing to choose a suitably large ϵ value can produce artifacts such as **shadow acne** (i.e., **self-shadowing**), speckled highlights and reflections, and darkening of the indirect components of illumination shown in Figure 36.4. The noisy nature of these artifacts arises from the sensitivity of the comparison operations to the small-magnitude representation error in the floating-point values.

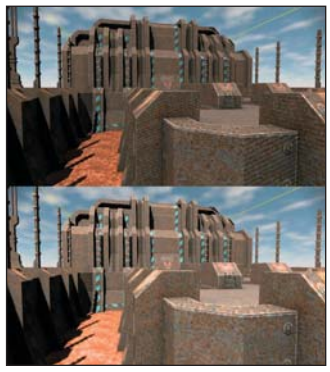


Figure 36.4: (Top) Self-occlusion from insufficient numerical precision or offset values causes the artifacts of shadow acne and speckling in indirect illumination terms such as mirror reflections. (Bottom) The same scene with the shadow acne removed.

36.1.2 Primary Visibility

Primary visibility (a.k.a. **eye ray visibility, camera visibility**) is visibility between a point on the aperture of a camera and a point in the scene. To render an image, one visibility test must be performed per light ray sample on the image plane. In the simplest case, there is one sample at the center of each pixel. Computing multiple samples at each pixel often improves image quality. See Section 36.9 for a discussion of visibility in the presence of multiple samples per pixel.

A pinhole camera has a zero-area aperture, so for each sample point on the image plane there is only one ray along which light can travel. That is the **primary ray** for that point on the image plane. Consider three points on the primary ray: sample point Q on the imager, the aperture A, and a point P in the scene. Since there are no occluding objects inside the camera, $V(Q, P) = V(A, P)$.

Since the visibility function evaluations or intersection queries at all samples share a common endpoint of the pinhole aperture, there are opportunities to amortize operations across the image. **Ray packet tracing** and **rasterization** are two algorithms that exploit this technique. For more details, see Chapter 15, which develops the amortized aspect of rasterization and presents the equivalence of intersection queries under rasterization and ray tracing.

36.1.3 (Binary) Coverage

Coverage is the special case of visibility for points on the image plane. For a scene composed of a single geometric primitive, the coverage of a primary ray is

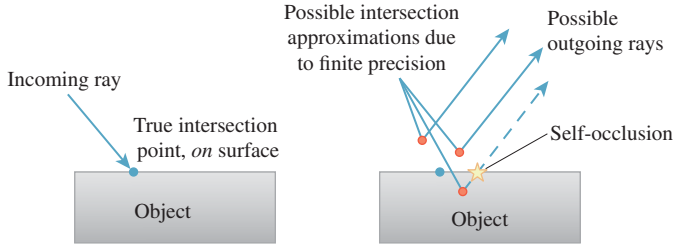


Figure 36.5: Finite precision leads to self-occlusions. "Bumping" the outgoing ray biases the representation error in a direction less likely to produce artifacts by favoring the points above the surface as the ray origin.

a binary value that is 1 if the ray intersects the primitive between the image plane and infinity, and 0 otherwise. This is equivalent to the visibility function applied to the primary ray origin and a point infinitely far away along the primary ray.

For a scene containing multiple primitives, primitives may occlude each other in the camera's view. The **depth complexity** of a ray is the number of times it intersects the scene. At any given intersection point P, the **quantitative invisibility** [App67] is the number of other primitive intersections that lie between the ray origin and P. Figure 36.6 shows examples of each of these concepts.

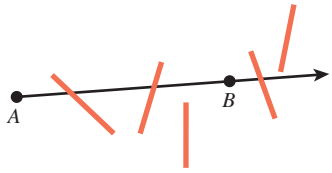


Figure 36.6: The quantitative invisibility of two points is the number of surface intersections on the segment between them. The quantitative invisibility of B with respect to A is 2 in this figure. The depth complexity of a ray is the total number of surface intersections along the ray. The ray from A through B has depth complexity 3 in this figure.

For some applications, backfaces—surfaces where the dot product of the geometric normal and the ray direction is positive—are ignored when computing depth complexity and quantitative invisibility. The case where the intersection of the ray and a surface is a segment (e.g., Figure 36.7) instead of a finite number of points is tricky. We've already discussed that the existence of such an intersection is already suspect due to limited precision, and in fact for it to occur at all with nonzero probability requires us to have explicitly placed geometry in just the right configuration to make such an unlikely intersection representable. Of course, humans are very good at constructing exactly those cases, for example, by placing edges at perfectly representable integer coordinates and aligning surfaces with axes in ways that could never occur in data measured from the real world. We note that for a closed polygonal model, it is common to ignore these line-segment intersections, but the definition of depth complexity and quantitative invisibility for this case varies throughout the literature.

Figure 36.7: A ray tangent to an intersected surface.

In the case of multiple primitive intersections, we say that the coverage is 1 at the first intersection and 0 at all later ones to match the visibility function definition.

At the end of this chapter, in Section 36.9, we extend binary coverage to **partial coverage** by considering multiple light paths per pixel.

36.1.4 Current Practice and Motivation

Most rendering today is on triangles. The triangles may be explicitly created, or they may be automatically generated from other shapes. Some common modeling primitives that are reducible to triangles are subdivision surfaces, implicit surfaces, point clouds, lines, font glyphs, quadrilaterals, and height field models.

Ray-tracing renderers solve exact visibility determination by **ray casting** (Section 36.2): intersecting the model with a ray to produce a sample. Data structures optimized for ray-triangle intersection queries are therefore important for efficient evaluation of the visibility function. Chapter 37 describes several of these data structures. **Backface culling** (Section 36.6) is implicitly part of ray casting.

Hardware rasterization renderers today tend to use **frustum culling** (Section 36.5), **frustum clipping** (Section 36.5), **backface culling** (Section 36.6), and a **depth buffer** (Section 36.3) for per-sample visible surface determination. Those methods provide correctness, but they require time linear in the number of primitives. So relying on them exclusively would not scale to large scenes. For efficiency it is therefore necessary to supplement those with conservative methods for determining occlusion and hierarchical methods for eliminating geometry outside the view frustum in sublinear time.

A handful of applications rely on the **painter's algorithm** (Section 36.4.1) of simply drawing everything in the scene in back-to-front order and letting the

ordering resolve visibility. This is neither exact nor conservative[1], but it has the benefit of extreme simplicity. It is used almost exclusively for 2D graphical user interface rendering to handle overlapping windows. The primary 3D application of the painter's algorithm today is for rasterization of translucent surfaces, although recent trends favor more accurate per-sample stochastic and lossy-volumetric alternatives [ESSL10, LV00, Car84, SML11, JB10, MB07].

Many applications combine multiple visibility determination algorithms. For example, a hybrid renderer might rasterize primary and shadow rays but perform ray casting for visibility determination on other global illumination paths. A real-time hardware rasterization renderer might augment its depth buffer with hierarchical occlusion culling or precomputed conservative visibility. Many games rely on those techniques, but they include a ray-casting algorithm for visibility determination used to determine line of sight for character AI logic and physical simulation.

36.2 Ray Casting

Ray casting is a direct process for answering an intersection query. As previously shown, it also computes the visibility function: $V(P, Q) = 1$, if P is the result of the intersection query on the ray with origin Q and direction $(P-Q)/|P-Q|$; and $V(P, Q) = 0$ otherwise. Chapter 15 introduced an algorithm for casting rays in scenes described by arrays of triangles, and showed that the same algorithm can be applied to primary and indirect (in that case, shadow) visibility.

The time cost of casting a ray against n triangles in an array is $O(n)$ operations. If $V(P, Q) = 1$, then the algorithm must actually test every ray-triangle pair. If $V(P, Q) = 0$, then the algorithm can terminate early when it encounters any intersection with the open segment PQ. In practice this means that computing the visibility function by ray casting may be faster than solving the intersection query; hence, resolving one shadow ray may be faster than finding the surface to shade for one ray from the camera. For a dense scene with high depth complexity, the performance ratio between them may be significant.

Terminating on any intersection still only makes ray casting against an array of surfaces faster by a constant, so it still requires $\mathrm{O}(n)$ operations for n surfaces. Linear performance is impractical for large and complex scenes, especially given that such scenes are exactly those in which almost all surfaces are not visible from a given point. Thus, there are few cases in which one would actually cast rays against an array of surfaces, and for almost all applications some other data structure is used to achieve sublinear scaling.

Chapter 37 describes many spatial data structures for accelerating ray intersection queries. These can substantially reduce the time cost of visibility determination compared to the linear cost under an array representation. However, building such a structure may only be worthwhile if there will be many visibility tests over which to amortize the build time. Constants vary with algorithms and architectures, but for more than one hundred triangles or other primitives and a few

1. … nor how artists actually paint—for example, sometimes the sky is painted *after* foreground objects—but the name is now both a technical term and appropriately evocative.

thousand visibility tests, building some kind of spatial data structure will usually provide a net performance gain.

We now give an example of how a ray-primitive intersection query operates within the **binary space partition tree** data structure. The issues encountered in this example apply to most other spatial data structures, including bounding volume hierarchies and grids.

36.2.1 BSP Ray-Primitive Intersection

The **binary space partition tree** (**BSP tree**) [SBGS69, FKN80] is a data structure for arranging geometric primitives based on their locations and extents. Chapter 37 describes in detail how to build and maintain such a structure, and some alternative data structures. The BSP tree supports finding the first intersection between a ray and a primitive. The algorithm for doing this often has only logarithmic running time in the number of primitives. We say "often" because there are many pathological tree structures and scene distributions that can make the intersection time linear, but these are easily avoidable for many scenes. This logarithmic scaling makes ray casting practical for large scenes.

The BSP tree can also be used to compute the visibility function. The algorithm for this is nearly identical to the first-intersection query. It simply terminates with a return value of false when *any* intersection is detected, and returns true otherwise.

There are a few variations on the BSP structure. For the following example, we consider a simple one to focus on the algorithm. In our simple tree, every internal node represents a **splitting plane** (which is not part of the scene geometry) and every leaf node represents a geometric primitive in the scene. A plane divides space into two half-spaces. Let the positive half-space contain all points in the plane and on the side to which the normal points. Let the negative half-space contain all points in the plane and on the side opposite to which the normal points. Figure 36.8 shows one such plane (for a 2D scene, so the "plane" is a line). Both the positive and negative half-spaces will be subdivided by additional planes when creating a full tree, until each sphere primitive is separated from the others by at least one plane.

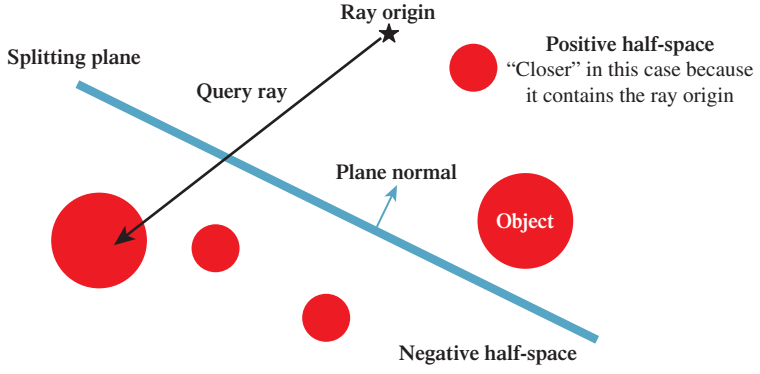


Figure 36.8: The splitting plane for a single internal BSP node divides this scene composed of five spheres into two half-spaces.

The internal nodes in the BSP tree have at most two children, which we label **positive** and **negative.** The construction algorithm for the tree ensures that the positive subtree contains only primitives that are in the positive half-space of the plane (or on the plane itself) and that the negative subtree contains only those in the negative half-space of the plane. If a primitive from the scene crosses a splitting plane, then the construction algorithm divides it into two primitives, split at that plane.

Listing 36.1 gives the algorithm to evaluate the visibility function in this simple BSP tree. The recursive intersects function performs the work. Point Q is visible to point P if no intersection exists between the line segment PQ and the geometry in the subtree with `node` at its root. When `node` is a leaf, it contains one geometric primitive, so `intersects` tests whether the line-primitive intersection is empty. Chapter 7 describes intersection algorithms that implement this test for different types of geometric primitives, and Chapter 15 contains C++ code for ray-triangle intersection.

Figure 36.9 visualizes the algorithm's iteration through a 2D tree for a scene consisting of disks.

If `node` is an internal node, then it contains a splitting plane that creates two half-spaces. We categorize the child nodes corresponding to these as being closer and farther with respect to P. Figure 36.8 shows an example classification at an internal node. With an eye toward reusing this algorithm's structure for the related problem of finding the *first* intersection, we choose to visit the closer node first. That is because if there is *any* intersection between segment PQ and the scene in `closer`, it must be closer to P than every intersection in `farther` [SBGS69].

If PQ lies entirely in one half-space, the result of the intersection test for the current node reduces to the result of the test on that half-space. Otherwise, there is an intersection if one is found in either half-space, so the algorithm recursively visits both.

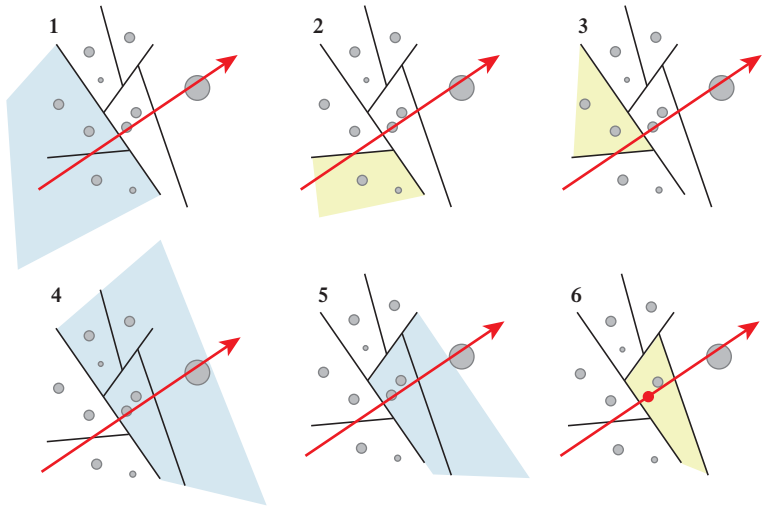


Figure 36.9: Tracing a ray through a scene containing disks stored in a 2D BSP tree. Highlighted portions of space correspond to the node at which the algorithm is operating in each step. Iteration proceeds depth-first, preferring to descend into the geometrically closer of the two children at each node.

Listing 36.1: Pseudocode for visibility testing in a BSP tree.

```
function V(P, Q):
   return not intersects(P, Q, root)

function intersects(P, Q, node):
   if node is a leaf:
      return (PQ intersects the primitive at the node)

   closer = node.positiveChild
   farther = node.negativeChild

   if P is in the negative half-space of node:
      // The negative side of the plane is closer to P
      swap closer, farther

   if intersects(P, Q, closer):
      // Terminate early because an intersection was found
      return true

   if P and Q are in the same half-space of node:
      // Segment PQ does not extend into the farther side
      return false

   // After searching the closer side, recursively search
   // for an intersection on the farther side
   return intersects(P, Q, farther)
```

Inline Exercise 36.1: The visibility testing code assumes that there's a closer and a farther half-space. What will this code do when the splitting plane contains the camera point P? Will it still perform correctly? If not, what modifications are needed?

In the worst case the routine must visit every node of the tree. In practice this rarely occurs. Typically, PQ is small with respect to the size of the scene and the planes carve space into convex regions that do not all lie along the same line. So we expect a relatively tight depth-first search with runtime proportional to the height of the tree.

There are many ways to improve the performance of this algorithm by a constant factor. These include clever algorithms for constructing the tree and extending the binary tree to higher branching factors. For sparse scenes, alternative spatial partitions can be advantageous. The convex spaces created by the splitting planes often have a lot of empty space compared to the volumes bounded by the geometric primitives within them in a BSP tree. A regular grid or bounding volume hierarchy may increase the primitive density within leaf nodes, thus reducing the number of primitive intersections performed.

Where BSP tree iteration is limited by memory bandwidth, substantial savings can be gained by using techniques for compressing the plane and node pointer representation [SSW+06].

36.2.2 Parallel Evaluation of Ray Tests

The previous analysis considered serial processing on a single scalar core. Parallel execution architectures change the analysis. Tree search is notoriously hard to execute concurrently for a *single query*. Near the root of a tree there isn't enough

work to distribute over multiple execution units. Deeper in the tree there is plenty of work, but because the lengths of paths may vary, the work of load balancing across multiple units may dominate the actual search itself. Furthermore, if the computational units share a single global memory, then the bandwidth constraints of that memory may still limit net performance. In that case, adding more computational units can *reduce* net performance because they will overwhelm the memory and reduce the coherence of memory access, eliminating any global cache efficiency.

There are opportunities for scaling nearly linearly in the number of computational units when performing *multiple* visibility queries simultaneously, if they have sufficient main memory bandwidth or independent on-processor caches. In this case, all threads can search the tree in parallel. Historically the architectures capable of massively parallel search have required some level of programmer instruction batching, called **vectorization.** Also called Single Instruction Multiple Data (SIMD), vector instructions mean that groups of logical threads must all branch the same way to achieve peak computational efficiency. When they do branch the same way, they are said to be branch-coherent; when they do not, they are said to be divergent. In practice, branch coherence is also a de facto requirement for any memory-limited search on a parallel architecture, since otherwise, executing more threads will require more bandwidth because they will fetch different values from memory.

There are many strategies for BSP search on SIMD architectures. Two that have been successful in both research and industry practice are **ray packet tracing** [WSBW01, WBS07, ORM08] and **megakernel tracing** [PBD+10]. Each essentially constrains a group of threads to descend the same way through the tree, even if some of the threads are forced to ignore the result because they should have branched the other way (see Figure 36.10). There are some architecture-specific subtleties about how far to iterate before recompacting threads based on their coherence, and how to schedule and cache memory results.

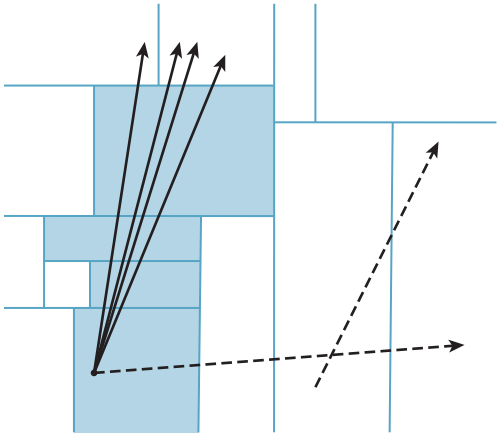


Figure 36.10: Packets of rays with similar direction and origin may perform similar traversals of a BSP tree. Processing them simultaneously on a parallel architecture can amortize the memory cost of fetching nodes and leverage vector registers and instructions. Rays that diverge from the common traversal (illustrated by dashed lines) reduce the efficiency of this approach.

Adding this kind of parallelism is more complicated than merely spawning multiple threads. It is also hard to ignore when considering high-performance visibility computations. At the time of this writing, vector instructions yield an 8x to 32*x* peak performance boost over scalar or branch-divergent instructions on the same processors. Fortunately, this kind of low-level visibility testing is increasingly provided by libraries, so you may never have to implement such an algorithm outside of an educational context. From a high level, one can look at hardware rasterization as an extreme optimization of parallel ray visibility testing for the particular application of primary rays under pinhole projection.

36.3 The Depth Buffer

A **depth buffer** [Cat74] is a 2D array parallel to the rendered image. It is also known as a **z-buffer, w-buffer,** and **depth map.** In the simplest form, there is one color sample per image pixel, and one scalar associated with each pixel representing some measure of the distance from the center of projection to the surface that colored the pixel.

To reduce the aliasing arising from taking a single sample per pixel, renderers frequently store many color and depth samples within a pixel. When resolving to an image for display, the color values are filtered (e.g., by averaging them), and the depth values are discarded. A variety of strategies for efficient rendering allow more independent depth samples than color samples, separating "shading" (color) from "coverage" (visibility). This section is limited to discussions of a single depth sample per pixel. We address strategies for multiple samples in Section 36.9.

Notice that the use of the depth buffer *assumes* that a single surface determines the value of each pixel. If the scene does not satisfy this assumption, then aliasing artifacts (see Chapter 18) are likely in the resultant rendering. This assumption often fails for distant complex objects, where many surfaces may all project within the same pixel. It makes sense, in these cases, to use a level-of-detail representation in which distant objects (or even collections of objects) are represented more and more simply to ensure that the assumption is valid. Chapter 25 discusses simplification techniques for meshes. Other approaches, like carefully choosing a far clipping plane, or masking distant objects with fog, can also help to address this problem.

Obviously, more than one surface contributes to pixels that contain the silhouette of an object, regardless of how large the object is in screen space. This is a pity, because this is a location at which the human visual system is extremely sensitive to artifacts—we are much less likely to notice aliasing within the interior of a shape.

Figure 36.11 reproduces Dürer's etching of himself and an assistant manually rendering a musical instrument under perspective projection. We have seen and referred to this classic etching before. In it, one man holds a pen at the location where a string crosses the image plane to dot a canvas that corresponds to our color buffer. The pulley on the wall is the center of projection and the string corresponds to a ray of light. Now note the plumb bob on the other side of the pulley. It maintains tension in the string. Dürer's primary interest was the 2D image

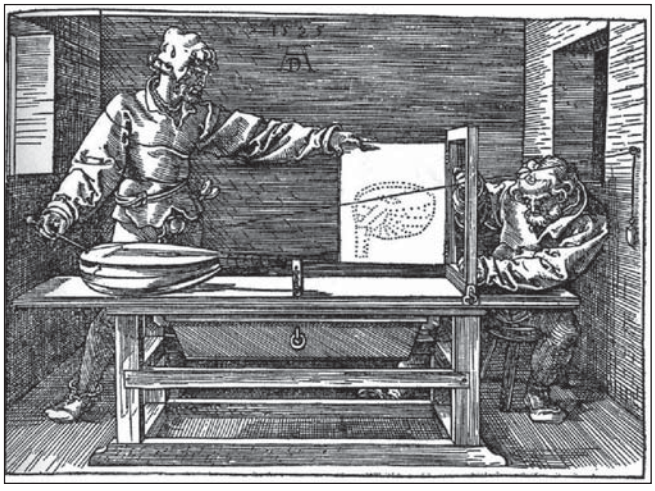


Figure 36.11: Two people using an early "rendering engine" to make a picture of a lute.

produced by marking intersections of that string with the image plane. But, as we noted in Chapter 3, this apparatus can in fact measure more than just the image. Consider what would happen if the artist were to annotate each point he marked on the image plane with the length of string between the plumb bob and the pulley corresponding to that point. He would then record a depth buffer for the scene, encoding samples of all *visible* three-dimensional geometry.

Dürer's artist had little need of a depth buffer for a single image. The physical object in front of him ensured correct visibility. However, given two images with depth buffers, he could have composited them into a single scene with correct visibility at each point. At each sample, only the nearer depth value (which in this case means a longer string below the pulley) could be visible in the combined scene. Our rendering algorithms work with virtual objects and lack the benefit of automatic physical occlusion. For simple convex or planar primitives such as points and triangles we know that each primitive does not occlude itself. This means we can render a single image plus depth buffer for each primitive without any visibility determination. The depth buffer allows us to combine rendering of multiple primitives and ensure correct visibility at each sample point.

The depth buffer is often visualized with white values in the distance and black values close to the camera, as if black shapes were emerging from white fog (see Figure 36.12). There are many methods for encoding the distance. The end of this section describes some that you may encounter. Depth buffers are commonly employed to ensure correct visibility under rasterization. However, they are also useful for computing shadowing and depth-based post-processing in other rendering frameworks, such as ray tracers.

There are three common applications of a depth buffer in visibility determination. First, while rendering a scene, the depth buffer provides implicit visible surface determination. A new surface may cover a sample only if its camera-space depth is less than the value in the depth buffer. If it is, then that new surface overwrites the color in the depth buffer and its depth value overwrites the depth in the depth buffer. This is implicit visibility because until rendering completes it is unknown what the closest visible surface is at a sample, or whether a given

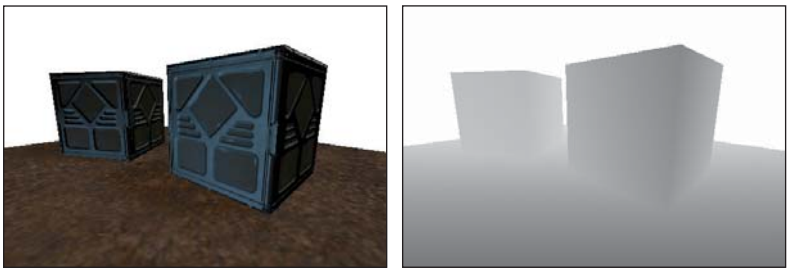


Figure 36.12: Rendering of a scene (left), and a visualization of its depth buffer (right).

surface is visible to the camera. Yet when rendering is complete, correct visibility is ensured.

Second, after the scene is rendered, the depth buffer describes the first scene intersection for any ray from the center of projection through a sample. Because the position of each sample on the image plane and the camera parameters are all known, the depth value of a sample is the only additional information needed to reconstruct the 3D position of the sample that colored it.

Third, after the scene is rendered, the depth buffer can directly evaluate the visibility function relative to the center of projection. For camera-space point Q, $V((0,0,0),Q) = 1$ if and only if the depth value at the projection of Q is less than the depth of Q.

The second and third applications deserve some more explanation of why one would want to solve visibility queries *after* rendering is already completed. Many rendering algorithms make multiple **passes** over the scene and the framebuffer. The ability to efficiently evaluate ray intersection queries and visibility after an initial pass means that subsequent rendering passes can be more efficient. One common technique exploiting this is the **depth prepass** [HW96]. In that pass, the renderer renders only the depth buffer, with no shading computations performed. Such a limited rendering pass may be substantially more efficient than a typical rendering pass, for two reasons. First, fixed-function circuitry can be employed because there is no shading. Second, minimal memory bandwidth is required when writing only to the depth buffer, which is often stored in compressed form [HAM06].

Note that a depth buffer must be paired with another algorithm such as rasterization for finding intersections of primary rays with the scene. Chapter 15 gives C++ code for ray casting and rasterization implementations of that intersection test. The rasterization implementation includes the code for a simple depth buffer. That implementation assumes that all polygons lie beyond the near clipping plane (see Chapter 13 for a discussion of clipping planes). This is to work around one of the drawbacks of the depth buffer: It is not a complete solution for visibility. Polygons need to be clipped against the near plane during rasterization to avoid the projection singularity at $z = 0$. The depth buffer can represent depth values behind the camera; however, rasterization algorithms are awkward and often inefficient to implement on triangles before projection. As a result, most rasterization algorithms pair a depth buffer with a geometric clipping algorithm. That geometric algorithm effectively performs a conservative visibility test by eliminating the parts of primitives that lie behind the camera before rasterization. The depth buffer then ensures correctness at the screen-space samples.

The depth buffer has proved to be a powerful solution for screen-space visibility determination. It is so powerful that not only has it been built in dedicated graphics circuitry since the 1990s, but it has also inspired many image-space techniques. Image space is a good place to solve many graphics problems because solving at the resolution of the output avoids excessive computation. In exchange for a constant memory factor overhead, many algorithms can run in time proportional to the number of pixels and sublinear to, if not independent of, the scene complexity. That is a very good algorithmic tradeoff. Furthermore, geometric algorithms are susceptible to numerical instability as infinitely thin rays and planes pass near one another on a computer with finite precision. This makes rasterization/image-space methods a more robust way of solving many graphics problems, albeit at the expense of aliasing and quantization in the result.

Inline Exercise 36.2: If there are T triangles in the scene and P pixels in the image, under what conditions on T and P would you expect image-space methods to be a good approach to visibility or related problems?

Inline Exercise 36.3: Image-space algorithms seem like a panacea. Describe a situation in which the discrete nature of image-space data makes it inappropriate for solving a problem.

36.3.1 Common Depth Buffer Encodings

Broadly speaking, there are two common choices for encoding depth: **hyperbolic** in camera-space z, and **linear** in camera-space z. Each has several variations for scaling conventions within the mapping. All have the property that they are monotonic, so the comparison $z_1 < z_2$ can be performed as $m(z_1) < m(z_2)$ (perhaps with negation) so that the inverse mapping is not necessary to implement correct visibility determination.

There are many factors to weigh in choosing a depth encoding. The operation count of encoding and decoding (for depth-based post-processing) may be significant. The underlying numeric representation, that is, floating point versus fixed point, affects how the mapping ultimately reduces to numeric precision. The dominant factor is often the relative amount of precision with respect to depth. This is because the accuracy of the visibility determination provided by a depth buffer is limited by its precision. If two surfaces are so close that their depths reduce to the same digital representation, then the depth buffer is unable to distinguish which is closer to the ray origin or camera. This means that the visibility determination will be arbitrarily resolved by primitive ordering or by small roundoff errors in the intersection algorithm. The resultant artifact is the appearance of individual samples with visibility results inconsistent with their neighbors. This is called ***z*-fighting.** Often z-fighting artifacts reveal the iteration order of the rasterizer or other intersection algorithm, which tends to cause regular patterns of small bias in depth. Different mappings and underlying numerical representation for depth vary the amount of precision throughout the scene. Depending on the kind of scene and rendering application, it may be desirable to have more precision close to the camera, uniform precision throughout, or possibly even high precision at some specific depth. Akeley and Su give an extensive and authoritative treatment [AS06] of this

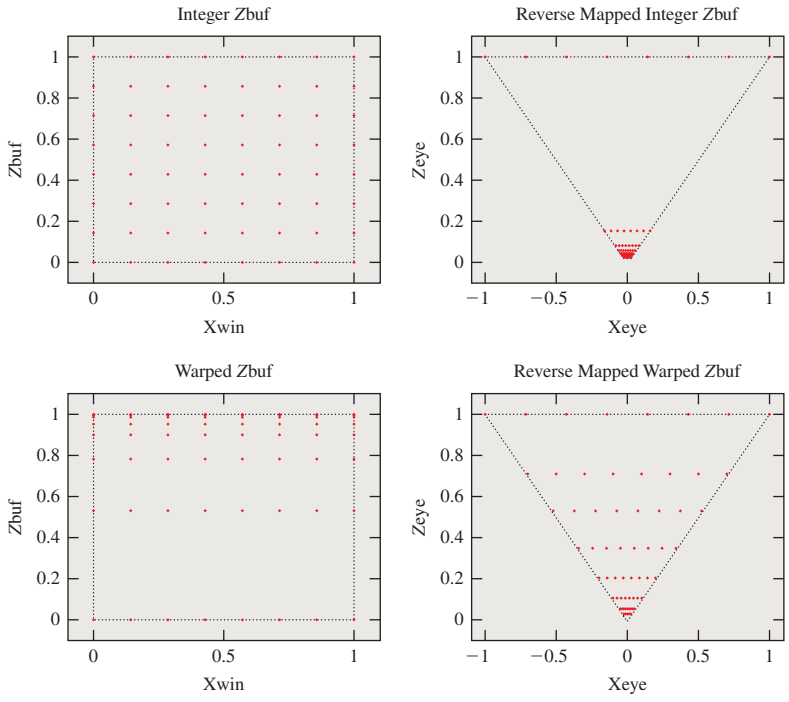


Figure 36.13: The points in (x, z bufferValue) space that are exactly representable under fixed-point, reverse-mapped fixed-point, and floating-point schemes. Fixed-point representations result in wildly varying depth precision with respect to screen-space x (or y).

topic. We summarize the basic ideas of the common mappings here and show Figures 36.13 and 36.14 by them to give a sense of the impact of representation on precision throughout the frustum.

Following (arbitrary) OpenGL conventions, for the following definitions let z be the position on the camera-space z-axis of the point coloring the sample. It is always a negative value. Let the far and near clipping planes be at $z_f = -f$ and $z_n = -n$.

36.3.1.1 Hyperbolic

The classic graphics choice describes a hyperbolically scaled normalized value arising from a projection matrix. This is typically called the ***z*-buffer** because it stores the z-component of points after multiplication by an API-specified perspective projection matrix and homogeneous division. This representation is also known as a **warped** z-buffer because it distorts world-space distances.

The OpenGL convention maps $-n$ to 0, $-f$ to 1, and values in between hyperbolically by

$$z \to \frac{f+n}{f-n} + \frac{2fn}{f-n}\frac{1}{z}. \tag{36.1}$$

Direct3D maps to the interval $[-1, 1]$ by

$$z \to \frac{f}{n-f} - \frac{fn}{n-f}\frac{1}{z}. \tag{36.2}$$

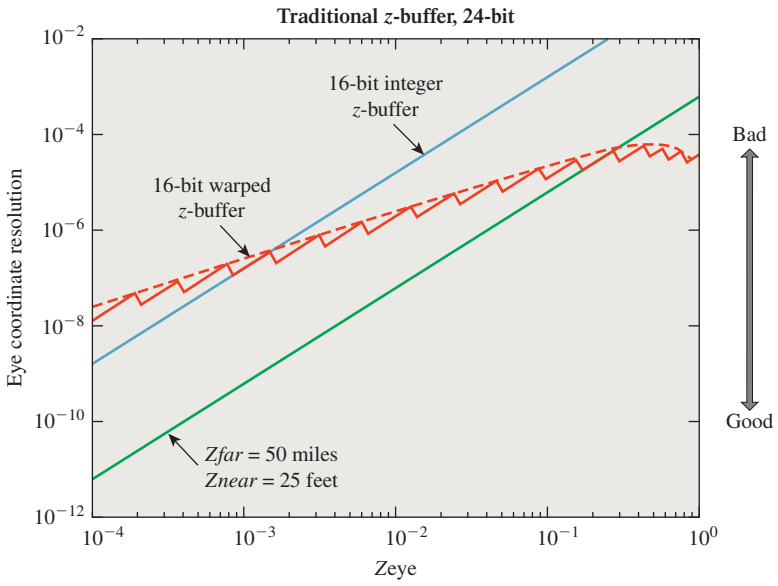


Figure 36.14: Comparison of precision versus depth for various z-buffer representations: 24-bit fixed point (green) is obviously strictly more accurate than 16-bit fixed point (blue); 16-bit floating point is more accurate than 16-bit fixed point when far from the camera (on the right), but has less precision very near to the camera (on the left). The blue and green curves are lines in log-log space, but would appear as hyperbolas in a linear plot. The red floating-point line is jagged because floating-point spacing is uniform within a single exponent and then jumps at the next exponent; the red curve is a smoothed trendline.

These mappings assign relatively more precision close to the near plane (where z-fighting artifacts may be more visible), have a normalized range that is appropriate for fixed-point implementation, and are expressible as a matrix multiplication followed by a homogeneous division. The amount of precision close to the near plane is based on the relative distance of the near and far planes from the center of projection. As the near plane moves closer to the center of projection, all precision rapidly shifts toward it, giving poor depth resolution deep in the scene.

A **complementary** or **reversed** hyperbolic [LJ99] encoding maps the far plane to the low end of the range and the near plane to the high end. For a fixed-point representation this is usually undesirable because nearby objects would receive higher depth representation errors, but under a floating-point representation this assigns nearly equal accuracy throughout the scene.

Another advantage of the nonlinear depth range is that it is possible to take the limit of the mapping as $f \to \infty$ [Bli93]. This allows a representation of depth within an infinite frustum using finite precision.

Inline Exercise 36.4: For $n = 1\text{m}, f = 101\text{m}$, compute the range of z-values within the view frustum that map to $[0, 0.9]$ under the OpenGL projection matrix. Repeat the exercise for $n = 0.1\text{m}$. How would this inform your choice of near and far plane locations? What is the drawback of pushing the near plane farther into the scene?

This was the preferred depth encoding until fairly recently. It was preferred because it is mathematically elegant and efficient in fixed-function circuitry to express the entire vertex transformation process as a matrix product. However, the widespread adoption of programmable vertex transformations and floating-point buffers in consumer hardware has made other formats viable. This reopened a classic debate on the ideal depth buffer representation. Of course, the ideal representation depends on the application, so while this mapping may no longer be preferred for some applications, it remains well suited for others. More than storage precision is at stake. For example, algorithms that expect to read world-space distances from the depth buffer pay some cost to reconstruct those values from warped ones, and the precision of the world-space value and cost of recovering it may be significant considerations.

Linear The terms **linear *z*, linear depth,** and ***w*-buffer** describe a family of possible values that are all linear in z. The "w" refers to the w-component of a point after multiplication by a perspective projection matrix but before homogeneous division.

These representations include the direct z-value for convenience; the positive "depth" value $-z$; the normalized value $(z + n)/(n - f)$ that is 0 at the near plane and 1 at the far plane; and $1 - (z + n)/(n - f)$, which happens to have nice precision properties in floating-point representation [LJ99]. In fixed point these give uniform world-space depth precision throughout the camera frustum, which makes z-fighting consistent in depth and can simplify the process of assigning decal offsets and other "epsilon" values. Linear depth is often conceptually (and computationally!) easier to work with in pixel shaders that require depth as an input. Examples include soft particles [Lor07] and screen-space ambient occlusion [SA07].

36.4 List-Priority Algorithms

The list-priority algorithms implicitly resolve visibility by rendering scene elements in an order where occluded objects have higher priority, and are thus hidden by overdraw later in the rendering process. These algorithms were an important part of the development of real-time mesh rendering.

Today list-priority algorithms are employed infrequently because better alternatives are available. Spatial data structures can explicitly resolve visibility for ray casts. For rasterization, the memory for a depth buffer is now fast and inexpensive. In that sense, brute force image-space visibility determination has come to dominate rasterization. But the depth buffer also supports an intelligent algorithmic choice. Early depth tests and early depth rendering passes avoid the inefficiency of overdrawing samples, and today's renderers spend significantly more time shading samples than resolving visibility for them because shading models have grown very sophisticated. So a list-priority visibility algorithm that increases shading time is making the expensive part of rendering more expensive. Despite their current limited application, we discuss three list-priority algorithms.

However, the implicit and refreshingly simple approach of implicit visibility by priority is a counterpoint to the relative complexity of something like hierarchical occlusion culling. There are also some isolated applications, especially graphics for nonraster output, where list priority may be the right approach. We

find that the painter's algorithm is embedded in most user interface systems, and is often encountered even in sophisticated 3D renderers for handling issues like transparency when memory or render time is severely limited.

Some list-priority algorithms are heuristics that often generate a correct ordering, but fail in some cases. Others produce an exact ordering, which may require splitting the input primitives to resolve cases such as Figure 36.15. There's an important caveat for the algorithms that produce exact results: If we are going to do the work of subdivision, we can achieve more efficient rendering by simply culling *all* occluded portions, rather than just overdrawing them later in rendering. This was a popular approach in the 1970s. Area-subdivision algorithms such as Warnock's Algorithm [War69] and the Weiler-Atherton Algorithm [WA77] eliminate culled areas in 2D. There are similar per-scanline 1D algorithms such as those by Wylie et al. [WREE67], Bouknight [Bou70], Watkins [Wat70], and Sechrest and Greenberg [SG81] that use an **active edge table** to maintain the current-closest polygon along a horizontal line. These were historically extended to single-line depth buffers [Mye75, Cro84]. The obvious trend ensued, and today all of these are largely ignored in favor of full-screen depth buffers. This is a cautionary tale for algorithm development in the long run, since the simplicity found in the depth buffer and painter's algorithm leads to better performance and more practical implementation than decades of sophisticated visibility algorithms.

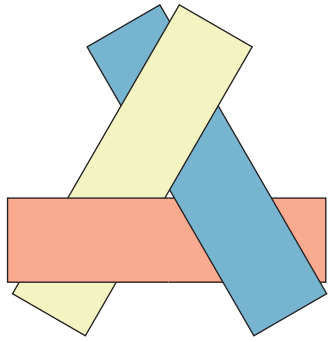


Figure 36.15: Three convex polygons that cannot be rendered properly using the painter's algorithm due to their mutual overlaps. At each point, a strict depth ordering exists, but there is no correct ordering of whole rectangles.

36.4.1 The Painter's Algorithm

Consider a possible process for an artist painting a landscape. The artist paints the sky first, and then the mountains occluding the sky. In the foreground, the artist paints trees over the mountains. This is called **the painter's algorithm** in computer graphics. Occlusion and visibility are achieved by overwriting colors due to distant points with colors due to nearer points. We can apply this idea to each sample location because at each sample there is always a correct back-to-front ordering of the points directly affecting it. In this case, the algorithm is potentially inefficient because it requires sorting all of the points, but it gives a correct result.

For efficiency, the painter's algorithm is frequently applied to whole primitives, such as triangles. Here it fails as an algorithm. While primitives larger than points can *often* be ordered so as to give correct visibility, there are situations where this cannot be done. Figure 36.15 shows three triangles for which there is no correct back-to-front order. Here the "algorithm" is merely a heuristic, although it can be an effective one. If we allow subdividing primitives where their projections cross, then we can achieve an ordering. This is discussed in the following section.

Despite its inability to generate either correct or conservative results in the general case, the painter's algorithm is employed for some niche applications in computer graphics and is a useful concept in many cases. It is trivially simple, requires no space, and can operate out of core (provided the sort is implemented out of core). It is the predominant visibility algorithm in 2D user interfaces and presentation graphics. In these systems, all objects are modeled as lying in planes parallel to the image plane. For that special case, the primitives can always be ordered, and the ordering is trivial to achieve because those planes are typically parallel to the image plane.

Primarily-2D rendering systems such as windowed GUIs that also incorporate occlusion or "stacking" are called **2.5D** because they have a depth ordering (a.k.a. ***z*-order**) and overlap even though explicit position in a third dimension is ambiguous. We've argued that this is a reasonable application of the painter's algorithm, especially when the primitives obey a strict depth ordering so that sorting is guaranteed to correctly resolve visibility.

An alternative for occlusion in 2.5D systems is to consider only local solutions and relax the need for a consistent global interpretation of the scene (even though one may exist). Two examples of this are Live Paint and Local Layering. **Live Paint** [ASP07] is a system for editing 2D drawings composed of curves and flood-fill regions, that is, "presentation vector graphics" such as those found in Microsoft PowerPoint or created in Adobe Illustrator. Most systems for such graphics treat the elements as 2.5D filled primitives, with stacking creating the occlusion. Operations for computing intersections, unions, and subtractions between them allow editing based on occlusion but destroy the underlying curves in the process. Live Paint instead works with the curves themselves as a planar graph and attempts to maintain the consistency of the fill commands through a series of clever methods. This allows for more natural editing of the image and preserves curves during editing operations.

Local Layering [MP09] is an alternative system for managing discrete, nonconvex primitives in a 2.5D system. Instead of assigning a strict global depth ordering, it allows the artist to specify which primitive is closer to the viewer separately at each intersection. This allows complex overlapping behavior such as braiding different primitives together.

When we render with a depth buffer and an early depth test, it is advantageous to encounter proximate surfaces before distant ones. Distant surfaces will then fail the early depth test where they are occluded, and not require shading. A reverse painter's algorithm improves efficiency in that case: Render from front to back. A depth prepass eliminates the need for ordering. During the prepass itself the ordering provides a speedup. Surprisingly, for many models a static ordering of primitives can be precomputed that provides a good front-to-back ordering from any viewpoint [SNB07]. This allows the runtime performance without the runtime cost.

The painter's algorithm is often employed for translucency, which can be modeled as fractional visibility values between zero and one, as done in OpenGL and Direct3D. Compositing translucent surfaces from back to front allows a good approximation of their fractional occlusion of each other and the background in many cases. However, stochastic methods yield more robust results for this case at the expense of noise and a larger memory footprint. See Section 36.9 for a more complete discussion.

36.4.2 The Depth-Sort Algorithm

Newell et al.'s **depth-sort algorithm** [NNS72] extends the painter's algorithm for polygons to produce correct output in all cases. It operates in four steps.

1. Assign each polygon a sort key equal to the camera-space z-value of the vertex *farthest* from the viewport.
2. Sort all polygons from farthest to nearest according to their keys.
3. Detect cases where two polygons have ambiguous ordering. Subdivide such polygons until the pieces have an explicit ordering, and place those in the sort list in the correct priority order.
4. Render all polygons in priority order, from farthest to nearest.

The ordering of two polygons is considered ambiguous under this algorithm if their z-extents and 2D projections (i.e., their homogeneous clip-space, axis-aligned bounding boxes) overlap and one polygon intersects the plane of the other.

36.4.3 Clusters and BSP Sort

Consider a scene defined by a set of polygons and a viewer using a pinhole projection model. Schumacker [SBGS69] noted that a plane passing through the scene that does not intersect any polygons divides them into two sets. Those polygons on the same side of the plane as the viewer must be strictly closer to the viewer, and thus cannot be occluded by those polygons on the farther side. He grouped polygons into **clusters,** and recursively subdivided them when suitable partition planes could not be found. Within each cluster he precomputed a viewer-independent ordering (see later work by Sander et al. [SNB07] on a related problem), and employed a special-purpose rasterizer that followed these orderings.

Fuchs, Kedem, and Naylor [FKN80] generalized these ideas into the binary space partition tree. We have already discussed in this chapter how BSP trees can solve visible surface determination by accelerating ray-primitive intersection. They can also be applied in the context of a list-priority algorithm, and that was their original motivating application. (We will shortly see two more applications of BSP trees to visibility: portals and mirrors, and precomputed visibility.)

The same logic found in the ray-intersection algorithm applies to the list-priority rendering with a BSP tree; we are conceptually performing ray intersection on all possible view rays. Listing 36.2 gives an implementation that sorts all polygons from farthest to nearest, given a previously computed BSP tree. We can look at this as a variation on the depth-sort algorithm where we are guaranteed to never encounter the case requiring subdivision. We never need to subdivide during traversal because the tree's construction already performed subdivision at partition planes between nearby polygons.

Listing 36.2: The list-priority algorithm for rendering polygons in a BSP tree with root `node` *as observed by a viewer at* `P`.

```
function BSPPriorityRender(P, node):
    if node is a leaf:
        render the polygon at node
        return

    closer = node.positiveChild
    farther = node.negativeChild

    if P is in the negative half-space of node:
        swap closer, farther

    BSPPriorityRender(P, farther)
    BSPPriorityRender(P, closer)
```

36.5 Frustum Culling and Clipping

Assume that we rely on an exact method like a depth buffer or Newell et al.'s depth-sort algorithm for correct visibility under rasterization. To avoid writing to illegal or incorrect memory addresses, assume that we perform 2D **scissoring** to the viewport. This just means that the rasterizer may generate (x, y) locations that are outside the viewport, but we only allow it to write to memory at locations inside the viewport. We can also scissor in depth: No sample may be written whose depth indicates that it is behind the camera or past the far plane.

Recall that Chapter 13 showed that a rectangular viewport with near and far planes parallel to the image plane defines a volume of 3D space called the view frustum. This is a pyramid with a rectangular base and the top cut off. Clipping to the sides of the view frustum in 3D corresponds to clipping the projection of primitives to the viewport, and produces equivalent results to simply scissoring in 2D.

Scissoring alone ensures correctness, but it may lead to poor efficiency. For example, most primitives that are rasterized may fail the scissor test. There are three common approaches to increasing efficiency in this case that are related to the view frustum.

- **Frustum culling:** Eliminate polygons that are entirely outside the frustum, for efficiency.
- **Near-plane clipping:** Clip polygons against the near plane to enable simpler rasterization algorithms and avoid spending work on samples that fail depth scissoring.
- **Whole-frustum clipping:** Clip polygons to the side and far planes, for efficiency.

In general, it is a good strategy to use scissoring and clipping to complement each other. Use each only for the case where it has high efficiency and low implementation complexity. For example, use a coarse culling based on the view frustum followed by clipping to the near plane and scissoring in 2D. For primitives whose projections are small compared to the viewport, this leads to the scissor test usually passing, which means that most parts of most rasterized primitives for which significant computation is performed are usually on the screen.

36.5.1 Frustum Culling

Eliminating polygons outside the view frustum is simple. One 3D algorithm for this tests each vertex of a polygon against each plane bounding the view frustum. Assume that the planes are oriented so that the view frustum is the intersection of the six positive half-spaces. If there exists some plane for which all vertices of a polygon are in the negative half-space, then that polygon must lie entirely outside the view frustum and can be culled. For small polygons, it may not be efficient to perform this test on each polygon. For example, if a polygon affects at most one sample, then a 3D bounding box test on a single point yields the same result. So frustum culling may be performed on a bounding box hierarchy.

A drawback of the 3D frustum culling algorithm just described is that it may be too conservative. Polygons that are outside the view frustum but near a corner or edge may intersect multiple planes.

36.5.2 Clipping

36.5.2.1 Sutherland-Hodgman 2D Clipping

There are many clipping algorithms. Perhaps the simplest is the 2D variation of Sutherland's and Hodgman's [SH74] algorithm. It clips one (arbitrary) **source** polygon against a second, convex **boundary** polygon (see Figure 36.16). The algorithm proceeds by incrementally clipping the source against the line through each edge of the boundary polygon, as shown in Listing 36.3.

Inline Exercise 36.5: Construct an example input polygon which, when clipped against the unit square by the Sutherland-Hodgman algorithm, produces a polygon with degenerate edges (i.e., edges that meet at a vertex v with an exterior angle of 180°).

For viewport clipping, Sutherland-Hodgman is applied to a projected polygon and the rectangle of the viewport. For projected polygons that have significant area outside the viewport, clipping to the viewport is an efficient alternative to scissor-testing each sampled point. The algorithm is further useful as a general geometric operation in many contexts, including modeling shapes in the first place.

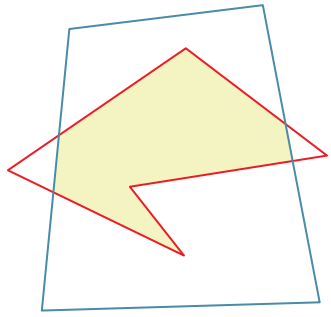


Figure 36.16: The red input polygon is clipped against the convex blue boundary polygon; the result is the boundary of the yellow shaded area.

Listing 36.3: Pseudocode for Sutherland-Hodgman clipping in 2D.

```
// The arrays are the vertices of the polygons.
// boundaryPoly must be convex.
function polyClip(Point sourcePoly[], Point boundaryPoly[]):
   for each edge (A, B) in boundaryPoly:
       sourcePoly = clip(sourcePoly, A, Vector(A.y-B.y, B.x-A.x))
   return sourcePoly

// True if vertex V is on the "inside" of the line through P
// with normal n. The definition of inside depends on the
// direction of the y-axes and whether the winding rule is
// clockwise or counter-clockwise.
function inside(Point V, Point P, Vector n):
   return (V - P).dot(n) > 0

// Intersection of edge CD with the line through P with normal n
function intersection(Point C, Point D, Point P, Vector n):
   distance = (C - P).dot(n) / n.length()
   t = (D - C).length()
   return D * t + C * (1 - t)

// Clip polygon sourcePoly against the line through P with normal n
function clip(Point sourcePoly[], Point P, Vector n):
   Point result[];

   // Add the last point, if it is inside
   D = sourcePoly[sourcePoly.length - 1]
   Din = inside(D, P, n)
   if (Din): result.append(D)

   for (i = 0; i < sourcePoly.length; ++i) :
       C = D, Cin = Din

       D = sourcePoly[i]
       Din = inside(D, P, n)

```

```
36          if (Din != Cin): // Crossed the line
37              result.append(intersection(C, D, P, n))
38
39          if (Din): result.append(D)
40
41      return result
```

The algorithm produces some degenerate edges, which don't matter for polygon rasterization but can cause problems when we apply the same ideas in other contexts.

36.5.2.2 Near-Plane Clipping

The 2D Sutherland-Hodgman algorithm generalizes to higher dimensions. To clip a polygon to a plane, we walk the edges finding intersections with the plane. We can do this for the whole view frustum, processing one plane at a time. Consider just the step of clipping to the near plane for now, however.

In camera space, the intersections between polygon edges and the near plane are easy to find because the near plane has a simple equation: $z = -n$. In fact, this is exactly the same problem as clipping a polygon by a line, since we can project the problem orthogonally into either the *xz*- or *yz*-plane. We interpolate vertex attributes that vary linearly across the polygon linearly to the new vertices introduced by clipping, as if they were additional spatial dimensions. Listing 36.4 gives the details in pseudocode for clipping a polygon specified by its vertex list against the plane `z = zn`, where `zn < 0`.

Listing 36.4: Clipping of the polygon represented by the `vertex` *array against the near plane* `z = zn`.

```
1  function clipPolygon(inVertices, zn):
2    outVertices = []
3    Let start = inputVertices.last();
4    for end in inputVertices:
5       if end.z <= zn:
6          if start.z > zn:
7              // We crossed into the frustum
8              outVertices.append( clipLine(start, end, zn) )
9
10           // the endpoint of this edge is in the frustum
11           outVertices.append( end )
12
13      elif start.z <= zn:
14         // We crossed out of the frustum
15         outVertices.append( clipLine(start, end, zn) )
16
17      start = end
18
19   return outVertices
20
21
22 function clipLine(start, end, zn):
23    a = (zn - start.z) / (end.z - start.z)
24    // This holds for any vertex properties that we
25    // wish to linearly interpolate, not just position
26    return start * a + end * (1 - a)
```

36.5.3 Clipping to the Whole Frustum

Having clipped against the near plane, we are guaranteed that for every vertex in the polygon $z < 0$. This means that we can project the polygon into **homogeneous clip space,** mapping the frustum to a cube through perspective projection as described in Chapter 13.

We could continue with Sutherland-Hodgman clipping for the other frustum planes in 3D before projection. However, the side planes are not orthogonal to an axis the way that the near plane is, so clipping to those planes takes more operations per vertex. In comparison, *after* perspective projection every frustum plane is orthogonal to some axis, so clipping is just as efficient for a side plane as it was for the near plane. That is, clipping is again a 2D operation. Clipping to the far plane can be performed either before or after projection.

When clipping in Cartesian 3D space against the near plane, we were able to linearly interpolate per-vertex attributes such as texture coordinates. In homogeneous clip space, those attributes do not vary linearly along an edge, so we cannot directly linearly interpolate. However, the relationship is nearly as simple.

In practice, we don't need all of those operations for each edge clipping operation. Instead, we can project every attribute by $u' = -u/z$ when projecting position. Then we can perform all clipping on the u' attributes as if they were linear and all operations were 2D. Recall that rasterization needs to interpolate attributes in a perspective-correct fashion, so it operates on the u' attributes along a scan line anyway (see The Depth Buffer). Only at the per-sample "shading" step do we return to the original attribute space, by computing $u = -u'z$ with the hyperbolically interpolated z-value. Thus, in practice, the clipping (and rasterization) cost for 3D attributes is the same as for 2D attributes, and all of the 2D optimization techniques such as finite differences can be applied to the u'-values.

36.6 Backface Culling

The back of an opaque, solid object is necessarily hidden from direct line of sight from an observer. The object itself occludes the view rays. Culling primitives that lie on the backs of objects can therefore conservatively eliminate about half of the scene geometry. As pointed out previously, backface culling is a good optimization when computing the visibility function, but not for the entire rendering pipeline. When we consider the entire rendering pipeline, the image may be affected by points not directly visible to the camera, such as objects seen by their reflections in mirrors and shadows cast by objects outside the field of view. So, while backface culling is one of the first tools that we reach for when optimizing visibility, it is important to apply it at the correct level. One can occasionally glimpse errors arising from programs culling at the wrong stage, such as shadows disappearing when their caster is not in view.

Although backface culling could be applied to parametric curved surfaces, it is typically performed on polygons. That is because a test at a single point on the polygon indicates whether the entire polygon lies on the front or back of an object, so it is very efficient to test polygons. A curve may require tests at multiple points or an analytic test over the entire surface to make the same determination.

We intuitively recognize the back of an object—it is what we can't see!—but how can we distinguish it geometrically? Consider a closed polygonal mesh with no self-intersections, and an observer at point Q that lies outside the polyhedron

defined by the mesh. Let P be the vertex of a polygon and $\hat{n}$ be the normal to the polygon (specifically, the true geometric normal to the polygon, not some implied surface normal at the vertex to be used in smooth shading). The polygon defines a plane that passes through P and has normal $\hat{n}$. We say that the polygon is a **frontface** with respect to Q if Q lies in the positive half-plane of the polygon and that the polygon is a **backface** if Q lies in the negative half-plane. If Q lies exactly in the plane, then the polygon lies on the **contour** curve dividing the front and back of the object.

$$(Q - P) \cdot \hat{n} > 0 : \text{Polygon is a frontface} \tag{36.3}$$

$$(Q - P) \cdot \hat{n} < 0 : \text{Polygon is a backface} \tag{36.4}$$

$$(Q - P) \cdot \hat{n} = 0 : \text{Polygon is on the contour} \tag{36.5}$$

The result will be the same for the polygon regardless of which vertex we choose for a polygon, and regardless of the field of view of the camera. However, Figure 36.17 shows that an object close to the camera tends to have more backfaces than the same object far from the camera, and that this is a general phenomenon, at least for convex objects.

Inline Exercise 36.6: Prove that for a triangle and a viewer, the backface classification is independent of which vertex we consider.

Most ray tracers and rasterizers perform backface culling on whole triangles before progressing to per-sample intersection tests. This is a more effective strategy for rasterizers because they can amortize the backface test (and the corresponding cost of reading the triangle into memory) over the whole triangle. A ray tracer generally must perform the test once per sample.

Backface culling assumes opaque, solid objects. If the ray starting point Q (e.g., the viewpoint for a primary ray) is inside the volume bounded by a mesh, then it is not conservative to cull backfaces. It also isn't obvious what the result

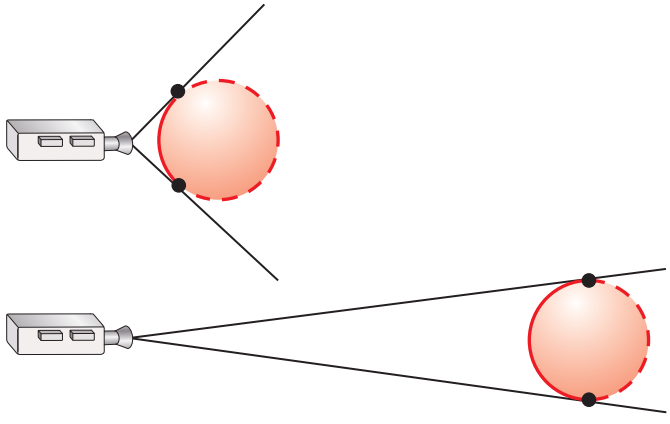


Figure 36.17: Two cameras facing to the right, toward spheres. The long lines depict the rays from the center of projection to the silhouette of the sphere, which is where the backfacing and frontfacing surfaces meet. The top camera is near a sphere, so most of the sphere's surface is backfacing. The bottom camera is distant from its sphere, so only about half of the sphere's surface is backfacing.

of visibility determination should be in such a situation because it does not correspond to a physically plausible scene. If an object is composed of a material that transmits light, then a geometric "visibility" test does not correspond to a light transport visibility test.

If we abuse geometric visibility in this case and apply backface culling, the back surface will disappear. In practice one can model a transmissive object with coincident and oppositely oriented surfaces. For example, a glass ball consists of a spherical air-glass interface oriented outward from the center of the ball and a glass-air interface oriented inward. Backface culling remains conservative under this model. Along a path that does not experience total internal refraction, light originating outside the ball first interacts with an air-glass frontface to enter the ball and then with a glass-air frontface to exit again, as shown in Figure 36.18.

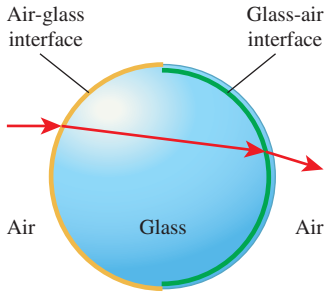


Figure 36.18: Backface culling allows a ray to intersect the correct one of the two coincident air-glass and glass-air interfaces of a glass ball surrounded by air.

36.7 Hierarchical Occlusion Culling

If no part of a box is visible from a point Q that is outside of the box, and if the box is replaced with some new object that fits inside it, then no part of that object can possibly be visible either. This observation holds for any shape, not just a box. This is the key idea of **occlusion culling.** It seeks to conservatively identify that a complex object is not visible by proving that a geometrically simpler bounding volume around the object is also not visible. It is an excellent strategy for efficient conservative visibility determination on dynamic scenes.

How much simpler should the bounding volume be? If it is too simple, then it may be much larger than the original object and will generate too many false-positive results (i.e., the bounding volume will often be visible even when the actual object is not). If it is too complex, then we gain little net efficiency even if it is a good predictor. The natural solution is to divide and conquer. Create a **Bounding Volume Hierarchy** (**BVH**; see Chapter 37) and walk its tree. If a node is not visible, then all of its children must not be visible. This is one form of **hierarchical occlusion culling.**

A ray tracer with a hierarchical spatial data structure effectively performs occlusion culling, although the term is not typically applied to that case. For example, a ray cast through a BVH corresponds exactly to the algorithm from the previous paragraph.

For rasterization, occlusion culling is only useful if we can test visibility for the bounding volumes substantially faster than we can for the primitives themselves. There are two implementation strategies, commonly built directly into rasterization hardware, that support this.

The first is a special rasterization operation called an **occlusion query** [Sek04] that invokes no shading or changes to the depth buffer. Its only output is a count of the number of samples that would have passed the depth buffer visibility test. It can be substantially faster than full rasterization because it requires no output bandwidth to the framebuffer and no synchronous access to the depth buffer for updates, interpolates no attributes except depth, and launches no shading operations. Chapter 38 shows that those are often the expensive operations in rasterization, so eliminating them decreases the cost of rasterization substantially.

Occlusion culling with an occlusion query issues several queries asynchronously from the main rendering thread. When the result of a query is available, it then renders the corresponding object if and only if one or more pixels

passed the occlusion query. This is made hierarchical by recursively considering bounds on smaller parts of the scene when a visible bound is observed.

The second strategy for efficient hardware occlusion culling is the **hierarchical depth buffer,** a.k.a. **hierarchical *z*-buffer** [GKM93]. This is an image pyramid similar to a MIP map in depth. Level zero of the tree is the full-resolution depth buffer. Every subsequent level has half the resolution of its parent level. A sample in level $k + 1$ of the hierarchical depth buffer stores two values: the minimum and maximum depths of four corresponding samples from level k. A hierarchical rasterizer [GKM93, Gre96] working with such a structure solves for minimum (or maximum, depending on the desired depth test) depth of a primitive within the area surrounding each of the depth samples at the lowest resolution. If the primitive's minimum depth is smaller (i.e., closer to the camera) than the maximum depth at a sample, then some part of that primitive may be visible at the highest resolution, so the rasterizer progresses to the next level for that sample. If the primitive's minimum depth is greater than or equal to the maximum depth at a depth buffer sample, then even the farthest point in the subtree represented by that sample is still closer to the camera than the closest point on the primitive and the primitive must be occluded at all resolutions represented.

A hierarchical depth buffer is naturally most efficient for large primitives. This is because because when parts of a large primitive are conservatively classified as invisible near the top of the tree, many high-resolution visibility tests are eliminated. Small primitives force the rasterizer to begin tests deeper in the tree where the potential cost savings are smaller. Because hierarchical occlusion queries on BVHs tend to bound meshes of small primitives with a few large ones, they benefit from both the occlusion query and the improved hierarchical depth buffer efficiency. Thus, the "hierarchical" in hierarchical occlusion query often refers to both the image-space depth buffer tree and the geometric bounding volume tree.

36.8 Sector-based Conservative Visibility

In this section we explore alternative uses of the spatial partitioning created by the partition operations that occur when building a BSP tree. These techniques, especially stabbing trees, were critical for real-time rendering of indoor environments from the early 1990s through the 2000s. They contain some beautiful computer science and geometric ideas. However, at the time of this writing they are passing out of favor because hierarchical occlusion culling allows more flexibility for managing dynamic and arbitrarily shaped environments. Although these algorithms can be applied for explicit point-to-point visibility tests, they are typically used to generate a **potentially visible set (PVS)** of primitives for a given viewpoint.

Recall that the leaves of a BSP tree are polygons that have been repeatedly clipped against planes, at least for the version that we have discussed in this chapter. If we assume that the original polygons, before all of the subdivision, were convex and planar, then the leaves must also be convex and planar.

Any point in space can be classified by some path through the tree, corresponding to whether it is in the positive or negative half-space of each node's splitting plane. That series of planes carves space into a convex polyhedron that contains the point. The space inside this polyhedron is called a **sector.** The polyhedron may have infinite volume if the point is near the edge of the scene.

Some of the faces of a sector correspond to leaves in the BSP tree. These polygons block visibility. Others are empty space. Those are called **portals** because they are windows to adjacent sectors. A convex space has a nice visibility property: All points within the space are visible to all other points. (For a pinhole camera, one may choose to then restrict that visibility by the field of view to the image frame.) From any point in a sector, one can look through any portal into any adjacent sector. However, in doing so, our visibility *within the adjacent sector* is restricted to a frustum that is the extrusion of the portal's boundary away from the viewer. If we look straight through a sector into another sector through a second portal, visibility becomes restricted by the intersection of the two portals' extrusions. After many portals, this frustum can become small; if one portal lies outside the frustum of another, the intersection of their frusta is empty. Most of these key observations are due to Jones [Jon71] and form a family of visibility determination algorithms.

Consider the graph in which sectors are nodes and portals are edges. A point is visible to another point only if it is reachable in this graph. Furthermore, we can detect cases where the specific geometry of adjacent sectors prevents looking straight through more than two portals. For example, in a grid of nonrefractive and nonreflective windows, we can look through a window to our north into an adjacent sector and then through its east window into another one; but we cannot look through that sector's south window because that would require bending a viewing ray into an arc. This corresponds to the case of the third sector's south window lying entirely outside the intersection for the frusta of the first two.

For an interesting scene there might be a tremendous number of sectors, so following the graph line of thinking might be inefficient. But if we exclude small objects such as furniture from the sector-building algorithm and only consider large objects such as walls when building sectors and portals, we may find a relatively small number of sectors [Air90]. Any approach to visibility computation that ignores the so-called small **detail objects** only provides conservative visibility, and must be succeeded by another algorithm for correctness—the depth buffer is a common choice here.

Explicitly, $V(P, Q)$ must be zero if no part of the sector containing point P is visible to the sector containing Q. Given some algorithm for determining sector-to-sector visibility, we can then conservatively approximate point and primitive visibility. Airey [Air90] pioneered a number of sector-to-sector visibility algorithms including ray casting and shadow volumes. The methods that he described are all correct with high probability for large numbers of samples, but are not all conservative; thus, the net visibility computation is not conservative under those approaches.

36.8.1 Stabbing Trees

Teller [Tel92] devised a closed-form, conservative, analytic algorithm for conservative visibility between sectors. His algorithm computes a BSP tree on n polygons in $O(n^2)$ operations to form the sectors. It then computes all possible straight-line **stabbing line** traversals through the sector adjacency graph until occlusion in worst-case $O(n^3)$ time, using a linear programming optimization framework. The full set of traversals from one sector is called a **stabbing tree.** In practice, the $O(n^3)$ asymptotic bound is misleading. There are many trivial rejection cases, such as when a portal is a backface to the viewer, and the constant factors are

fairly small. Teller observed $O(n)$ behavior in practice on complex indoor models. Nonetheless, the latter step must be performed for each pair of sectors, so the entire algorithm is slow enough to be an offline process that may take minutes or hours. When the algorithm terminates, each sector is annotated with a list of all other sectors that are conservatively visible to it.

At runtime in Teller's framework, a point in space is associated with a sector by walking the original BSP tree in typically $O(\log n)$ time. That sector then provides its list of all potentially visible sectors, which quickly culls most of the scene geometry. Individual point-to-point visibility tests by ray casting can then be performed for explicit visibility tests. Alternatively, all polygons (including detail objects) in the potentially visible sectors can be rasterized with a depth buffer for implicit visibility during rendering. The latter approach was introduced in the *Quake* video game in 1996 and quickly spread throughout the game industry.

36.8.2 Portals and Mirrors

Teller's stabbing trees require a long precomputation step that precludes their application to scenes with dynamic (nondetail) geometry. The Portals and Mirrors algorithm [LG95] is an alternative. It is simpler to implement and extends to both dynamic scenes and a notion of indirect visibility through mirrors. It revisits the frustum created when looking through a portal, but in the absence of the BSP tree and in the case where sectors may be nonconvex. That allows the sector geometry to change at runtime. The idea of the algorithm is to recursively trace the view frustum through portals, clipping it to each portal traversed. Figure 36.19 shows two visualizations of the algorithm. The first image is the camera's view, with clipping regions at portals and mirrors highlighted. The second image is a top view of the scene, showing how the frustum shrinks as it passes through successive portals and reflects when it hits a mirror.

Let the scene be represented as the sector polyhedra, the adjacency information between them, and the detail objects. Assume that at the beginning of an interactive sequence we know which sector contains the viewer, and that the viewer must move continuously through space (i.e., no teleportation!). Whenever the viewer moves through a portal, update the viewer state to retain a pointer to the sector that now contains the center of projection by following the adjacency information associated with that portal. This allows identifying the viewer's sector in $O(1)$ time during any frame.

To render a frame, let `clipPoly` initially be the screen-space rectangle bounding the viewport and `sector` be the viewer's current sector. Invoke the `portalRender(sector, clipPoly)` function from Listing 36.5. It will recursively render objects seen through portals, recursing when there are no new portals that can be seen. The `intersect` routine is simply 2D convex polygon-polygon intersection, which can be performed using the Sutherland-Hodgman algorithm.

For a scene in which the sectors *are* convex and there is no detail geometry, this algorithm provides exact visibility—no depth buffer is needed. A strength of the method is that if we do use a depth buffer, not only can it accommodate nonconvex portals and detail geometry, but also the general 2D clipping can be replaced with conservative rectangle clipping on the bounding box of `projPoly`. This may trigger a few extra recursive calls, but it dramatically simplifies the clipping/intersection process because we need only work with screen-space, axis-aligned rectangles.

Figure 36.19: (Top) View inside Fred Brooks' bedroom. There are two open doors and a mirror between them. The resultant portals are outlined in white and the mirrors are outlined in red. (Bottom) Schematic of visible regions for the observer from the top image. Note how the sight lines to the mirror give rise to a reflected visibility frustum that passes behind the viewer. (Courtesy of David Luebke ©1995 ACM, Inc. Reprinted by permission.)

The extension to mirrors is conceptually straightforward. We can model a mirror as a portal to a virtual world that resembles the real world but is flipped left-to-right. To implement this, augment `portalRender` to track whether the viewpoint has been reflected through an even or odd number of mirrors, and reflect the viewer through the plane of the mirror. Two complications of the mirrors are that they

Listing 36.5: Portal portion of the Portals and Mirrors algorithm.

```
function portalRender(sector, clipPoly):
   render all detail objects in sector

   for each face F in sector:
      if F is not a backface:
         if F is a portal:
            // Limit visibility by the bounds of the portal
            projPoly = project( clipToNearPlane(F.polygon) )
            newClip = intersect( projPoly, clipPoly )

            if newClip is not empty:
                 portalRender( F.nextSector, newClip )
         else:
            // F is an opaque wall
            render F.polygon
```

must lie on the faces of a sector (i.e., they cannot be detail polygons) and that for a nonconvex sector, care must be taken not to reflect geometry that is *behind* the mirror in front of it in the virtual world. That is, the virtual world seen through the mirror must be clipped against the plane of the mirror, until the next mirror is encountered. The issues that arise in this case are analogous to those for stencil-masked mirrors; see Kilgard's technical report [Kil99] for an explanation of how to perform the clipping and reflect the projection matrix.

36.9 Partial Coverage

There are several situations in which it is useful to extend binary visibility, also known as binary coverage, to **partial coverage** values on the interval $[0, 1]$. Many of these relate to the imaging model.

Consider the imaging model under which we defined binary coverage. For a pinhole camera with an instantaneous shutter there is a single ray that can transport light to each point on the image plane. The scene does not move relative to the camera because the exposure time is zero. At a single point Q on the image plane we can thus directly apply the binary visibility function to a point P in the scene.

Now consider a physically based lens camera model that has a nonzero shutter time and pixels of nonzero area. The radiant energy measured by a pixel in an image is an integral of the incident radiance function over the area of the pixel, solid angle subtended by the aperture, and exposure time. This creates a set of five parameters, sometimes labeled (x, y, u, v, t), that identify the path from a point $Q'(x, y, u, v, t)$ on the image plane through the lens to a point $P'(t)$ in the scene. We introduce the primes to distinguish these from the points with which we have previously been concerned.

The binary visibility function between points on P' and Q' may vary with the parameters. The **partial coverage** (a.k.a. **coverage**) of points in the set $P'(t)$ from points in the set $Q'(x, y, u, v, t)$ is the integral of the binary visibility function over all parameter variations within the domain of those sets. As the area-weighted average of binary visibility, partial coverage values are necessarily on the range $[0, 1]$. To extend the definition to whole surfaces, we can consider points on a surface $P'(i, j, t)$ parameterized by both surface location (i, j) and time t.

36.9.1 Spatial Antialiasing (*xy*)

Visibility under instantaneous pinhole projection, and thus coverage as well, between points P and Q is a binary value. However, coverage between a *set* of points in the scene and a *set* of points on the image plane can be fractional, since those sets give rise to many possible rays that may have different binary visibility results.

Of particular interest is the case where the region in the scene is a surface defined by values of the function $P'(i,j,t)$—a moving patch—and the region on the image plane is a pixel. For simplicity in definitions, assume the surface is a convex polygon so that it cannot occlude itself. We say that a surface defined by $P'(i,j,t)$ *fully covers* the pixel when the binary visibility function to all points of the form $Q(x,y,u,v,t)$ is 1 for all parameters. We say that the surface *partially covers* the pixel if the integral of the binary visibility function over the parameter space is less than 1.

Aliasing is, broadly speaking, caused by trying to store too many values in too few slots (see Figure 36.21 and Chapter 18). As in a game of musical chairs, some values cannot be stored. Aliasing arises in imaging when we try to represent all of the light from the scene that passes through a pixel using a single sample point (e.g., the one in the center). In this case, a single, binary visibility value represents the visibility of the whole portion of the surface that projects within the pixel area. The single sample covers infinitesimal area. Over that area, the binary visibility result is in fact accurate. But the pixel's area[2] is much larger, so the binary result is insufficient to represent the true coverage—there may be only fractional coverage. Rounding that fractional coverage to 0 or 1 creates inaccuracies that often appear in the form of a blocky image. Introducing a better approximation of partial coverage that considers multiple light paths can reduce the impact of this artifact. The process of considering more paths (or equivalently, samples) is thus called **antialiasing.**

Ideally, we'd integrate the incident radiance function over the entire pixel area, or perhaps the support of a sensor-response function, which may be larger than a pixel. For now, let's assume that pixels respond only to light rays that pass through their bounds and have uniform response regardless of where within the pixel we sample the light.

Some definitions will maintain precise languages when we distinguish between the value of a pixel and a value at a point within the pixel. Figure 36.20 shows a single primitive (a triangle) overlaid on a pixel grid. A **fragment** is the part of a pixel that lies within a given pixel. Each pixel contains one or more **samples,** which correspond to primary rays. To produce an image, we need to compute a color for each pixel. This color is computed from values at each sample. The samples, however, need not be computed independently. For example, all of the samples in the central pixel are completely covered by one fragment, so perhaps we could compute a single value and apply it to all of them. We refer to the process of computing a value for one or more samples as **shading,** to distinguish it from computing coverage, that is, which samples are covered by a fragment. Although our discussion has been couched in the language of physically based rendering, "shading" applies equally well to arbitrary color computations (e.g., for text or expressive rendering).

2. More precisely: the support of the measurement or response function for the pixel.

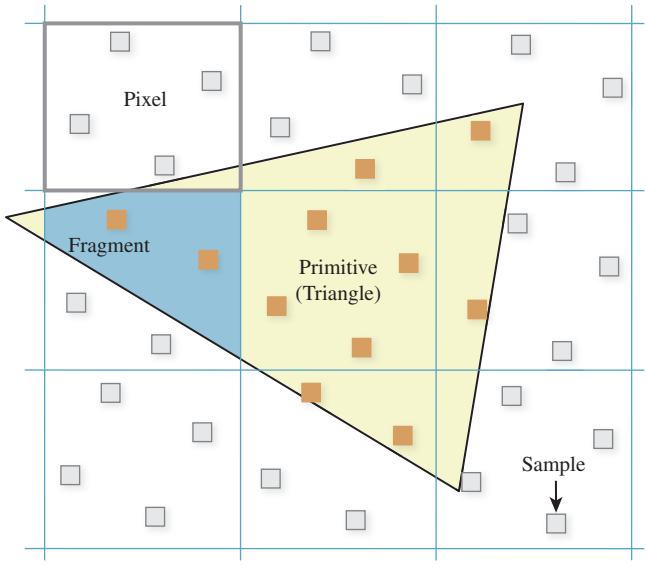


*Figure 36.20: A **pixel** is a rectangular region of the framebuffer. A **primitive** is the geometric shape to be rendered, such as a triangle. A **fragment** is the portion of a primitive that lies within a pixel. A **sample** is a point within a pixel. Coverage and shading are computed at samples (although possibly not at the same samples). The color of a pixel is determined by a **resolve** operation that filters nearby samples, for example, averaging all sample values within a pixel.*

The easiest way to convert a renderer built for one sample per pixel to approximate the integral of incident radiance across its area is to numerically integrate by taking many samples. This is the strategy employed by many algorithms.

Supersampled antialiasing (SSAA) computes binary visibility for each sample. Those samples at which visibility is one for a given fragment are then shaded independently. A single fragment may produce a different shade at each sample within a pixel. The final supersampled framebuffer is **resolved** to a one-value-per-pixel image by averaging all samples that lie within each pixel.

There are several advantages to SSAA. Increasing the sample count automatically increases the sampling of geometry, materials, lighting, and any other inputs to the shading computation. This aliasing arising from many sources can be addressed using a single algorithm. Implementing SSAA is very simple, and using it is intuitive and yields predictable results. One way to implement SSAA on a renderer that does not natively support it is to use an **accumulation buffer.** The accumulation buffer allows rendering in multiple one-sample-per-pixel passes and averaging the results of all passes. If each pass is rendered with a different subpixel offset, the net result is identical to that created using multiple samples within each pixel. The accumulation buffer implementation reduces the peak memory requirement of the framebuffer but increases the cost of transforming and rasterizing geometry.

The primary drawback of SSAA is that computing N samples per pixel is typically N times more expensive than computing a single sample per pixel, yet it may be unnecessary. In many situations, the shading result arising from illumination and materials either varies more slowly across a primitive than between primitives,

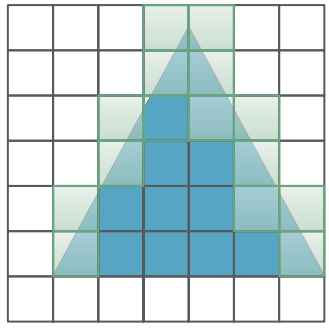


Figure 36.21: The triangle has binary visibility 1 on the pixels marked with solid blue and 0 on the pixels that are white. A binary value cannot accurately represent the triangle's visibility on the pixels along the triangle's slanted sides that are marked in light green. Attempting to compute binary visibility at those pixels necessarily produces aliasing.

or is at least amenable to filtering strategies such as MIP mapping so that it can be band-limited. This means that all of the samples within a pixel that are shaded by a single fragment are likely to have similar values. When this is the case, computing shading at each sample independently is inefficient.

The A-buffer [Car84] separates the coverage computation from the shading computation to address the overshading problem of SSAA. An A-buffer renderer computes coverage at each sample for a fragment. If at least one sample is covered, it appends the fragment to a list of fragments within that pixel. If that fragment completely occludes some fragment that was previously in the list, the occluded fragment is removed. When all primitives have been rasterized, shading proceeds on the fragments that remain within each pixel. The renderer computes a single shade per fragment and applies it to the covered and unoccluded samples within the pixel. This allows more accurate measurement of coverage due to geometric variation than shading. Because coverage is a simple and fixed computation, it can be computed far more efficiently than shading in general, and MSAA (covered shortly) enjoys the advantage that there is minimal overhead in increasing the number of samples per pixel. Thus, N samples may cost nearly the same as 1 sample.

The A-buffer can also be used to composite translucent fragments in back-to-front order. In this case, a fragment is not considered occluded if the occluding fragment is translucent. Fragments are sorted before shading and compositing. The drawback of the A-buffer is that the implementation requires variable space and significant logic within the coverage and rasterization process for updating the state. As a result, it is commonly used for offline software rendering. However, variations on the A-buffer that bound the storage space have recently been demonstrated to yield sufficiently good results for production use [MB07, SML11, You10]. These simply replace existing values when the maximum per-pixel sample list length is exceeded. A more sophisticated approach is to store some higher-order curve within a pixel that approximates all coverage and depth samples that have been observed. This approach has predominantly been applied to volumes of relatively homogeneous, translucent material, like smoke and hair [LV00, JB10].

Multisample antialiasing (MSAA) is similar to the A-buffer, but it applies a depth test and shades immediately after per-sample coverage computation to avoid managing per-pixel lists. This limits its application to spatial antialiasing, although stochastic approaches can extend temporal (motion blur), translucent, and lens (depth-of-field) sampling into the spatial domain [MESL10, ESSL10].

For each fragment, MSAA computes a coverage mask representing the binary visibility at each sample. In the most common application this is done by rasterizing and applying a depth buffer test with more samples than color pixels. If any sample was visible, an MSAA renderer then computes a single shading value for the entire pixel. The location of this shading sample varies between implementations; some choose the sample closest to the center of the pixel (regardless of whether it was visible!), and others choose the first visible sample. This single shading value is then used to approximate the shading at every sample in the pixel.

MSAA has several drawbacks. The first is that the shading computation must estimate the average value of the shade across the pixel rather than at a single point. In other words, MSAA leaves the problem of aliasing due to variation in

the shading function to the shader programmer—it only reduces aliasing due to geometric edges. For a shader that computes the color of a flat, matte surface with uniform appearance, this is a good approximation. In contrast, a shader that samples specular reflections off of a bumpy surface may exhibit significant undersampling because the shading function rapidly varies in space.

The second problem is that MSAA requires storage for a separate shade at each pixel. Thus, it has the same memory bandwidth and space cost as SSAA, even though it often reduces computation substantially.

Third, MSAA works best for primitives that are substantially larger than a pixel, and for pixels with many samples. If a mesh is tessellated so that each primitive covers only a small number of samples, then many shading computations are still needed.

Fourth and finally, MSAA does not work with the shading step of deferred shading algorithms. Deferred shading separates visible surface determination from shading. It operates in two passes. The first pass computes the inputs to the shading function at each sample and stores them in a buffer. The second pass iterates over each sample, reads the input values, and computes a shading value for the sample. When there are more samples than pixels, a final "resolve" operation is needed to filter and downsample the results to screen resolution. MSAA depends on amortizing a single shading computation over multiple samples. Under forward shading, that is possible because the coverage information for a single fragment is in memory at the same time that shading is being computed. Because deferred shading resolves coverage for all fragments before any shading occurs, the information about which samples within a pixel corresponded to the same fragment has been lost at shading time. Thus, one is faced with the two undesirable options of rediscovering which samples can share a shading result or of shading all samples by brute force.

Shading caches and decoupled shading [SaLY+08, RKLC+11, LD12] are methods currently under active research for adding a layer of indirection to capture the relationship between shading and coverage samples under a bounded memory footprint. The goal of this line of research is to combine the advantages of MSAA and deferred shading without necessarily creating an algorithmic framework that resembles either.

Coverage sampling antialiasing (CSAA) [You06] combines the strengths of the A-buffer and MSAA. It separates the resolution of coverage from *both* the resolution of shading and the resolution of the depth buffer. The key idea is that within each pixel there is a large set of high-resolution samples, but those samples are pointers to a small set of unique color (and depth, and stencil, etc.) values rather than explicit color values. For example, Figure 36.22 shows 16 sample locations within a pixel that reference into a table of four potentially unique color values.

CSAA allows a more accurate estimate of the area of the fragment within the pixel than the estimate of the occlusion of a fragment by other fragments. As each fragment is rasterized, CSAA computes a single shade per pixel, many binary visibility samples ignoring occlusion, and a few binary visibility samples taking occlusion into account. A small, fixed number of slots (usually four) are maintained within each pixel that are similar to entries in an A-buffer's list. Each slot stores the high-resolution coverage mask, shade value, and depth (and stencil) sample for a single fragment. Fragments are retained while there are some

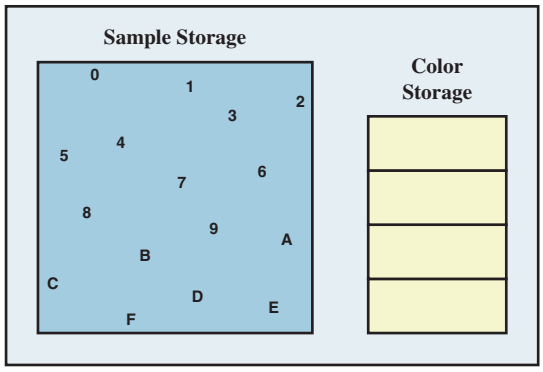


Figure 36.22: Depiction of the storage allocated for a single pixel under 16x CSAA [You06]. The large box on the left depicts the coverage samples. Each color sample contains a 2-bit integer that indexes into the four slots in the color table depicted by the smaller box on the right. When a fifth unique color is required, one of the four color samples is replaced. Thus, CSAA is heuristic and cannot guarantee correctness when many different surfaces color a pixel. (Courtesy of NVIDIA)

high-resolution coverage samples unique to them, and while there are available slots. When all of the slots are filled, some fragments are replaced. Thus, a pixel with 16 coverage samples and four slots may be forced to drop up to 12 fragments that actually should affect the pixel value. However, for primitives of screen-space area greater than one pixel it is often the case that only a few fragments will have nonzero coverage at a pixel. So, although CSAA is lossy, it often succeeds at representing fine-grained coverage using less storage than MSAA and the A-buffer, and with a low shading rate.

Analytic coverage historically predates CSAA, but can be thought of today as the limit of the CSAA process. Rather than taking many discrete samples of fragment coverage ignoring occlusion, one might simply compute the true coverage of a pixel by a fragment from the underlying geometric intersection. In the case of an instantaneous pinhole camera and simple surface geometry, this is a straightforward measure of the area of a convex polygon. The ratio of that area to the area of the pixel is the partial coverage of the pixel by that surface.

In the context of rasterization, for some primitives this computation can be amortized over many pixels so that computing analytic partial coverage information adds little cost to the rasterization itself. Primitives for which efficient partial coverage rasterization algorithms are known include lines [Wu91, CD05a] and circles [Wu91], which naturally extend to polygons and disks by neglecting one side.

Modern hardware rasterization APIs such as OpenGL and Direct3D include options to compute partial coverage for these primitives as part of the rasterization process. The implementation details vary. Sometimes the underlying process involves taking a large number of discrete samples rather than computing the true analytic result. Since there is a fixed precision for the result, the difference is irrelevant once enough samples are taken.

Analytic coverage has the advantage of potentially significantly higher precision than multiple discrete samples, with no additional memory or shading cost per pixel. It is often used for rasterization of thin lines and for polygons and curves in 2.5D presentation graphics and user interfaces.

The tremendous drawback of analytic coverage is that by computing a single partial-coverage value per fragment, it loses the information about which *part* of the pixel was covered. Thus, the coverage for two separate fragments within the pixel cannot be accurately combined. This problem is compounded when occlusion between fragments is considered, because that alters the net coverage mask of each. Porter and Duff's seminal paper [PD84] on this topic enumerates the ways that coverage can combine and explains the problem in depth (see Chapter 17). In practice, their OVER operator is commonly employed to combine fragment colors within a pixel under analytic antialiasing. In this case, there is a single depth sample per pixel, a single shade, and continuous estimation of coverage. Let α represent the partial coverage of a new fragment, s be its shading value, and d be the shading value previously stored at the pixel. If the new fragment's depth indicates that it is closer to the viewer than the fragment that previously shaded the pixel, then the stored shade is overwritten by $\alpha s + (1-\alpha)d$. This result produces correct shading *on average,* provided that two conditions are met. First, fragments with $\alpha < 1$ must be rendered in farthest-to-nearest order so that the shade at a pixel can be updated without knowledge of the fragments that previously contributed to it. Second, all of the fragments with nonunit coverage that contribute to a pixel must have uncorrelated coverage areas. If this does not hold, then it may be the case, for example, that some new fragment with $\alpha = 0.1$ entirely occludes a previous fragment with the same coverage, so the shade of the new one should overwrite the contribution of the former one, not combine with it.

In the case of 2.5D presentation graphics, it is easy to ensure the back-to-front ordering. The uncorrelated property is hard to ensure. When it is violated, the pixels at the edges between adjacent primitives in the same layer are miscolored. This can also occur at edges between primitives in different layers, although the effect is frequently less noticeable in that case.

Inline Exercise 36.7: Give an example, using specific coverage values and geometry, of a case where the monochrome shades from two fragments combine incorrectly at a pixel under analytic occlusion despite correct ordering.

36.9.2 Defocus (*uv*)

For a lens camera, there are many transport paths to each point on the image plane. The last segment of each path is between a point on the aperture and a point on the image plane. The "rays" between points on the image plane and points in the scene are not simple geometric rays, since they refract at the lens. However, we only need to model visibility between the aperture and the scene, since we know that there are no occluders inside the camera body.

For a scene point P there is a **pencil** of rays that radiate toward the aperture. For example, if the aperture is shaped like a disk, these rays lie within a cone. We can apply the binary visibility function to the rays within the pencil.

If there are no occluding objects in the scene and the camera is focused on that point, the lens refracts all of these rays to a single point on the image plane (assuming no chromatic aberration; see Chapter 26). The point Q on the image plane to which P projects in a corresponding pinhole camera thus receives full coverage from the light transported along the original pencil of rays.

If the point is still in focus, but an occluder lies between the scene point and the

lens in such a way that it blocks only some of the rays in the pencil from reaching the aperture, then only some of the light leaving the point toward the aperture will actually form an image. In this case, depicted in Figure 36.23, Q only receives partial coverage from P.

If there are no occluders but P is out of focus, then the light from the pencil originating at P is spread over a region on the image plane. Point Q now receives only a fraction of the light that it did in the in-focus case, so it now receives partial coverage by P.

Of course, a point may have partial coverage because it is both out of focus and partly occluded, and other sources of partial coverage can combine with these as well.

Note that in a sense any point is partially occluded by the camera case and finite lens—there are light rays from a scene point that would have struck the aperture had the lens only been larger. For a lens camera it is always necessary to know the size of the lens to compute the total incident light. The total light is proportional to the partial coverage, but we must know the size of the lens to compute the total power.

One way to compute partial coverage due to defocus is to sample visibility at many rays within the pencil and average the result. Because all the rays of the pencil share a common origin, there is an opportunity to amortize the cost of these binary visibility operations. The **packet tracing** and **rasterization** algorithms discussed in Chapter 15 leverage this observation.

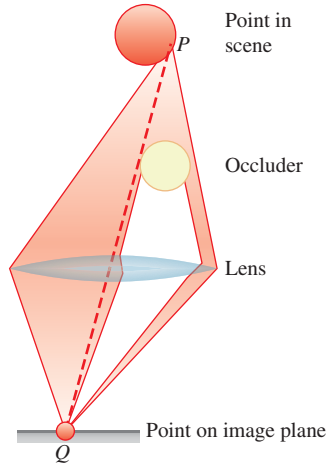


Figure 36.23: Partial occlusion of the lens leads to partial occlusion of the single point P at point Q.

36.9.3 Motion Blur (t)

Just as real cameras have nonzero aperture areas, they also have nonzero exposure times. This means that visibility can vary throughout the exposure. For any specific time, binary visibility may be determined between two points. The net visibility during an exposure will be the integral of that binary visibility over the exposure period, during which primitives may potentially cross between the points, producing an effect known as **motion blur.** For primary visibility in the presence of motion blur, we must consider the fact that the points for which we are testing visibility are on curves through space and time. This is easily resolved by performing all tests in camera space, where the primary rays are static with respect to time. Then we need only consider the motion of the scene relative to the camera.

Spatial data structures must be extended to represent motion. In particular, a spatial structure needs to bound the extrusion of each primitive along its motion path during the exposure. This step was not necessary for defocus because in that case we were performing ray-intersection queries that varied only the rays, not the triangles. When the triangles move with respect to each other a data structure built for a single position is no longer valid. A common strategy is to first replace each primitive with a conservative (and typically convex) bound on its motion. The second step is then to build the hierarchy on those proxies rather than the primitives themselves. When thin primitives rotate this can create excessively conservative bounds, but on the other hand, this approach is relatively straightforward to implement compared to considering the complex shapes that tightly bound rotating primitives.

This strategy generalizes to simply considering ray casting as a "four-dimensional" problem, where both rays and surfaces exist in a 4D space [Gla88]. The first three dimensions happen to be spatial and the fourth is temporal, but mathematically one can frame both the ray-intersection problem and the spatial data structure construction so that they are oblivious to this distinction. For

a bounding box hierarchy, the result is the same as was described in the previous paragraph. However, with this generality we can consider arbitrary bounding structures, such as a 4D BSP tree or bounding sphere hierarchy, which may provide tighter (if less intuitive) bounds.

36.9.4 Coverage as a Material Property (α)

A single geometric primitive may be used as a proxy for more complex geometry, for example, representing a window screen or a maple leaf as single rectangle. In this case, the small-scale coverage can be stored as a material property. By convention this property is represented by the variable α. Alpha is explicit coverage: $\alpha = 0$ is no coverage (e.g., areas outside of the leaf's silhouette), $\alpha = 1$ is total coverage (e.g., areas inside the leaf's silhouette), and $0 < \alpha < 1$ is partial coverage (e.g., the entire window screen may be represented with $\alpha = 0.5$).

Inline Exercise 36.8: Under what circumstances (e.g., position of the object relative to other objects or the camera) might the use of coverage-as-material-property lead to substantial errors in an image?

Note that a single α value is insufficient to represent colored translucency. A red wall viewed through a green wine bottle should appear black. Yet, if we model a bottle with a green surface as $\alpha = 0.5$, we will observe a brown wall through the bottle, whose color is 50% red and 50% green. This is a common artifact in real-time rendering. Offline rendering tends to model this situation more accurately with one coverage value per frequency of light simulated, or by sampling the light passing through the bottle as scattered rather than composited. This can also be done efficiently for real-time rendering by trading spatial precision for coverage precision [ME11].

An explicit coverage α must still be injected into the coverage resolution scheme for the entire framebuffer. Two common approaches are analytic and stochastic coverage. For the analytic approach, one simply renders in back-to-front order (with all of the limitations this implies) and explicitly composites each fragment as it is rendered, or injects the fragments into an A-buffer for it to process in that manner during resolution.

Stochastic approaches randomly set the fraction of coverage mask bits approximately equal to α and then allow another scheme such as MSAA to drive the shading and resolve operations. It is essential to ensure that the choice of which coverage bits are set is statistically independent for each fragment [ESSL10] because this is an underlying assumption of the compositing operations [PD84] implicit in the resolve filter.

Recent work shows that the quality of the resolve operation for stochastic antialiasing methods can be improved by filters more sophisticated than the typical box filter, although it has yet to be shown that the cost of complex resolve operations is less than the cost of simply increasing the number of samples [SAC+11].

36.10 Discussion and Further Reading

The classic paper "A characterization of ten hidden-surface algorithms" by Sutherland and Sproul [SSS74] surveys the state of the art for visibility in 1974, when the

typical goal was to determine the set triangles visible to the camera. None of those algorithms are in common use today for primary visibility on triangles. They've been replaced by the brute force z-buffer and by coarse hierarchical occlusion culling. However, it is worth familiarizing yourself with the algorithms Sutherland and Sproul surveyed for other potential applications. For example, we saw that BSP trees [SBGS69, FKN80] were originally designed to order polygons by depth so that they could be rendered perfectly using a variant of the painter's algorithm. BSP trees became popular for computing polygon-level visibility in game rendering engines in the 1990s and they are at the heart of a popular illumination algorithm [Jen01] used on most CG film effects today. Instead of ordering polygons for back-to-front rendering, BSP trees were found to be an effective way of carving up 3D space for $\mathrm{O}(\log n)$ access to surfaces and creating convex cells between which visibility determination is efficient. We anticipate that today's visible surface algorithms and data structures will find similar new uses in another 30 years, while newer and better approaches to visibility determination evolve as well.

36.11 Exercise

Exercise 36.1: Typically in Sutherland-Hodgman clipping, the polygon to be clipped is small compared to the boundary polygon, and it intersects at most one side of the boundary polygon; often the input polygon starts on the inside, crosses the boundary, remains outside for a while, and then recrosses the same boundary edge and returns inside. In this case, clipping an input polygon with n edges against a boundary with k edges involves generating and inserting only two intersection points, although it involves testing $O(nk)$ edge pairs for intersections. Build an example input in which there are $O(nk)$ intersection points computed and inserted.

This page intentionally left blank

Chapter 37

Spatial Data Structures

37.1 Introduction

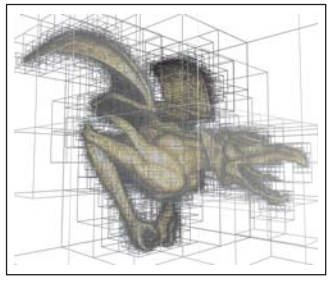


Figure 37.1: A gargoyle model embedded in an oct tree. The cube volume surrounding the model is recursively subdivided into smaller cubes, forming a tree data structure that allows efficient spatial intersection queries compared to iterating exhaustively over the triangles in the mesh. The boundaries of the cube cells are visualized as thin lines in this image. (Courtesy of Preshu Ajmera.)

Spatial data structures, such as the oct tree (Figure 37.1), are the multidimensional generalization of classic ordered data structures, such as the binary search tree. Because spatial data structures generally trade increased storage space for decreased query time, these are also known as **spatial acceleration data structures.** Spatial data structures are useful for finding intersections between different pieces of geometry. For example, they are used to identify the first triangle in a mesh that is intersected by a ray of light.

The development and analysis of spatial data structures is an area in which the field of computer graphics has contributed greatly to computer science in general. The practice of associating values with locations in spaces of various dimensions is of use in many fields. For example, many machine learning, finite element analysis, and statistics algorithms rely on the data structures originally developed for rendering and animation.

The ray-casting renderer in Chapter 15 represented surfaces in the scene using an unordered list of triangles. This chapter describes how to abstract that list with an interface. Once the implementation is abstracted, we can then change that implementation without constantly rewriting the ray caster. Why would we change it? Introductory computer science courses present data structures that improve the space and time cost of common operations. In this chapter we apply the same ideas to 3D graphics scenes. Of course, when comparing 3D points or whole shapes instead of scalars, we have to adjust our notions of "greater than" and "less than," and even "equals."

The original ray caster could render tens of triangles in a reasonable amount of time—maybe a few minutes, depending on the image resolution and your processor speed. A relatively small amount of elegant programming will speed this up by an amazing amount. Even a naive bounding volume hierarchy should enable your renderer to process millions of triangles in a few minutes. We hope you'll share the joy that we experienced when we first implemented this speedup. It is a great instance of algorithmic understanding leading directly to an impressive

and practical result—where a small amount of algorithmic cleverness brings the miraculous suddenly to hand.

To support the generalization of 1D to k-dimensional data structures and build intuition for the performance of the spatial data structures, we first describe classic 1D ordered data structures and how they are characterized.

Because triangle meshes are a common surface representation and rays are central to light transport, data structures for accelerating ray-triangle intersection receive particular attention in this chapter and in the literature.

We select our examples from the 2D and 3D spatial data structures most commonly encountered in computer graphics. However, the structures described in this chapter apply to arbitrary dimensions and have both graphics and nongraphics applications.

Read the first half of this chapter if you are new to spatial data structures; you may want to skip it if you are familiar with the concept and are looking for details. That first half of this chapter motivates their use, explains practical details of implementing in various programming languages, and reviews evaluation methodology for data structures.

The second half of this chapter assumes that you are familiar with the concepts behind spatial data structures. It explores four of the most commonly used structures: lists, trees, grids, and hash grids. For each, a few variations are presented; for example, the BSP, *k*d, oct, BVH, and sphere tree variants of "trees." The exposition emphasizes the tradeoffs between structures and how to tune a specific data structure once you have chosen to apply it.

Our experience is that it is relatively easy to understand the algorithms behind the spatial data structures, but converting that understanding to an interface and implementation that work smoothly in practice (e.g., ones that are efficient, convenient, maintainable, and general) may take years of experience.

37.1.1 Motivating Examples

Many graphics algorithms rely on queries that can be described by geometric intersections. For example, consider an animated scene containing a ball that is moving with constant velocity toward a pyramid of three boxes resting on the ground, as depicted in Figure 37.2. As the ball moves along its straight-line path, the ball traces a 3D volume called a **capsule,** which is a cylinder with radius equal to that of the ball and capped by two hemispheres bounding all of the trailing and leading sides of the sphere. For each discrete physical simulation time step, there is a capsule bounding all of the ball's locations during the time. The boxes are static, so the volume that they occupy remains constant. A collision between the moving sphere and the boxes will occur during some time step. To determine if the collision is in the current step, we can compute the geometric intersection of the capsule and the boxes. If it is empty, then at no time did the ball intersect the boxes, so there was no collision. If the intersection is nonempty, then there was a collision and the dynamics system can respond appropriately by knocking over the boxes and altering the ball's path.

Intersection computation is also essential for rendering. Some common intersection queries in graphics that arise from rendering are

- The *first* intersection of a light ray with a scene for ray casting or photon tracing
- The intersection of the camera's view frustum with the scene to determine which objects are potentially visible

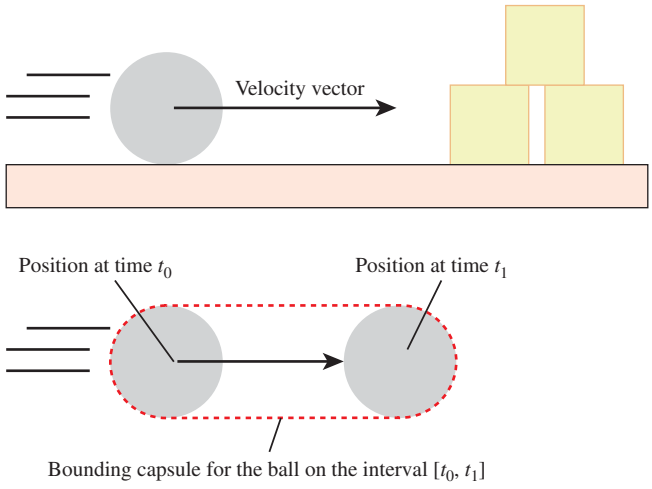


Figure 37.2: During any time interval, a sphere moving with constant velocity traces the volume of a capsule. If the intersection of the capsule and the boxes is not empty during a time interval, a collision occurred during that interval.

- The intersection of a ball[1] around a light source with a scene to determine which objects receive significant direct illumination from it
- The intersection of a ball with a set of stored incoming photon paths to estimate radiance in photon mapping

In the dynamics example of a scene with three boxes and a single moving sphere, it would be reasonable to compute the capsule at each time step and then iterate through all boxes testing for intersections with it. That strategy for collision detection would not scale to a scene with millions of geometric primitives. Informally, to scale well, we require an algorithm that can detect a collision in fewer than a linear number of operations in the number of primitives, n. We expect that using data structures such as trees can reduce the intersection-testing costs to $\mathrm{O}(\log n)$ in the average case, for example. The best asymptotic time performance that an algorithm could exhibit in a nontrivial case is $\mathrm{O}(m)$ for m actual intersections, since the intersections themselves must be enumerated. This is achievable if we place spatial distribution constraints on the input; for example, accepting an upper bound on the spatial density of primitives [SHH99]. Note that even with such constraints, $m = n$ in the worst case, so any spatial intersection query necessarily exhibits worst-case $\mathrm{O}(n)$ time complexity. Furthermore, obtaining optimal performance might require more storage space or algorithmic complexity than an implementation is able to support.

There is no "best" spatial data structure. Different ones are appropriate for different data and queries. For example, finding capsule-box intersections in a scene with four primitives lends itself to a different data structure than ray-triangle intersections in a scene with millions of rays and triangles. This is the same principle

1. A ball is the volume inside the sphere; technically, a geometric sphere $\mathbf{S}^k$ is the $(k-1)$ dimensional surface and not the k-dimensional volume within it. However, beware that intersections with balls are casually referred to as "sphere intersections" by most practitioners.

that holds for classic data structures: e.g., A hash table is neither better nor worse than a binary search tree in general; they are appropriate for different kinds of applications. With spatial data structures, a list of boxes may be a good fit for a small scene. For a large scene, a binary space partition tree may be a better choice.

The art of algorithm and system design involves selecting among alternative data structures and algorithms for a specific application. In doing so, one considers the time, space, and implementation complexity in light of the input size and characteristics, query frequency, and implementation language and resources. The rest of this chapter explores those issues.

37.2 Programmatic Interfaces

Spatial data structures are typically implemented as **polymorphic types.** That is, each data structure is parameterized on some primitive type that in this chapter we'll consistently name `Value`. For example, in most programming languages you don't just instantiate a list; instead, you make a list of something. These are known as **templated classes** in C++ and C# and **generics** in Java. Languages like Scheme, Python, JavaScript, and Matlab rely on runtime dynamic typing for polymorphic types, and languages like ML use type inference to resolve the polymorphism at compile time. The three popular graphics OOP languages Java/C++/C# use angle-bracket notation for this polymorphism, so we adopt it too (e.g., `List<Triangle>` is the name of a structure representing a list of triangles).

Throughout this chapter, we assume that `Value` is the type of the geometric primitive stored in the data structure. Common primitive choices include triangle, sphere, mesh, and point. A value must support certain spatial queries in order to be used in building general spatial data structures. We describe one scheme for abstracting this in Section 37.2.2.2. Briefly, a data structure maps keys to values. For a spatial data structure the keys are geometry and the values are the properties associated with that geometry. One frequently implements the key and value as different interfaces for the same object through some polymorphic mechanism of the implementation language. For example, a building object might present a rectangular slab of geometry as a **key** for arrangement within a data structure but also encodes as a **value** information about its surface materials, mass, occupants, and replacement cost for simulation purposes. Note that depending on the application, the key might assume radically different forms, such as a finely detailed mesh, a coarse bounding volume on the visible geometry, or even a single point at the center of mass for simulation purposes. The same value may be represented by different keys in different data structures, but extracting a specific class of key from a value must be a deterministic process for a specific data structure.

Beware that the use of the data-structure term "key" here is consistent with the general computer science terminology employed throughout the chapter, but it is uncommon in typical graphics usage. Instead, keys that are volumes are often called **bounding geometry, bounding volumes** [RW80], or **proxies.**

In this chapter, we use the variable name `key` to represent whatever form of geometric key is employed by the data structure at that location in the implementation. The type of the key depends on the data structure and application. We also use the name `value`, which will always have class `Value`.

In terms of an application interface, the methods implementing the intersection queries may be framed precisely to return the geometric intersections (see

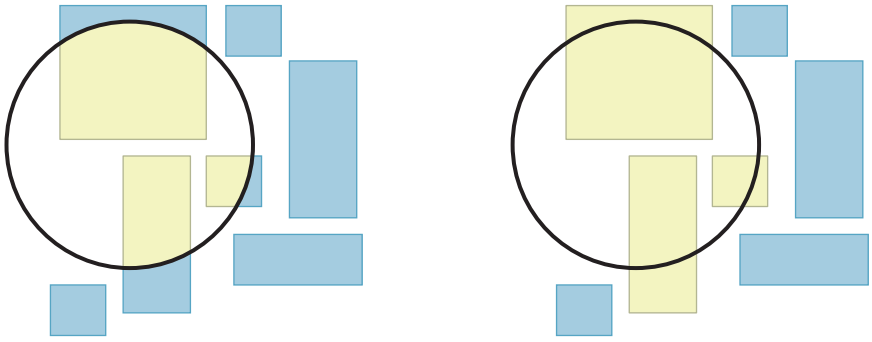


Figure 37.3: (Left) The geometric intersection of a ball with a set of boxes in 2D. (Right) A conservative result that may be more efficient to compute.

Figure 37.3, left) or conservatively to return *all* primitives containing the intersection geometry (see Figure 37.3, right). The latter is important because the intersections between simple shapes may be hard to represent efficiently. For example, consider the intersection of a set of triangles with a ball about a point light. The intersection probably contains many triangles that are cut by the surface of the ball. The resultant shapes are not representable as triangles. However, the rendering system that will use this information likely operates on triangles, since that was the representation chosen for the underlying scene. Creating a representation for triangles-cut-by-curves will create complexity in the system that only results in shapes that the renderer can't directly process anyway. In this case, a conservative query that returns all triangles that intersect the ball is more useful than a precise query that returns the geometric intersection of the scene and the ball.

To make the structures useful, traditional set operations such as insert, remove, and is-member are generally provided along with geometric intersection. The algorithms for those tend to be straightforward applications of nonspatial data structure techniques. In contrast, the intersection queries depend on the geometry. They are the core of what makes *spatial* data structures unique, so we'll focus on those.

37.2.1 Intersection Methods

Some of the most common methods of spatial data structures are those that are used to find the intersection of a set of primitives with a ball (the solid interior bounded by a sphere), an axis-aligned box, and a ray. Listing 37.1 gives a sample C++ interface for these.

Listing 37.1: Typical intersection methods on all spatial data structures.

```
template<class Value>
class SpatialDataStructure {
public:
  void getBallIntersection (const Ball& ball, std::vector<Value*>& result) const;
  void getAABoxIntersection(const AABox& box, std::vector<Value*>& result) const;
  bool firstRayIntersection(const Ray& ray, Value*& value, float& distance) const;
};
```

There are of course many other useful intersection queries, such as the capsule-box intersection from the collision detection example. For an application in which a particular intersection query occurs very frequently and affects the performance of the entire system, it may be a good idea to implement a data structure designed for that particular query.

In other cases, it may make more sense to perform intersection computations in two passes. In that case, the first pass leverages a general spatial data structure from a library to conservatively find intersections against some proxy geometry. The second pass then performs exhaustive and precise intersection testing against only the primitives returned from the first query. For the right kinds of queries, this combines the performance of the sophisticated data structure with the simplicity of list iteration.

For example, dynamics systems often step through short time intervals compared to the velocities involved, so the capsule swept by a moving ball may in fact not be significantly larger than the original ball. It can thus be reasonably approximated by a ball that encloses the entire capsule, as shown in Figure 37.4.

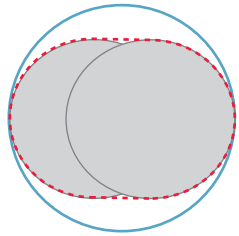


Figure 37.4: Approximating a short capsule (dashed line) with a bounding ball (solid line). To satisfy persistence of vision, the rendering frame rate for an animation is usually chosen so that objects move only a fraction of their own extent between frames. In this case, a static bounding ball around the path of a dynamic ball is not too conservative.

A generic ball-box intersection finds all boxes that are near the capsule, including some that may not actually intersect the capsule. If there are few boxes returned from this query, simply iterating through that set is efficient. In fact, the two-pass operation may be *more* efficient than a single-pass query on a special-purpose data structure for capsule-box intersection. This is because the geometric simplicity of a ball may allow optimizations within the data structure that a capsule may not admit.

Like a ball, an axis-aligned box and a ray have particularly simple geometry. Thus, while our three sample intersection query methods motivate the design patterns that could be applied to alternative query geometry, they are often good choices in themselves.

Some careful choices in the specifications of intersection query methods can make the implementation and the application particularly convenient. We now detail a particular specification as an example of a useful one for general application. Consider this as a small case study of some issues that can arise when designing spatial data structures, and add the solutions to your mental programmer's toolbox.

You may follow the exact specification given here at some point, but more likely you will apply these ideas to a slightly different specification of your own. You will also likely someday find yourself using someone else's spatial data structure API. That API will probably follow a different specification. Your first task will be to understand how it addresses the same issues under that specification, and how the differences affect performance and convenience.

Method `getBallIntersection`,

```
1 void getBallIntersection
2   (const Ball& ball,
3    std::vector<Value*>& result) const;
```

appends all primitives that overlap (intersect) the ball onto the result array.[2] It does not return the strict intersection itself, which would require clipping primitives to the ball in the general case.

2. In this chapter, "array" denotes a dynamic array data structure (e.g., `std::vector` in C++) to distinguish that concept from geometric vectors.

Appending to, rather than overwriting, the array allows the caller more flexibility. If the caller requires overwrite behavior, then it can simply clear the array itself before invoking the query. Some tasks in which overwriting is undesirable are accumulating the results of several queries, and applying the same query to primitive sets stored in different data structures.

We assume `getBallIntersection` returns only primitives that overlap the ball. The primitives stored in the data structure may contain additional information that is unnecessary for the intersection computation, such as reflectivity data for rendering or a network connection for a distributed application like a multiplayer game. They may also present different geometry to different subsystems. Thus, while a data structure may make conservative approximations for efficiency, it must ultimately query the primitive to determine exact intersections. Querying the primitives may be relatively expensive compared to the conservative approximations. An alternative interface to the ball intersection routine would return a conservative result of primitives that may overlap the ball, using only the conservative tests. This is useful, for example, if the caller intends to perform additional intersection tests on the result anyway.

Method `getAABoxIntersection`,

```
void getAABoxIntersection(const AABox& box, std::vector<Value*>& result) const;
```

is similar to `getBallIntersection`. It finds all primitives in the data structure that overlap the interior of the axis-aligned box named `box`. Overlap tests against an axis-aligned box are often significantly faster than those performed against an arbitrarily oriented box, and the box itself can be represented very compactly.

The use of axis-aligned boxes is important for more than primitive intersections. For data structures such as the *k*d-tree and grid that contain spatial partitions aligned with the axes, the axis-aligned box intersection against internal nodes in a data structure may be particularly efficient and implemented with only one or two comparisons per axis.

Method `firstRayIntersection`,

```
bool firstRayIntersection(const Ray& ray, Value*& value, float& distance) const;
```

tests for intersections between the primitive set and a ray (Figure 37.5). There are three possible results.

1. There is no intersection. The method returns `false` and the referenced parameters are unmodified.
2. The intersection closest to the ray origin is at distance r and $r \geq$ `distance`. The method returns `false` and the referenced parameters are unmodified.

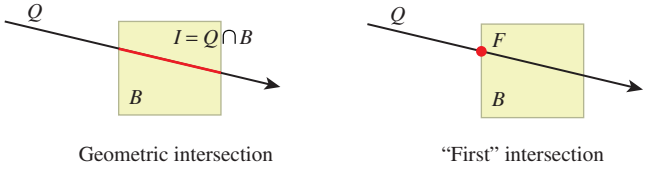


Figure 37.5: (Left) The geometric intersection I of a 2D query ray Q with box B. (Right) The point F that we call the first intersection between the ray and box, that is, the point on the geometric intersection that is closest to the ray origin.

3. The intersection closest to the ray origin is at distance $r <$ `distance`. The method returns `true`. Parameter distance is overwritten with r and `value` is a pointer to the primitive on which the intersection lies.

This interface is motivated by the common applications of the query method, such as the one from Listing 37.2. For example, consider the cases of ray casting for eye rays, light rays (photons), moving particles, or picking (a.k.a. selecting) an object whose projection lies under a mouse pointer. In each case, there may be many objects that have a nonzero intersection with the ray, but the caller only needs to know about the one producing the intersection closest to the ray origin. These objects might be stored in multiple data structures—perhaps distinguishing static and dynamic objects or ones whose different geometry is better suited to different data structures. Given an intersection query result from one data structure, the query to the second data structure need not search any farther than the previously found intersection. The caller almost always will overwrite the previously closest-known distance, so it makes sense to simply update that in place. Listing 37.2 demonstrates a terse yet readable implementation of this case. Note that there's potential for a subtle bug in this code—if the second ray intersection were accidentally written as `hit = hit || dynamicObjects...`, then *any* intersection with a static object would preclude ever testing for a closer intersection with a dynamic object. Misuse of the early-out behavior of the logical OR operator is a danger not limited to ray intersection or computer graphics, of course!

Listing 37.2: Typical application of `FirstRayIntersection` *in a ray-tracing renderer.*

```
Radiance rayTrace(const Ray& ray) {
  float distance = INFINITY;
  Value* ptr = NULL;

  bool hit = staticObjects.firstRayIntersection (ray, ptr, distance);
  hit = dynamicObjects.firstRayIntersection(ray, ptr, distance) || hit;

  if (hit) {
       return shade(ptr, ray.origin + distance * ray.direction);
  } else {
      return Radiance(0);
  }
}
```

Even when the high-level scene management code creates only one spatial data structure, there are likely many spatial data structures in the program. This is because, like any other sophisticated data structures, complex spatial data structures tend to be implemented using multiple instances of simpler spatial data structures. For example, the contents of each node in a spatial tree are typically stored in a spatial list, and the child subtrees of that node are simply other instances of spatial trees. Intersection methods like the ones proposed here that are designed to accumulate the results from multiple structures make it easy to design complex spatial data structures.

In addition to the convenience for the caller of passing a distance limit that will be updated, there are some applications for which there is inherently some limit to how far the caller wants to search. Examples include picking with a range-limited virtual tool or casting a shadow ray to a light source. Communicating the search limit to the data structure allows it to optimize searching for the intersection.

For ray-triangle intersection, `firstRayIntersection` is often extended to return the barycentric coordinates of the intersection location on the triangle hit. As described in Section 9.2, these three weights are sufficient to reconstruct both the location on the triangle where the ray first intersects it and any shading parameters interpolated from the vertices of the triangle. Returning the barycentric coordinates is not necessary because, given the distance along the ray to the intersection, the caller could reconstruct them. However, they are naturally computed during the intersection computation, so it is efficient and convenient to pass them back to the caller rather than imposing the cost of recomputing them outside the query operation.

Some data structures can resolve the query of whether *any* intersection exists for $r < $ `distance` faster than they can find the *closest* one. This is useful for shadow and ambient occlusion ray casting. In that case a method variant that takes no value parameter and does not update distance is a useful extension.

37.2.2 Extracting Keys and Bounds

We want to be able to make data structures that can be instantiated for different kinds of primitives. For example, a tree of triangles and a tree of boxes should share an implementation. This notion of the tree template as separate from a specific type of tree is called **polymorphism.** It is something that you are probably very familiar with from classic data structures. For example, `std::vector<int>` and `std::vector<std::string>` share an implementation but are specialized for different value types.

For spatial data structures, we therefore require some polymorphic interface both for each data structure and for extracting a **key** from each `Value`. The choice of interface depends on the implementation language and the needs of the surrounding program.

37.2.2.1 Inheritance

In an object-oriented language such as Java, one typically uses inheritance to extract keys, as shown in Listings 37.3 and 37.4 (the latter listing has the example).

Listing 37.3: A Java inheritance interface for expressing axis-aligned bounding box, sphere, and point keys on a primitive and responding to corresponding conservative intersection queries.

```
1  public interface Primitive {
2     /**  Returns a box that surrounds the primitive for use in
3          building spatial data structures. */
4     public void getBoxBounds (AABox bounds);
5
6     public void getSphereBounds (Sphere bounds);
7
8     public void getPosition (Point3 pos);
9
10
11    /** Returns true if the primitive overlaps a box for use
12        in responding to spatial queries. */
13    public bool intersectsBox (AABox box);
14
15    public bool intersectsBall (Ball ball);
16
```

```
    /** Returns the distance to the intersection, or inf if
        there is none before maxDistance */
    public float findFirstRayIntersection(Ray ray, float maxDistance);
}

public class SomeStructure<Value> {
    ...
    void insert(Value value) {
        Point3 key = new Point3();
        value.getPosition(key);
        ...
    }
}
```

Listing 37.4: One possible implementation of a triangle under an inheritance-based scheme. The `getBoxBounds` *implementation computes the bounds as needed; an alternative is to precompute and store them.*

```
public class Triangle implements Primitive {

  private Point3 _vertex[3];

  public Point3 vertex(int i) {
    return _vertex[i];
  }

  public void getBoxBounds(AABox bounds) {
    bounds.set(Point3::min(vertex[0], vertex[1], vertex[2]),
               Point3::max(vertex[0], vertex[1], vertex[2]));
  }
  ...

}
```

Inheritance is usually well understood by programmers working in an object-oriented language. It also keeps the implementation of program features related to a `Value` within that `Value`'s class. This makes it a very attractive choice for extracting keys. The simplicity comes at a cost in flexibility, however. Using an inheritance approach, one cannot associate two different key extraction methods with the same class, and the needs of a spatial data structure impose on the design of the `Value` class, forcing them to be designed concurrently.

37.2.2.2 Traits

A C++ implementation might use a **trait** data structure in the style of the design of the C++ Standard Template Library (STL). In this design pattern, a templated trait class defines a set of method prototypes, and then specialized templates give implementations of those methods for particular `Value` classes. Listing 37.5 shows an example of one such interface named `PrimitiveKeyTrait` that supports box, ball, and point keys. Below that definition is a specialization of the template for a `Triangle`, and an example of how a spatial data structure would use the trait class to obtain a position key from a value.

Listing 37.5: A C++ trait for exposing axis-aligned bounding box, sphere, and point keys from primitives.

```
template<class Value>
class PrimitiveKeyTrait {
public:
   static void getBoxBounds (const Value& primitive, AABox& bounds);
   static void getBallBounds (const Value& primitive, Ball& bounds);
   static void getPosition (const Value& primitive, Point3& pos);

   static bool intersectsBox (const Value& primitive, const AABox& box);
   static bool intersectsBall(const Value& primitive, const Ball& ball);
   static bool findFirstRayIntersection(const Value& primitive, const Ray& ray,
       float& distance);
};

template<>
class PrimitiveKeyTrait<Triangle> {
public:
   static void getBoxBounds(const Triangle& tri, AABox& bounds) {
       bounds = AABox(min(tri.vertex(0), tri.vertex(1), tri.vertex(2)),
                   max(tri.vertex(0), tri.vertex(1), tri.vertex(2)));
   }
   ...
};

template< class Value, class Bounds = PrimitiveKeyTrait<Value> >
class SomeStructure {
    ...
    void insert(const Value& value) {
        Box key;
        Bounds<Value>::getBoxBounds(value, key);
        ...
    }
};
```

Overloaded functions are a viable alternative to partial template specialization in languages that support them. An example of providing an interface through overloading in C++ is shown in Listing 37.6. This is similar to the template specialization, but is a bit more prone to misuse because some languages (notably C++) dispatch on the compile-time type instead of the runtime type of an object. (ML is an example of a language that dispatches on runtime type.) If mixed with inheritance, overloading can thus lead to semantic errors.

Listing 37.6: A C++ trait implemented with overloading instead of templates.

```
void getBoxBounds(const Triangle& primitive, AABox& bounds) { ... }
void getBoxBounds(const Ball& primitive, AABox& bounds) { ... }
void getBoxBounds(const Mesh& primitive, AABox& bounds) { ... }
...

template<class Value>
class SomeStructure {
    ...
    void insert(const Value& value) {
        Box key;
        // Automatically finds the closest overload
        getBoxBounds(value, key);
        ...
    }
};
```

Traits can also be implemented at runtime in languages with first-class functions or closures. This forgoes static type checking but allows the flexibility of the design pattern with less boilerplate and in more languages. For example, Listing 37.7 uses the trait pattern in Python and depends on dynamic typing and runtime error checks to ensure correctness.

Listing 37.7: A Python trait for exposing axis-aligned bounding box, sphere, and point keys from primitives.

```
def getTriangleBoxBounds(triangle, box):
    box = AABox(min(tri.vertex(0), tri.vertex(1), tri.vertex(2)),
                max(tri.vertex(0), tri.vertex(1), tri.vertex(2)))
}

class SomeStructure:
    _bounds = null

    def __init__(self, boundsFunction):
        self._bounds = boundsfunction

    ...
    def insert(self, value):
        Box key;
        self._bounds(value, key)
        ...
    }
};

SomeStructure s(getTriangleBoxBounds)
```

The trait design pattern for extracting keys has three main advantages. Traits allow the data structure implementor to make the data structure work with `Value` classes that predate it. For example, you can create your own new binary space partition tree spatial data structure and write a trait to make it work with the `Triangle` class from an existing library that you cannot modify. A related advantage is that traits move the complexity of the key-extraction operation out of the `Value` class.

Another advantage of traits is that they allow instances of data structures with the same `Value` to use different traits. For example, you may wish to build one tree that uses the vertex of a triangle that is closest to the origin as its position key, and another that uses the centroid of the triangle as its position key. However, if `getPositionKey` is a method of `Triangle`, then this is not possible. Under the pattern shown in Listing 37.5, these would be instantiated as `SomeStructure<Triangle, MinKey>` and `SomeStructure<Triangle, CentroidKey>`.

A disadvantage of traits compared to inheritance is that traits separate the implementation of a `Value` class into multiple pieces. This can increase the cost of designing and maintaining such a class.

Traits also involve more complicated semantics and syntax than other approaches. This is particularly true for the variant shown in Listing 37.5 that uses C++ templates. Many C++ programmers have never written their own trait-based classes, or even their own templated classes. Almost all have *used* templated classes and traits, however. So this design pattern significantly increases the barrier to creating a new kind of data structure and slightly increases the barrier to creating a new kind of primitive, but introduces little barrier to *using* a data structure.

Overall, we find the trait design necessary in practice for efficiency and modularity, but we regret the modularity lost in `Value` classes. We regard traits as a necessary evil in practice.

37.3 Characterizing Data Structures

Classic ordered structures contain a set of elements. Each element has **value** and a single integer or real number **key.** For example, the values might be student records and the keys the students' grades. We say that the data structures are **ordered** because there is a total ordering on the keys.

Regardless of their original denotation, one can interpret the key as a position on the real number line. This leads to our later generalization of multidimensional keys representing points in space.

There are many ways to analyze data structures. In this chapter, you'll see two kinds of analysis intermixed. It is impossible to give a one-size-fits-all characterization of these structures and advice on which is "best"—it depends on the kind of data you expect to encounter in the scene. We therefore sketch asymptotic analyses and offer practical considerations of how they apply to real use. The conclusions of each section aren't really the goal. Instead, the issues raised in the course of reaching them are what we want you to think about and apply to problems.

We do recognize two usage patterns and prescribe the following high-level advice in selecting data structures.

- **When using data structures in a generic way,** trust asymptotic bounds ("big-O"). Generic use means for a minor aspect of an algorithm, for fairly large problems, and where you know little about the distribution of keys and queries. Trusting bounds means, for example, that operations on trees are often faster than on lists at comparable storage cost. Parameterize the bound on the factor that you really expect to dominate, which is often the number of elements.
- **When considering a specific problem** for which you have some domain knowledge and really care about performance, use all of your engineering skill. Consider the actual kinds of scenes/distributions and computer architecture involved, and perform some experiments. Perhaps for the size of problem at hand the scattered memory access for a tree is inferior to that of an array, or perhaps the keys fall into clusters that can be exploited.

To make the analysis tools concrete and set up an analogy to spatial data structures, we show an example on two familiar classic data structures in the following subsections. Consider two alternatives for storing n values that are student records in a course database at a college, where each record is associated with a real-number key that is the student's grade in the course.

The first structure to consider is a **linked list.** Although we say that a list is an "ordered" data structure, recall that the description applies to the fact that any two *keys* have a mathematical ordering: less than, greater than, or equal to. The order of elements within the list itself will be arbitrary. More sophisticated data structures exploit the ordering of the keys to improve performance.

The second data structure is a balanced **binary tree.** The elements within the tree are arranged so that every element in the right subtree of a node contains elements whose keys are larger than or equal to the key at the node. The left subtree of a node contains elements whose keys are less than or equal to the key at the node.

37.3.1 1D Linked List Example

The list requires storage space proportional to n, the number of student records. The exact size of the list is probably n times the sum of the size of one record and the size of one pointer to the next record, plus some additional storage in a wrapper class, such as the one in Listing 37.8. We describe this as the list occupying $O(n)$ space, which means that for sufficiently large n the actual size is less than or equal to $c \cdot n$ for some constant c.

Concretely, consider the time cost of finding student records in the list that lie within a continuous range of grades. If we think of the records as distributed along a number line at points indicated by their keys, the find operation is equivalent to finding the geometric intersection of the desired grade interval and the set of records. This geometric interpretation will later guide us in building higher-dimensional keys, where we want to perform intersection queries against higher-dimensional shapes such as balls and boxes.

To find records intersecting the grade interval, we must examine all n records and accept only those whose keys lie within the interval. Note that because the list index is independent of the key, we must *always* look at all n records. There is probably some overhead time for launching the find operation and the time cost of each comparison depends on the processor architecture. These make it hard to predict the exact runtime, but we can characterize it as $O(n)$ for a single find operation.

When employed with care, the big-O notation is useful for characterizing the asymptotic growth of the data structure without the distraction of small overhead constants. The important idea is that for lists that we are likely to encounter in practice, if we double the length of the list we approximately double the memory requirement. That is, if the list has length 100, we probably don't care about the small overhead cost of storing a constant number of extra values.

When we are ready to implement a system, with some idea of the data's size and capabilities of the hardware, we augment the asymptotic analysis with more detailed engineering analysis. At this stage it is important to consider the impact of constant space and time factors. For example, say that our list stores a few kilobytes of data in addition to the records. This might be information about a freelist buffer pool for fast memory allocation, the length of the list, and an extra pointer to the tail of the list for fast allocation, as described in Listing 37.8.

Listing 37.8: Sample templated list class.

```
template<class Value>
class List {
   class Node {
   public:
      float key;
      Value value;
      Node* next;
   };

   Node* head;
   Node* tail;
   int length;
   Node* freelist;
   ...
};
```

Under this list representation, a list of three small elements might require almost the same amount of space as a list of six elements. For small lists, we'll find that the length of the freelist dominates the total space cost of the list.

Under an engineering analysis we should thus extend our implementation-independent asymptotic analysis with both the constant factors and the newly revealed parameters from the implementation, such as the size of the freelist and the overhead of accessing a record.

37.3.2 1D Tree Example

A binary search tree exploits the ordering on the keys to increase the performance of some find operations. In doing so it incurs some additional constant performance overhead for every search and increases the storage space.

The motivation for using a tree is that we'd like to be able to find all students with grades in some interval and do so faster than we could with the list. The tree representation is similar to the list, although there are two child pointers in each node. So the space bounds are slightly higher by a constant factor, but the storage cost for n elements is still $\mathrm{O}(n)$.

The height of a balanced binary tree of n students is $\lceil \log_2 n \rceil$. If there is only one student whose grade is in the query interval, then we can find that student using at most $\lceil \log_2 n \rceil$ comparisons. Furthermore, if that student's record is at an internal (versus leaf) node of the tree, it may take as few as three comparison operations to locate the student and eliminate all others.

Thus, for the case of a single student in the interval, the search takes worst-case $\mathrm{O}(\lg n)$ time, which is better than the list. In practice, the constants for iterating through a tree are similar to those for iterating through a list, so it is reasonable to expect the tree to always outperform the list.

Depending on the size of the interval in the query and the distribution of students, there may be more than one student in the result. This means that the time cost of the find operation is **output-sensitive,** and the size of the output appears in the bound. If there are s students that satisfy the query, the runtime of the query is necessarily $\mathrm{O}(s + \lg n)$. Since $0 \leq s \leq n$, in the worst case *all* students may be in the output. Thus, the tightest upper bound for the general case is $\mathrm{O}(n)$ time.

We could try to characterize the time cost based on the distribution of grades and queries that we expect. However, at that point we're mixing our theory with engineering and we're unlikely to produce a bound that informs either theory or practice. Once we know something about the distribution and implementation environment, we should perform back-of-the-envelope analysis with the actual specifications or start performing some experiments.

At a practical level, we might consider the cache coherence implications of chasing tree pointers through memory versus sequential access for a tree packed into an array as a vector heap, or a list packed into an array. We might also look at the complexity of the data structure and the runtime to build and update the tree to support the fast queries.

In the case where queries and geometry are unevenly distributed, balancing the tree might not lead to optimal performance over many queries. For example, say that there are many students whose grades cluster around 34 but we don't expect to query for grades less than 50 very often. In that case, we want a deep subtree for grades less than 50 so that students with grades greater than 50 can appear near the root. Nodes near the root can be reached more quickly and are more likely to stay in cache.

37.4 Overview of *k*-dimensional Structures

The remainder of this chapter discusses four classes of spatial data structures. These correspond fairly closely to *k*-dimensional generalizations of classic one-dimensional lists, trees, arrays, and hash tables under the mapping shown in Table 37.1. In that table, the actual intersection time depends on the spatial distribution of the elements and the intersection query geometry, but the big-O values provide a useful bound.

Table 37.1: Approximate time and space costs of various data structures containing n elements in k dimensions. The grids have g subdivisions per axis, which is not modeled by these expressions.

1D Data Structure	*k*d Analog	Intersection Time	Total Space
List	**List and Array**	$O(n)$	$O(n)$
Tree	**BVH and BSP tree**	$O(\log n)$	$O(n)$
Array of cells	**Grid of cells**	$O(n/g^k)$	$O(g^k + n)$
Hash table	**Hash grid**	$O(1)$	$O(n)$

We can build some intuition for the runtimes of spatial data structures: Operations should take linear time on lists, log time on trees, and constant time on regular grids. The space requirements of a regular grid are problematic because the grid allocates memory even for areas of the scene that contain no geometry. However, using a hash table, we can store only the nonempty cells. Thus, we can expect list, tree, and hash grid structures to require only a linear amount of space in the number of primitives that they encode.

With some knowledge of the scene's structure, one can often tune the data structures to achieve amortized constant factor speedups through architecture-independent means. These might be factors of two to ten. Those are often necessary to achieve real-time performance and enable interaction. They may not be worthwhile for smaller data sets or applications that do not have real-time constraints.

All of these observations are predicated on some informal notions of what is a "reasonable" scene and set of parameters. It is also possible to fall off the asymptotic path if the scene's structure is poor. In fact, you can end up making computations *slower* than linear time. The worst-case space and time bounds for many data structures are actually unbounded. To address this, for each data structure we describe some of the major problems that people have encountered. From that you'll be able to recognize and address the common problems. That is, the time and space bounds given in the table are only to build intuition for the general case. As discussed in the previous section, one has to assume certain distributions and large problems to prove these bounds, and doing so is probably reductive. Understanding your scenes' geometry distributions and managing constant factors are important parts of computer graphics. Concealing those behind assumptions and abstractions is useful when learning about the data structures. Rolling back assumptions and breaking those mathematical abstractions are necessary when applying the data structures in a real implementation.

In addition to algorithmic changes and architecture-independent constant-factor optimizations, there are some big (maybe 50*x* over "naive" implementations) constant-factor speedups available. These are often achieved by minding details like minimizing memory traffic, avoiding unnecessary comparisons, and

exploiting small-scale instruction parallelism. Which of these micro-optimizations are worthwhile depends on your target platform, scenes, and queries, and how much you are willing to tailor the data structure to them. We describe some of the more timeless conventional wisdom accumulated over a few decades of the field's collective experience. Check the latest SIGGRAPH course notes, EGSR STAR report, and books in series like *GPU Pro* for the latest advice on current architectures.

37.5 List

A 1D **list** is an ordered set of values stored in a way that does not take advantage of the keys to lower the asymptotic order of growth time for query operations. For instance, consider the example of a mapping from student grades to student records encoded as a linked list that is described in Section 37.3.1. Each element of the list contains a record with a grade and other student information. Searching for a specific record by grade takes n comparisons for a list of n records if they are unordered. At best, we could keep the list sorted on the keys and bring this down to expected $n/2$ comparisons, but it is still linear and still has a worst case of n comparisons.

For the *unsorted* 1D list, there's little distinction between the expected space and time costs of a dynamic array (see Listing 37.9) and a linked-list implementation (see Listing 37.8). For the *sorted* case, the array admits binary search while the linked list simplifies insertion and deletion. The array implementation is good for small sets of primitives with small memory footprints each; the linked list is a better underlying representation for primitives with a large memory footprint, and even more sophisticated spatial data structures are preferred for larger sets.

Listing 37.9: C++ implementation of a spatial list, using an array as the underlying structure.

```
template<class Value, class Trait = PrimitiveKeyTrait<Value> >
class List {
  std::vector<Value> data

public:

  int size() const {
    return data.size();
  }

  /* O(1) time */
  void insert(const Value& v) {
    data.push_back(v);
  }

  /* O(1) time */
  void remove(const Value& v) {
    int i = data.find(v);
    if (i != -1) {
      // Remove the last
      data[i] = data[data.size() - 1];
      data.pop();
    }
  }

  ...
};
```

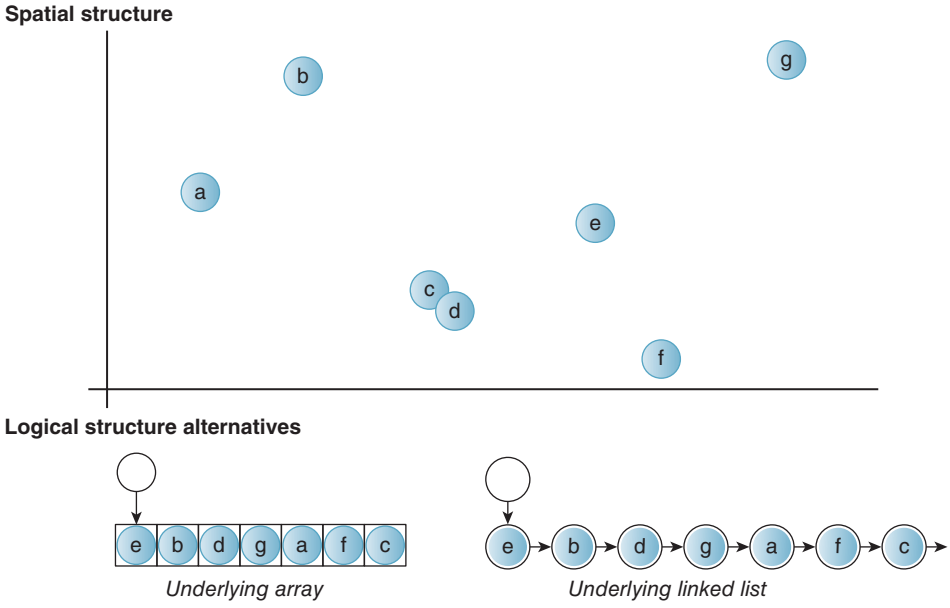


Figure 37.6: A set of points distributed in 2D and two logical data structures describing them with list semantics. The list interface on the left relies on an underlying array. The one on the right uses an underlying linked list. Note that the order of elements within the list is arbitrary.

Now consider the implications of generalizing from a 1D key such as a grade to a higher-dimensional key such as a 2D location. Figure 37.6 shows a sample 2D data set with seven points labeled *a*–*g* and two alternative list implementations corresponding to those data.

Compared to a 1D list, there is no longer a total ordering on keys. For example, there is no general definition of "greater" between (0, 1) and (1, 0) the way that there is between 3 and 6. Note that this problem occurs because the *key* has two or more dimensions. There are still many cases in computer graphics where one has *values* that describe 3D data and associates those with 1D keys like the depth of each object in a scene from the camera. One-dimensional data structures remain as useful in graphics as in any field of computer science or software development.

Given a set of n values each paired with a kd key (Figure 37.6), a list data structure backed by a linked list or array implementation requires $O(n)$ space. In practice, both implementations require space for more than just the n elements if the data structure is dynamic. In the linked list there is the overhead of the link pointers. A dynamic array must allocate an underlying buffer that is larger by a constant factor to amortize the cost of resizing.

Because we cannot order the values in an effective way for general queries, a query such as ray or box intersection requires n individual tests (see Listings 37.10–11). It must consider each element, even after some results that satisfy the query have been obtained. We could imagine imposing a specific ordering such as sorting by distance from the origin or by the first dimension, but unless we know that our queries will favor early termination under that sorting, this cannot even promise a constant performance improvement.

Listing 37.10: C++ implementation of ray-primitive intersection in a list. The method signature choices streamline the implementation.

```
/* O(n) time for n = size() */
bool firstRayIntersection(const Ray& ray, Value*& value, float& distance) const {
  bool anyHit = false;

  for (int i = 0; i < data.size(); ++i) {
    if (Trait::intersectRay(ray, data[i], distance)) {
       // distance was already updated for us!
       value = &data[i];
       anyHit = true;
    }
  }

  return anyHit;
}
```

Listing 37.11: C++ implementation of conservative ball-primitive intersection in a list.

```
/* O(n) time for n = size() */
void getBallIntersection(const Ball& ball, std::vector<Value*>& result) const {
  for (int i = 0; i < data.size(); ++i) {
    if (Trait::intersectsBall(ray, data[i])) {
       result.push_back(&data[i]);
    }
  }
}
```

Both structures allow insertion of new elements in amortized $O(1)$ time. The linked list prefers insertion at the head and the array at the end. Finding an element to delete is a query, so it takes n operations. Once found, the linked list can delete the element in $O(1)$ time by adjusting pointers. The array can *also* delete in $O(1)$ time—it copies the last element over the one to be deleted and then reduces the element count by 1. Because the array is unordered, there is no need to copy more elements. It is critical, however, that the array and its contents be *private* so that there can be no external references to the now-moved last entry.

There may be some advantage to the array's packing of values in a cache-friendly fashion, but only if the elements are small. If the values are large, then they may not fit in cache lines anyway, and the cost of copying them during insertion and removal may dwarf the main memory latency and bandwidth savings.

37.6 Trees

Just as they do for sorted data in one dimension, trees provide substantial speedups in two or more dimensions. In 1D, "splitting" the real line is easy: We consider all numbers greater than or less than some splitting value v. In higher dimensions, we generally use hyperplanes for dividing space, and different choices for hyperplanes lead to different data structures, some of which we now describe.

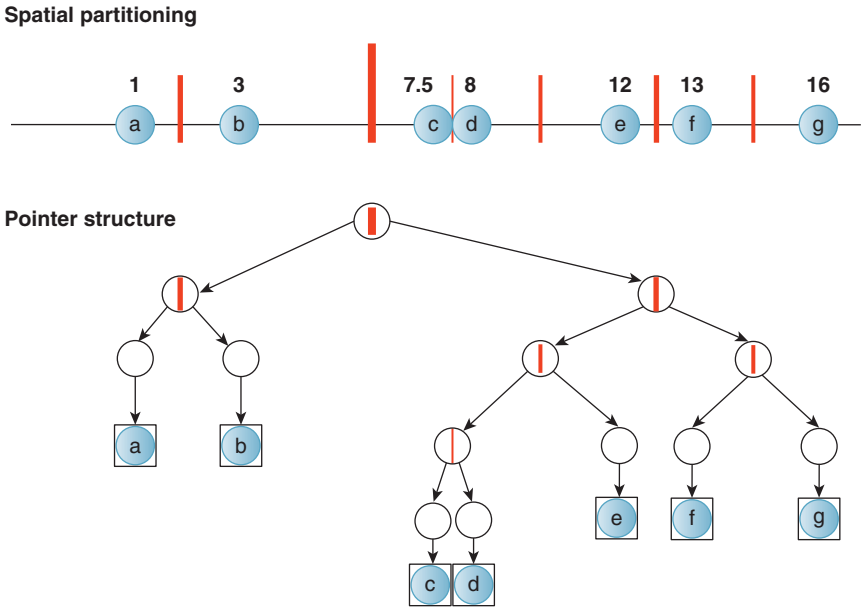


Figure 37.7: A depiction of a 1D binary tree as partitions (black lines) of values with associated keys (red disks). The thickness of the partition line represents the tree depth of that partition node in the tree; the root is the thickest.

37.6.1 Binary Space Partition (BSP) Trees

A 1D binary search **tree** recursively separates the number line with splitting points, as depicted in Figures 37.7 and 37.8. In 2D, a spatial **tree** separates 2-space with splitting lines, as depicted in Figure 37.9. In 3D, a spatial tree separates 3-space with splitting planes.

The analogy continues to higher dimensions. For any number of dimensions, a **binary space partition** (**BSP**) tree expresses a recursive binary (i.e., two-sided) partition (i.e., division) of space [SBGS69, FUCH80]. This partitioning divides space into convex subspaces, that is, convex polygons, polyhedra, or their higher-dimensional analogs, called polytopes. The leaves of the tree correspond to these subspaces, which we'll call polyhedra in general. The internal nodes correspond to the partition planes. They also represent convex spaces that are unions of their children.

BSP trees can support roughly logarithmic-time intersection queries under appropriate conditions. These intersection queries can be framed as intersecting some query geometry with either the convex polyhedra corresponding to the leaves, or the primitives inside the leaves. In the latter case, note that the tree is only accelerating the intersection computation on nodes. For primitives stored within a node, it delegates the intersection operation to a list data structure. That provides no further acceleration. But it allows the tree to have an interface that is more useful to an application programmer. The application programmer is primarily concerned with the primitives, and wants the tree's structure to provide acceleration but have no semantic impact on the query result.

In addition to the ray, box, and ball queries, BSP trees can also enumerate their elements in front-to-back overlap order relative to some reference point, and they

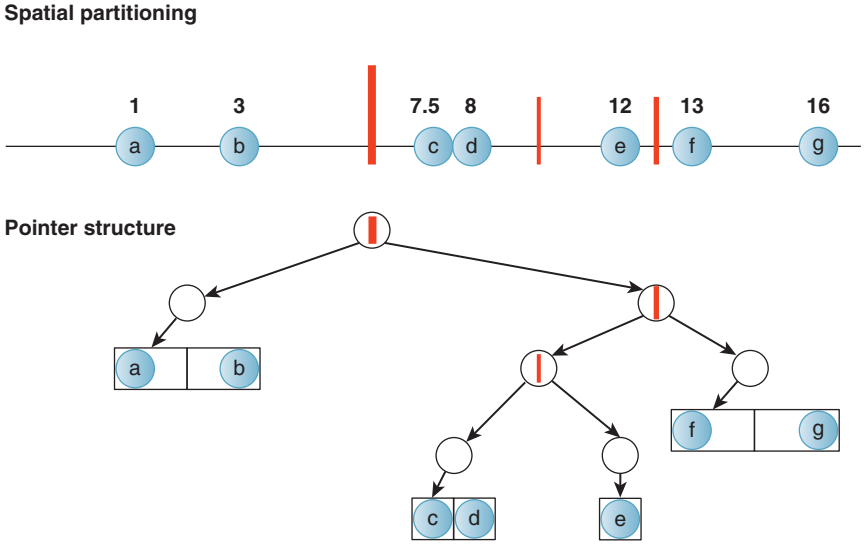


Figure 37.8: An alternative choice of tree structure for the same data shown in Figure 37.7. This tree is shallower and contains multiple primitives per node, indicated by adjacent boxes. Such a structure might be preferred if the overhead of processing a node is high.

can produce a convex polyhedron bounding the empty space around a point. Many video games use the latter operation to locate the convex space that contains the viewer for efficient access to precomputed visible-surface information.

The ordered enumerations are possible because the partitions impose an ordering of the convex spaces along a ray. This allows partially ordered enumeration of primitives encountered along a ray. It is not a total ordering because some nodes may contain more than one primitive in an unordered list. The ordering of nodes allows early termination when ray casting, which is why they have been frequently employed for accelerating ray casting as described in Section 36.2.1. It also allows hierarchical culling of occluded nodes within a camera frustum and early termination in that case, as explained in Section 36.7.

The logical (i.e., pointer) structure of the tree in memory is simply that of a binary tree (see Listing 37.12), regardless of the number of spatial dimensions. Trees are typically built over primitives, such as polygons. This leaves a design choice of how to handle primitives that span a partition. In one form, primitives that span a partition plane are cut by the plane, and the leaves store lists of the cut primitives that lie entirely within their convex spaces. To preserve the precision of the input, the cutting operation may be performed while building the tree and the original primitives stored at the leaves. Under that scheme, the same primitive may appear in multiple leaves.

An alternative is to store primitives that span a partition plane in the node for that plane. This can destroy the asymptotic efficiency of the tree if many primitives span a plane near the root of the tree. However, for some scenes this problem can be avoided by choosing the partitions to avoid splitting primitives.

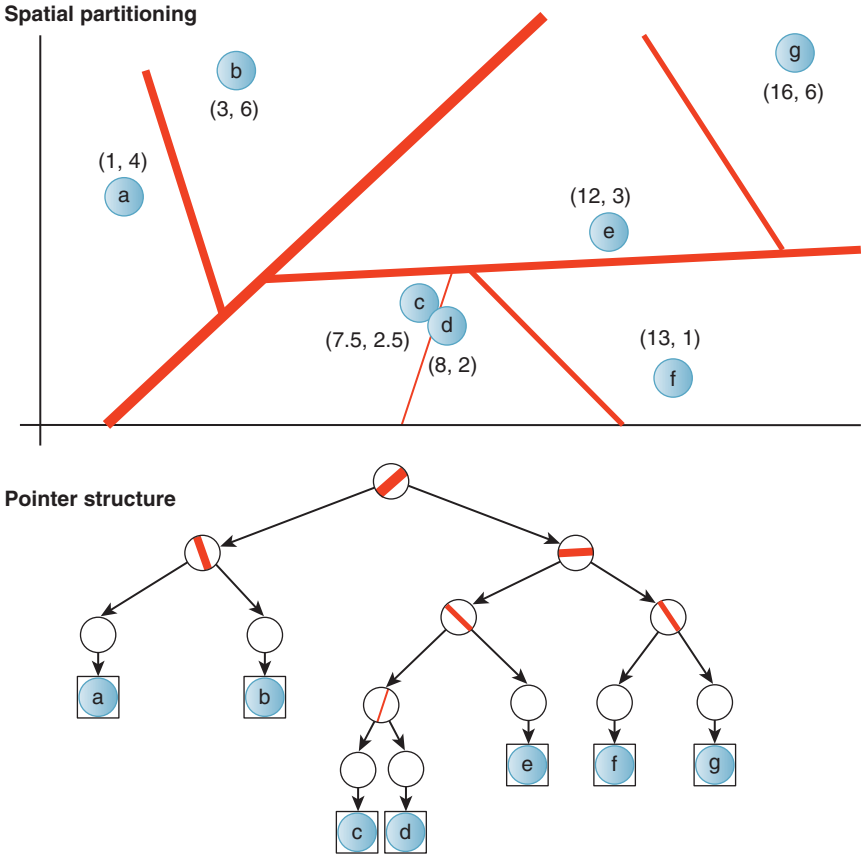


Figure 37.9: A depiction of a 2D binary space partition tree (BSP) as partitions (black lines) of values with associated keys (red disks). The thickness of the partition line represents the tree depth of that partition node in the tree; the root is the thickest.

Listing 37.12: A C++ implementation of a binary space partition tree.

```
template<class Value, class Bounds = PrimitiveKeyTrait<Value> >
class BSPTree {
  class Node {
  public:
    Plane partition;

    /* Values at this node */
    List<Value, Bounds> valueArray;

    Node* negativeHalfSpace;
    Node* positiveHalfSpace;
  };

  Node* root;

  ...
};
```

Although all leaves represent convex spaces, the ones at the spatial extremes of the tree happen to correspond to spaces with infinite volume. For example, the rightmost space on the number line in Figure 37.7 represents the interval from 14.5 to positive infinity. Infinite volume can be awkward for some computations. It is also a strength of the tree data structure because every BSP tree can represent all of space. Thus, without even changing the structure of the tree, one can dynamically add and remove primitives from nodes. This is useful for expressing arbitrary movement of primitives through a scene. For scenes dominated by static geometry, this is a significant advantage that the tree holds over other spatial data structures such as grids that represent only finite area. Those data structures must change structure when a primitive moves outside the former bounds of the scene.

A tree containing n primitives must store at least references to all of the primitives, so its size must be at least linear in n. The smallest space behavior occurs when the tree has a small number of nodes storing a large number of primitives. It also occurs when building a highly unbalanced tree at which most nodes have only one branch, as shown in Figure 37.10.

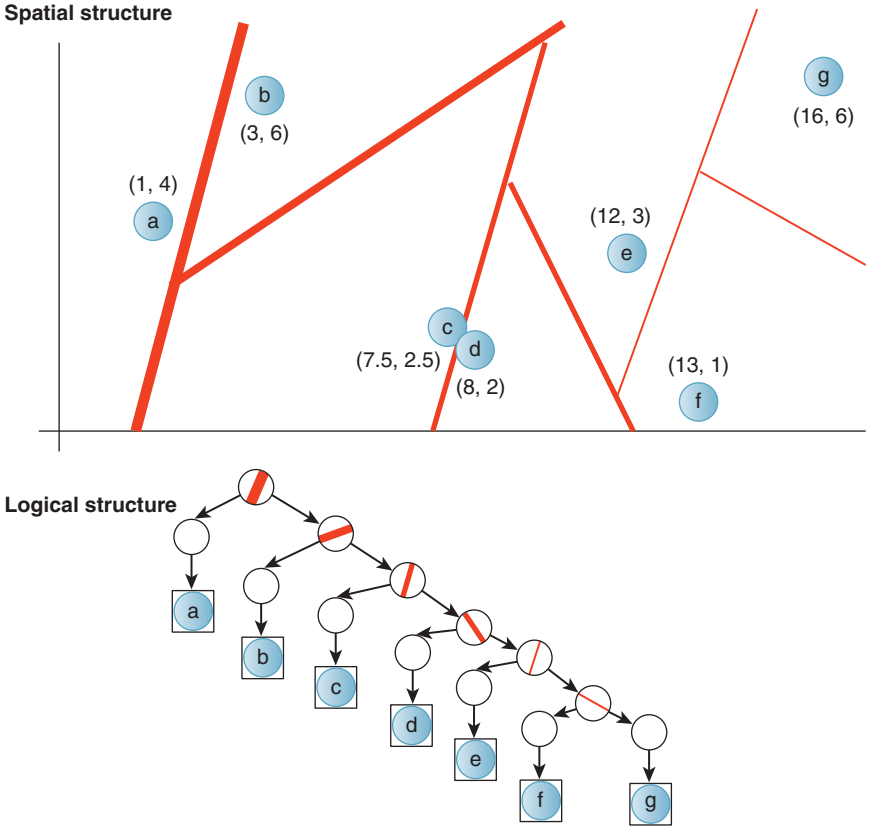


Figure 37.10: A degenerate BSP tree with the same asymptotic performance as a list. The poor performance results from an ineffective choice of partitions. This situation can arise even with a good tree-building strategy if elements move after the partitions have been placed. Even worse trees exist—there could be many empty partitions.

It is important to note that the size of the tree *has no upper bound*. There are two reasons for this, both of which can easily occur in practice. First, one can add partitions independent of the number of primitives. This may be useful if one expects additional primitives to be inserted later. This may occur naturally from an implementation that chooses not to remove nodes when primitives move out of them (e.g., for time efficiency in move and delete operations).

Second, in an implementation that splits primitives, poor choice of partition planes can significantly amplify the number of primitives stored. As discussed in Section 37.6.2, choosing partitions is a hard problem. For arbitrary scenes there is no reason to expect that the partitions can be chosen to avoid substantial amplification of primitives. Since sufficient amplification of the number of primitives can cancel any benefit from efficient structure, one must be careful when choosing the partition planes.

Most algorithms on trees assume that they have been built reasonably, with fewer than $2n$ nodes.

A *balanced* BSP tree can locate the leaf containing a point in $\mathrm{O}(\lg n)$ comparisons, which is why it is common to think of tree operations as being log-time. If the tree is unbalanced but has reasonable size, this may rise as high as $\mathrm{O}(n)$ comparisons, as in the degenerate tree shown in Figure 37.10. For a tree with an unreasonable size there is no upper bound on the time to locate a point, just as there is no bound on the size of the tree.

Our common intersection problems—first-ray intersection, conservative ball intersection, and conservative box intersection—are all at least as hard as locating the convex region containing a point. So all intersection queries on the BSP tree take *at least* logarithmic expected time for a balanced tree, yet have no upper bound on runtime if the tree structure is poor. The volume (i.e., ball and box) queries are also necessarily linear in the size of the output, which can be as large as n.

The actual runtime observed for intersection queries generally grows as the amount of space covered by the query geometry increases. Fortunately, it usually grows sublinearly. The basic idea is that if a ray travels a long distance before it strikes a primitive, it probably passed through many partitions, and if it travels a short distance, it probably passed through few. The same intuition holds for the volume of primitives.

A good intuition is that the runtime might grow logarithmically with the extent of the query geometry, since for uniformly distributed primitives we would expect partitions to form a balanced tree. However, one would have to assume a lot about the distribution of primitives and partitions to prove that as a bound.

The idea of the spatial distribution of primitives is very important to the theory supporting BSP trees, despite its confounding effects on formal analysis. One typically seeks to maintain a classic binary search tree in a balanced form to minimize the length of the longest root-to-leaf path and thus minimize the expected worst-case search behavior.

Balance (and big-O analysis) helps to minimize the worst-case runtime. But under a given pattern of queries and scene distribution, you might want to find the tree structure that minimizes the overall runtime. While your algorithms book stresses the first one, what actually matters in practice is the second one.

For example, a balanced spatial data structure tree is rarely optimal in practice [Nay93]. This at first seems surprising—classic binary tree data structures are usually designed to maintain balance and thus minimize the worst case. However,

since the average-case performance depends on the queries and we expect many queries for a graphics application, we should intentionally unbalance the tree to favor expected queries. This is directly analogous to the tree used to build a Huffman code for data compression. There, one knows the statistics of the input stream and wants to build a tree that minimizes the average node depth *weighted* by the number of occurrences of each node in the input stream. Unfortunately, we don't know the exact queries that will be made, so we can't apply Huffman's algorithm exactly to building a spatial data structure. But we can choose some reasonable heuristics because our values have a spatial interpretation. For example, we might assume that queries will be equally distributed in space or that the probability of a random query returning a primitive is proportional to the size of that primitive.

Finally, because wall-clock time is what matters for this kind of analysis, we have to factor in the tree-build time and the relative costs of memory operations and comparisons when optimizing.

37.6.2 Building BSP Trees: oct tree, quad tree, BSP tree, *k*d tree

It is hard to choose good partition planes. There are theoretically infinitely many planes to choose from. The choice depends on the types of queries, the distribution of data, and whether one wants to optimize the worst case, the best case, or the "average" case. If the tree structure will be precomputed, the build process can be made expensive—and some algorithms take $O(n^2)$ time to build a tree on n primitives. If the tree will be built or modified frequently inside an interactive program, it is important to minimize the combination of tree-build and query time, so one might choose partitions that can be identified quickly rather than ones that give optimal query behavior.

To address the problem of considering too many choices of partition planes, it is often easier to introduce constraints so as to consider a smaller number of options. This has the added advantage of requiring fewer bits to represent the partitions, which translates to reduced storage cost and reduced memory bandwidth during queries.

One such artificial constraint is to consider only axis-aligned partitions [RR78]. The resultant BSP tree is called a ***k*d *tree*** (also written as "*k*-d tree"). The axis to split along may be chosen based on the data, or simply rotated in round-robin fashion. In the latter case, each partition plane can be represented (see Listing 37.13) by a single number representing its distance from the origin, since the plane normal is determined by the depth of a plane's node in the tree.

*Listing 37.13: k*d *tree representation.*

```
template<class Value>
class KDTree {
  class Node {
  public:
    float partition;
    Node* negativeHalfSpace;
    Node* positiveHalfSpace;
    List<Value> valueArray;
  };

  Node* root;

  ...
};
```

Regardless of restrictions, there are still many choices of where to place a partition:

- Mean split on extent
- Mean split on primitives
- Median split on primitives
- Constant ray intersection probability [Hav00]
- Clustering

Splitting along the mean of the extent of all primitives at a node along each axis yields a set of nested k-cubes [RR78]. That is, each partition is at the center of its parent's polyhedron. In 3D, this is called an **oct tree** (a.k.a. "octree") (see Listing 37.14). Although they can be represented using binary trees, for efficiency oct trees are typically implemented with eight child pointers, hence their names. They have 2D analogs in **quad trees** (see Figure 37.11), and can of course be extended to 1D or 4D spaces and higher.

Listing 37.14: Oct tree representation.

```
template<class Value>
class OctTree {
 class Node {
 public:
   Node* child[8];
   List<Value> valueArray;
};

 Node* root;
 Vector3 extent;
 Point3 origin;
 ...
};
```

In general, building trees is a slow operation if implemented in a straightforward manner. It was classically considered a precomputation step. In that case, one avoided changing the structure of the tree at runtime.

However, if you are willing to complicate the tree-building process and work with a concurrent system, you can build trees fairly fast on modern machines. Lauterbach et al. [LGS+09] reported creating trees dynamically on a GPU at about half the speed of rasterization for a comparable number of polygons on that GPU. Having tree-building performance comparable to rasterization performance means that it can be performed on dynamic scenes. Furthermore, for many applications, most of the geometry in a scene does not change position between frames. In this case, only the subtrees containing dynamic elements need to be rebuilt. Fortunately, although implementing a fast tree-building system is difficult, there are libraries that implement the tree build for you.

Here are some conventional wisdom and details.

- Current hardware architectures favor very deep trees, so plan to keep subdividing down to one or two primitives per node [SSM+05].
- As discussed, we often want the tree to be *un*balanced [SSM+05].

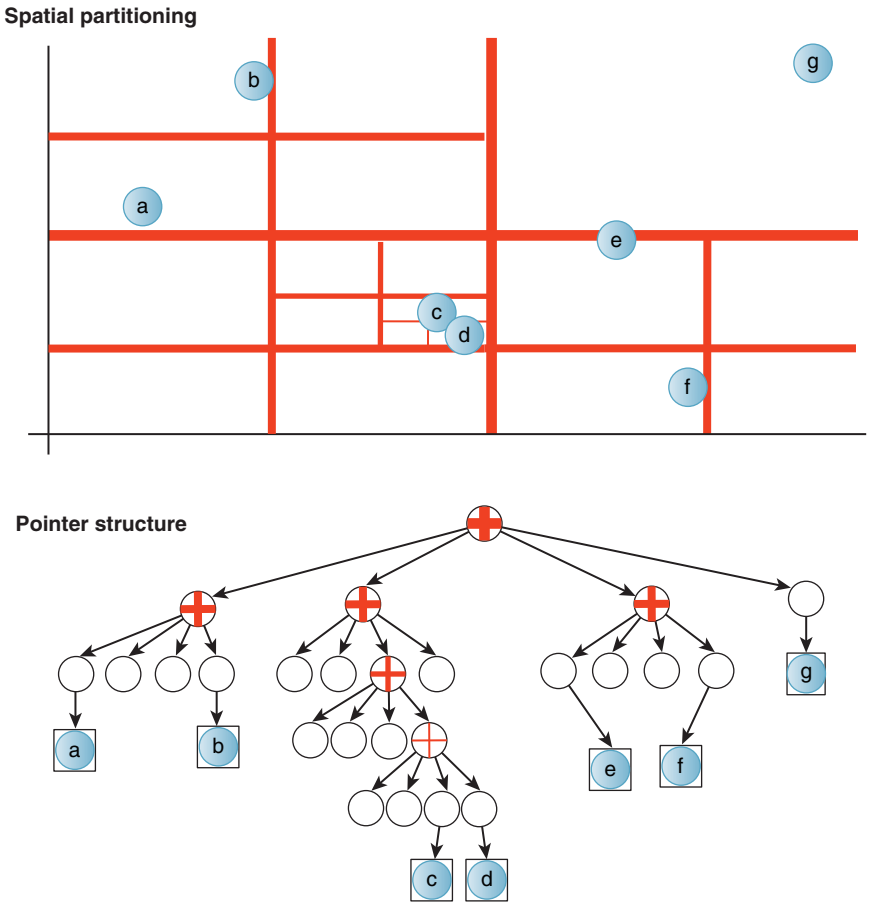


Figure 37.11: A quad tree. If a cluster of data points is not split by a low power-of-two plane position, the tree may become very deep, as near primitives c and d in this example. A better tree-building strategy for this particular data set would have been to make a second-level node containing two primitives, rather than subdividing until each node contained a single primitive. However, if c and d represented clusters of hundreds of primitives, no efficient tree would be possible—either many primitives would lie at each node, or there would be a long degenerate chain of internal nodes.

- Because of memory bandwidth and cache coherence, saving space saves runtime when constants are "small" (and we're often in that case). So compact tree representations can lead to substantial speedups. Here are some tricks for reducing the memory footprint of internal nodes.
  - Place children at adjacent locations in memory so that a single pointer can reference them.
  - Use *k*d trees instead of BSP trees to limit the footprint of each node to a single offset, or oct trees so that no per-node information is required.
  - Pack per-node values (such as the *k*d tree plane offset) directly into unused bits of the child pointer value by limiting the precision for each.

  These kinds of optimizations can lead to a 10*x* performance increase in practice [SSM+05].

- Be careful about precision when splitting elements. Due to finite precision, the new vertices introduced will rarely lie on a splitting plane, and will thus cause the split primitives to poke slightly into the sibling node.
- Be careful about less-than versus less-than-or-equal-to comparisons. Consider what happens to a point on a splitting plane and ensure that your tree-building and tree-traversal algorithms are consistent with one another. Unfortunately, a splitting plane passing exactly through a vertex or edge is not a rare situation because you often chose splitting planes based on the primitives themselves.

37.6.3 Bounding Volume Hierarchy

A **Bounding Volume Hierarchy** (**BVH**) is a spatial tree comprising recursively nested bounding volumes, such as axis-aligned boxes [Cla76]. Figure 37.12 shows a 2D axis-aligned box bounding volume hierarchy for a set of points. Listing 37.15 shows a typical representation of the tree.

Unlike a BSP tree, this type of tree provides tight bounds for clusters of primitives and has finite volume. BVHs are commonly built by constructing a BSP tree, and then recursively, from the leaves back to the root, constructing the bounding volumes. The bounding volumes of sibling nodes often overlap under this scheme; as with a BSP tree one can also create a BVH that splits primitives to maintain disjoint sibling spaces.

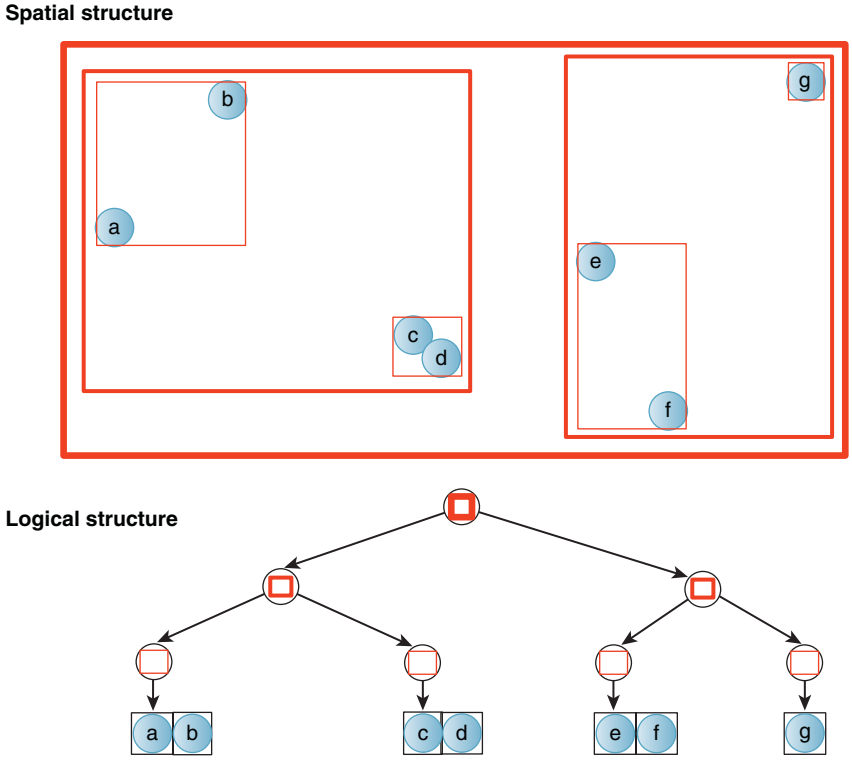


Figure 37.12: A depiction of a 2D axis-aligned box bounding volume hierarchy (BVH). This is an alternate form of tree to the binary space partition tree that provides tighter bounds but does not allow updates without modifying the tree structure.

Listing 37.15: Bounding volume hierarchy using axis-aligned boxes.

```
template< class Value, class Trait = PrimitiveKeyTrait<Value> >
class BoxBVH {
 class Node {
 public:
   std::vector<Node*> childArray;

   /* Children in this leaf node. These are pointers because a value
   may appear in two different nodes. */
   List<Value*> valueArray;

   AABox bounds;
 };

 Node* root;
 ...
};
```

The bounding volumes chosen tend to be balls or axis-aligned boxes because those admit compact storage and intersection queries. These are also known as **sphere trees** or **AABB trees,** respectively.

37.7 Grid

37.7.1 Construction

A **grid** is the generalization of a radix-based 1D array. It divides a finite rectangular region of space into equal-sized **grid cells** or **buckets** (see Figure 37.13). Any

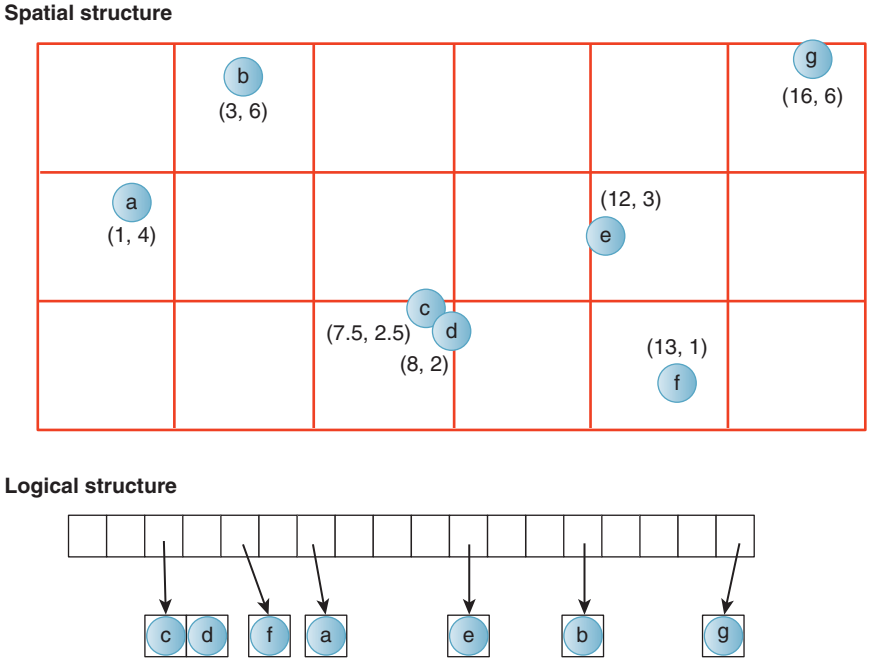


Figure 37.13: A depiction of a 2D grid of values with point keys. The logical structure is that of an implementation that unrolls the underlying 2D array in row-major order starting from the bottom row.

point P within the extent of the grid lies in exactly one cell. The multidimensional index of that cell is given by subtracting the minimal vertex of the grid from P, dividing by the extent of one cell, and then rounding down. For a grid with the minimal corner at the origin and cells of extent one distance unit in each dimension, this is simply the "floor" of the point. Thus, if we hold cell extent constant, the containing cell can be found in constant time, but the storage space for the structure will be proportional to the sum of the number of members *and* the volume of the grid. Intersection queries on a grid generally run in time proportional to the volume of the query shape (or length, for a ray).

Grids are simple and efficient to construct compared to trees, and insertion and removal incur comparatively little memory allocation or copying overhead. They are thus well suited to dynamic data and are frequently used for collision detection and other nearest-neighbor queries on dynamic objects.

Listing 37.16 shows one possible representation of a grid in three dimensions, with some auxiliary methods. The underlying representation is a multidimensional array. As for images, which are 2D arrays of pixels, grids are frequently implemented by unrolling along each axis in turn to form a 1D array. In the listing, the `cellIndex` method maps points to 3D indices and the `cell` method returns a reference to the cell given by a 3D index in constant time.

Listing 37.16: Sample implementation of a 3D x-major grid and some helper functions.

```
struct Index3 {
    int x, y, z;
};

template< class Value, class Bounds = PrimitiveKeyTrait<Value> >
class Grid3D {
  /* In length units, such as meters. */
  float cellSize;
  Index3 numCells;

  typedef List<Value, Bounds> Cell;

  /* There are numCells.x * numCells.y * numCells.z cells. */
  std::vector<Cell> cellArray;

  /* Map the 1D array to a 3D array. (x,y,z) is a 3D index. */
  Cell& cell(const Index3& i) {
    return cellArray[i.x + numCells.x * (i.y + numCells.y * z)];
  }

  Index3 cellIndex(const Point3& P) const {
    return Index3(floor(P.x / cellSize),
                  floor(P.y / cellSize),
                  floor(P.z / cellSize));
  }

  bool inBounds(const Index3& i) const {
    return
       (i.x >= 0) && (i.x < numCells.x) &&
       (i.y >= 0) && (i.y < numCells.y) &&
       (i.z >= 0) && (i.z < numCells.z);
  }

  ...
};
```

Since the number of operations involved in finding a grid cell is relatively small, each operation takes a large percentage of the cell dereference time and it is worth micro-optimizing them from the straightforward version shown here. If `cellSize` is `1.0`, the divisions in `cellIndex` can be completely eliminated. Where that is not possible, multiplying P by a precomputed `1/cellSize` value is faster on many architectures than performing the division operations. The floor operation is a relatively expensive floating-point operation that can be replaced with an intrinsic-supported truncate-and-cast-to-integer operation. When `numCells.x` and `numCells.y` are powers of two, the multiplications in the `cell` method become faster bit-shift operations. Finally, at the loss of some abstraction, it is sometimes worth exposing the 1D indexing scheme outside of the data structure. Callers can then iterate directly through the 1D array in steps chosen to walk along any given dimension.

By convention, elements in the 1D array varying along the lowest-indexed dimension (e.g., x) are the ones placed at adjacent memory addresses. When the iteration order is expected to favor some other axis, arranging to unroll along that axis first can yield increased cache coherence. For example, if an application is expected to trace many rays vertically, then making the vertical axis coherent is a good choice.

Sometimes the dominant ray iteration direction cannot be predicted, or iteration will be across multiple dimensions for volume queries such as ball intersection. If memory coherence is a concern for such applications, unrolling the multidimensional array along a space-filling curve can be a good solution. Curves such as the Hilbert [Hil91, Voo91] or Morton [Mor66] (a.k.a. "Z," for its shape) curves define an indexing scheme that, on average, assign spatially local elements to spatially local memory addresses. These can typically be implemented with a few bitwise operations per index computation.

Tuning the extent of each cell, or equivalently, the number of cells for a given grid extent, requires a good understanding of the kinds of queries that will be performed and the spatial distribution of the data. This issue is best discussed after the query algorithms, so we set it aside until Section 37.7.3.

The `inBounds` method is convenient for identifying legal indices, which is necessary because the grid has finite extent. For some applications data sets have inherent spatial bounds. For example, in a virtual environment like a video game level with a well-defined boundary, objects will never move beyond the boundary. In this case the finite nature of the grid presents no limitation. In other cases, a scene is known to be very dense in a specific region but contains a few elements that lie far from that region, possibly without any practical spatial bound. In this situation one can extend the grid with a single additional list of elements that lie outside the grid proper. This allows the data structure to represent unbounded extent yet still provide efficient queries within the dense region. If a data set is neither spatially bounded nor primarily clustered in a dense region, the simple grid is likely a poor choice of data structure. However, the sparse **hash grid** described later in this section may still be appropriate.

37.7.2 Ray Intersection

To find the first intersection of a ray with the primitives in a grid, the query algorithm must walk through the cells of the grid in the order in which the ray enters them, as shown in Figure 37.14. In 2D, this is equivalent to the problem of

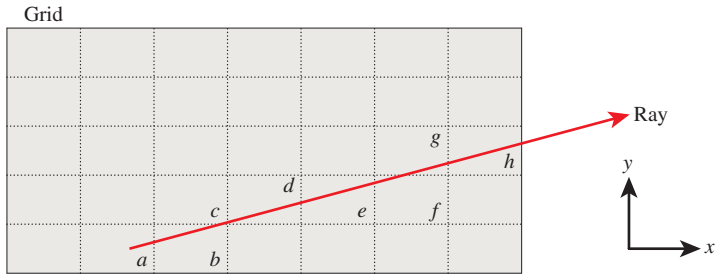


Figure 37.14: Ray traversal of a 2D grid (from [AW87]). To correctly traverse the grid, a traversal algorithm must visit cells a, b, c, d, e, f, g, and h in that order. (Reprinted from Eurographics 1987: Conference Proceedings, *1987 (ISBN 0444702911), Marechal ed., John Amanatides and Andrew Woo, pp. 1–10, "A Fast Voxel Traversal Algorithm for Ray Tracing," Figure 1.)*

(ordered) **conservative rasterization** of the line: Find all cells that touch any part of the line. In 3D, the process is called **conservative voxelization.**

Listings 37.17–37.19 give an algorithm for conservative voxelization first proposed by Amanatides and Woo [AW87]. During initialization (Listing 37.17), this algorithm finds the grid cell containing the ray origin. It then computes several vector quantities describing the relative rate of motion of a point $Q(t) = P + \vec{d}t$ on the ray with origin P and direction $\vec{d}$. From these it can iteratively march through the grid by stepping to the nearest grid line (Listing 37.19).

Listing 37.17: Interface for a 3D grid iterator.

```
1  class RayGridIterator {
2  public:
3     /* Current grid cell */
4     Index3 index;
5
6     /* Sign of the direction that the ray moves along each axis; +/-1 or 0 */
7     Index3 step;
8
9     /* Size of one cell in units of t along each axis. */
10    Vector3 tDelta;
11
12    /* Distance along the ray of the first intersection with the
13       current cell (i.e., that given by index). Initially zero. */
14    float tEnter;
15
16    /* Distance along the ray to the intersection with the next grid
17       cell. tEnter and tExit can be used to bracket ray ray-primitive
18       intersection tests within a cell. */
19    Vector3 tExit;
20
21    RayGridIterator(const Ray& ray, float cellSize);
22
23    /* Increment the iterator, stepping exactly one cell along exactly one axis */
24    RayGridIterator& operator++();
25 };
```

Listing 37.18: Initialization of a ray-grid iterator.

```
RayGridIterator::RayGridIterator(const Ray& ray, float cellSize) {
   tEnter = 0.0f;

   // Iterate over axes, treating points and vectors as linear algebra vectors
   for (int a = 0; a < 3; ++a) {
      index[a] = floor(ray.origin()[a] / cellSize);
      tDelta[a] = cellSize / ray.direction()[a];

      step[a] = sign(ray.direction()[a]);

      float d = ray.origin()[a] - index[a] * cellSize;
      if (step[a] > 0)
         // Measure from the other edge
         d = cellSize - d;

      if (ray.direction()[a] != 0)
         tExit[a] = d / ray.direction[a];
      else
         // Ray is parallel to this partition axis.
         // Avoid dividing by zero, which could be NaN if d == 0
         tExit[a] = INFINITY;
   }
}
```

Listing 37.19: Amanatides and Woo's algorithm for iterating over the 3D grid cells along a ray, in order.

```
RayGridIterator& operator++() {
  tEnter = tExit;

  // Find the axis of the closest partition along the ray
  int axis = 0;
  if (tExit.x < tExit.y)
    if (tExit.x < tExit.z)
       axis = 0;
    else
       axis = 2;
  else if (tExit.y < tExit.z)
    axis = 1;
  else
    axis = 2;

  index[axis] += step[axis];
  tExit[axis] += tDelta[axis];

  return *this;
}
```

A limitation of this algorithm is that all of the t-based quantities are stored in floating point and incremented by addition. They will thus accumulate the round-off error. If implemented in fixed-point arithmetic, the error in the intersection positions never increases, but the error in the approximation of the ray direction increases.

For points reasonably near the origin (e.g., with floating-point values less than 10^4) this algorithm is generally considered sufficiently robust for rendering. However, it may not be sufficiently robust for dynamics simulation, where a missed collision can result in objects becoming stuck or falling through the world.

Listing 37.20 gives the actual intersection algorithm for the grid, considering both primitives and cells. It iterates through all cells that are in bounds and along the ray. For each cell encountered, it uses the underlying list's intersection method to find the first (if any) intersection.

Listing 37.20: Ray intersection using a 3D grid and iterator.

```
class Grid3D {
  ...

  /* Assumes that the ray begins within the grid */
  bool firstRayIntersection(const Ray& ray, Value*& value, float& distance) const {

    for (RayGridIterator it(ray, cellSize); inBounds(it.index); ++it) {
        // Search for an intersection within this grid cell
        const Cell& c = cell(it.index);
        float maxdistance = min(distance, t.tExit);
        if (c.firstRayIntersection(ray, value, maxdistance)) {
              distance = maxdistance;
              return true;
        }
    }

    // Left the grid without ever finding an intersection
    return false;
  }
};
```

Because primitives may appear in more than one grid cell, it is essential to test only for intersections that occur before the end of the cell at each iteration. An example of a case that depends on this is shown in Figure 37.15. In that figure, consider the intersection test that occurs when the iterator is at the cell labeled *b*. Because the cells covered by object *Y* include cell *b*, during this iteration *Y* will be tested against the ray. There is in fact an intersection—yet it occurs in cell *c*, not cell *b*. Were the algorithm to return that intersection, it would miss the true first intersection, which occurs with object *X* in cell *c*.

Intuition indicates that the algorithm will run in time proportional to the number of grid cells traversed, because that is the cost of the iterator. The constant overhead of each iteration is very low—a handful of floating-point operations—so the algorithm is practical for cases where we don't expect the ray to travel too far before striking something. A grid with g subdivisions along k dimensions

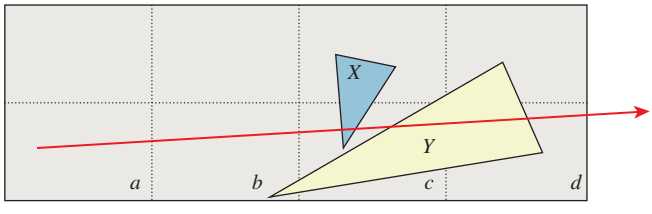


Figure 37.15: A case where it is essential only to test for intersections that lie within each cell during the ray-marching process of ray-triangle intersection accelerated by a spatial grid. (Redrawn from Eurographics 1987: Conference Proceedings, *1987 [ISBN 0444702911], Marechal ed., John Amanatides and Andrew Woo, pp. 1–10, "A Fast Voxel Traversal Algorithm for Ray Tracing," Figure 1.)*

contains $O(g^k)$ cells. The longest ray traversal is the $O(g)$-length diagonal of the grid. For a grid containing n primitives, in the worst case all of those primitives cover every grid cell along the diagonal but the ray intersects none of them. The intersection query cost is thus $O(g \cdot n)$.

Of course, for such a scene the grid is a poor choice of data structure, since even a spatial list beats its performance. So the worst-case bound is very different from the expected behavior. The grid is better suited to a scene containing roughly uniformly distributed primitives that tend to fit within a grid cell. Likewise, it is better suited to cases of the anticipated ray queries that will progress a small fraction of the scene before striking a primitive.

Trees will exhibit better asymptotic performance under queries than will grids for large scenes with uneven spatial distributions—we expect to see some kind of logarithmic versus linear behavior comparing these data structures in the long run. However, the iteration through *empty* cells in a grid is fairly fast. Thus, the constant applied to the linear (in the length of the array) time cost term is small and one can afford to traverse many cells for each ray. For scenes of bounded size, the grid may outperform the tree on ray-intersection queries, especially if the time to build the data structure is factored into the net runtime.

37.7.3 Selecting Grid Resolution

If we expect to perform many box or ball intersections, then we should size the grid based on the expected intersection object size so that the intersection algorithm will typically have to examine only a small number of cells (maybe one to four).

Ray intersection on a grid takes time linear in the length of the ray because the number of grid cells to examine is asymptotically proportional to the length of the ray. It also takes time linear in the number of primitives in the grid cells that are encountered. This creates a tension between minimizing the number of grid-intersection tests and the number of element-intersection tests. We assume that grid-intersection tests are less expensive than element intersections. The cost ratio for each kind of test is probably constant, but it may vary over a large range—say, 3:1 for ray-sphere : ray-grid time and 200:1 for ray-implicit-surface : ray-grid time. We might have millions of each kind of potential intersection with significantly varying probability of occurrence. Moreover, changing that ratio affects the space cost of the data structure, which impacts both viability and memory performance. So it is not obvious which kind of intersection test the data structure should favor for performance. The answer is that it depends on the structure of the scene as well as the cost of the intersection tests.

We do know that if we make g large, the grid squares are small, as depicted in Figure 37.16 (left). This reduces the number of elements to be tested in each nonempty cell, but it increases the number of cells that we need to examine. That is good for dense scenes. The figure depicts this by highlighting the primitives that are tested against the ray. That is a small fraction of the elements in the grid in the figure, but it is a large number of grid borders compared to the number of elements in the grid.

If we make g small, the grid squares are large. This allows the ray to quickly step through large volumes of empty space, but it increases the number of elements that must be tested in each nonempty cell encountered. That is good for sparse scenes. Figure 37.16 (right) shows a coarse grid over the same scene

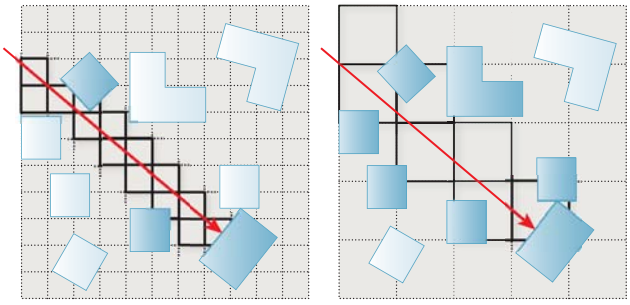


Figure 37.16: (Left) Large g gives small grid cells. A ray iterator must touch many cells (shown with solid black borders), but most of those will be empty. This is good if the cost of testing for a geometric intersection is large compared to the cost of iterating through (or storing) cells. (Right) Small g gives large grid cells. A ray iterates through few grid cells, but those contain many elements: The conservative ray-cell test produces many false positives. This is good if the cost of testing for a geometric intersection is small compared to that of iterating through cells.

from Figure 37.16 (left). For the coarse grid, there are relatively few ray-grid intersections, but many more ray-element intersection tests must be performed within each grid cell.

We describe scenes that contain a mixture of dense and sparse areas as having a **nonuniform spatial distribution** of elements. A grid is likely a poor data structure for ray intersection in a scene of nonuniform density because no single g constant can serve both areas well. A tree is a better choice in that case because it can adapt to varying spatial density.

There are three layers of subtlety that can subvert this conclusion. At first, it seems that a grid will rarely be a useful data structure. That is because many graphics scenes represent only surface geometry. This makes their spatial distribution of geometry inherently nonuniform, in the sense that the primitives cluster at the surfaces of objects and not on their interior.

The second point is that this characterization itself depends on scale, so it is not a safe generalization. The Manhattan area of New York City comprises tall buildings on a mostly regular grid. Figure 37.17 shows an idealized top view of such a layout. If we examined a computer graphics model of the building exteriors over a volume about the size of a single room (e.g., choosing g so that

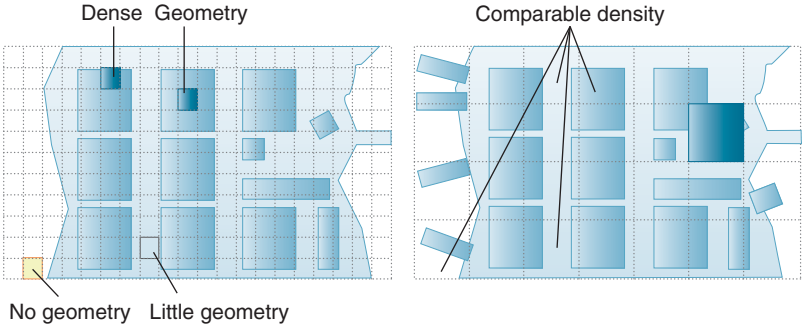


Figure 37.17: Hypothetical top view of a city on an island. (Left) Small grid cells give inhomogeneous geometric density. (Right) Sufficiently large cells produce approximately uniform density for this scene.

$g^3/n \approx 4 \times 4 \times 2.5\ \text{m}^3$), we'd find that the geometry is indeed distributed in a nonuniform fashion. That's because a room volume overlapping the edge of a building would contain geometry, but one on the interior or completely outside the building would be empty.

If we instead chose grid cells to have at least the footprint of a city block, we'd find that every cell contains about the same number of building exteriors, so the density is nearly uniform. (Of course, some buildings might have more detail in the model than others—let's just work with a consistent tessellation level.) So part of the challenge in choosing the grid size is that the density, which affects our choice, depends on the grid size itself.

The third point is that constant factors can cause orders of magnitude in performance difference in ray-intersection times with a tree and a grid's own subdivisions (separate from the cost of intersecting the elements within them). That is because the regular structure of the grid allows efficient memory packing and eliminates the need to explicitly store the grid's geometry. Where a bounding volume hierarchy must explicitly encode each bounding box, the geometry of each grid cell is implicit in the grid size and requires no memory access to obtain. The process of tracing a ray through a grid is equivalent to the problem of conservatively rasterizing a line. Line rasterization algorithms tend to simplify down to one or two branches and a few additions per pixel (grid cell) on the line's (ray's) path. If the ray-grid cell intersection is $50x$ faster than the ray-tree-node intersection, we can afford to look at many empty grid cells before the inefficiency of doing so overwhelms the performance advantage.

37.8 Discussion and Further Reading

We've covered a lot of ground in this chapter at a bunch of different levels of detail and abstraction. Now let's step back and review the big picture. Relatively straightforward application of basic computer science principles to spatial data structures for representing a scene can return huge speedups for rendering and collision detection. Of course, there are a lot of caveats about avoiding degenerate cases and optimizing for peak performance. But you'd be crazy not to use these kinds of structures, because no constant speedup could possibly make a program scale to large scenes as well as even the simplest spatial tree.

Common sense, backed by examples of some specific cases, led us to the following big observations.

1. Different data structures work well in different places, and the choice isn't always determined by their asymptotic behavior. Sometimes testing in practice is the best solution.
2. Even when you've found the best data structure, you may need to do some tuning.
3. Hardware tricks can sometimes buy you another factor of ten or more.

These items are shown in order: You should choose the right data structure before you try to do hardware-based optimizations; you should test on sample scenes before you do your data-structure tuning.

Spatial data structures continue to be a hot topic in computer graphics. You should experiment by nesting (e.g., `Grid3D< BSPTree<Triangle> >`) and combine elements of different structures. It is quite common to outperform results from the literature that use more generic structures if your particular application

has not been well researched; that is a large part of how the field moves forward. In any given year there will be several new papers on strategies for placing the partitions in a spatial tree or clustering for BVHs. For example, in 1987, one good idea was to build a ray-primitive intersection structure that took the ray's direction into account [AK87]; in 2000, Havran's work on modified surface area heuristics [Hav00] influenced the way that engineers optimized tree builds based on area; in 2010 Pantaleone and Luebke [PL10] targeted massively parallel real-time tree building and tuned for tree build time rather than absolute query time.

It is also common to store the same scene data in more than one structure. For example, you may want a list of characters in the world for efficient iteration during Artificial Intelligence simulation, a hash grid for collision detection, and a BSP tree for rendering them. Often a single complex data structure will contain multiple, simpler data structures. This presents a unified interface but allows it to combine the efficiency for different methods from different single data structures. This approach gains performance at the cost of storage space and implementation complexity. In particular, storing the same data multiple times creates potential for the duplicate state becoming desynchronized, and that must be weighed carefully against the performance advantages.

There are commonly employed data structures and increasingly exotic ones. For example, most nongraphics programs rely on a small number of classic data structures such as arrays, hash tables, and an occasional tree. Most graphics programs rely on those plus the spatial data structures presented in this chapter (with the list and BVH perhaps currently the most popular). There's a lot of practical wisdom in primarily relying on such a small set of workhorses. It amortizes the cost of polishing and optimizing those structures, and lets the programmer using them focus on his or her application instead of learning about a new interface.

But sometimes a more exotic classic data structure is really needed. For example, if your entire program's performance depends on having a fast priority queue for a large data set, then it may be time to read a paper about left-leaning red-black trees [Sed] and implementing one. But if your program depends heavily on ray-heightfield intersection performance, then it may be worth investing in a data structure optimized for *that* case, such as Musgrave's and Amanatides's and Woo's grid-tracing algorithms [MKM89, AW87]. Ray-intersection data structures are a constant topic in the global illumination literature. The ray-tracing state of the art "STAR" reports are a good place to find surveys of current thought (e.g., [WMG+09]). Sphere, box and other primitive intersections often arise in the physical simulation community. The design and analysis issues presented in this chapter apply equally well to the structures already discussed and to new data structures that you will invent on your own.

We've used ray-triangle intersection as a motivating example throughout this chapter, in part because ray casting is a common operation in most renderers, and triangle meshes are a preferred representation for most scene geometries. But in the future (and in some contexts even today), both of these may change. It may be, for instance, that tracing frusta (essentially bundles of rays) becomes essential, or that geometry is represented primarily by points, or by spline surfaces, or by some yet-unimagined new kind of primitive. When that happens, the choices of acceleration structures will change as well, but the kinds of analysis we've described in this chapter will persist: Tradeoffs involving memory coherence, patterns of access, questions of whether our typical problem is of a size large enough that asymptotic analysis is informative, and the costs of implementation versus use will still guide the choices you make as you compare new structures.

Chapter 38

Modern Graphics Hardware

> In computer design it is a truism that smaller means faster.
>
> Richard Russell

38.1 Introduction

All modern personal computers include special-purpose hardware to accelerate rasterized 2D and 3D rendering. With the exception of the interface that connects the PC to the display, this hardware is, strictly speaking, unnecessary, because the rendering could be done on the PC's general-purpose processor. In this chapter we consider why PCs include special-purpose rendering hardware, how that hardware is organized, how it is exposed to you, the graphics programmer, and how it efficiently accelerates 3D rendering and other algorithms.

The history of computing includes many examples of failed special-purpose computing hardware. Examples include language accelerators, such as Lisp machines [Moo85] and Java interpreters [O'G10], numeric accelerators, and even graphics accelerators, such as the Voxel Flinger [ST91]. Modern **graphics processing units,** whose complexity and transistor counts can exceed those of the general-purpose CPU, are a prominent exception to this rule. Four conditions have allowed special-purpose graphics processing units, which we will refer to as **GPUs,** to become and remain successful. These are performance differentiation, workload sufficiency, strong market demand, and the inertia of ubiquity.

- **Performance differentiation:** Special-purpose hardware *can* execute graphics algorithms far more quickly than a general-purpose CPU. The most important reason is parallelization: GPUs employ tens or hundreds [Ake93] of separate processors, all working simultaneously, to execute graphics algorithms. While it is difficult to parallelize general computations, graphics algorithms are easily partitioned into separate

tasks, each of which can be executed independently. They are thus more amenable to parallel implementation.[1] Historically, graphics parallelization employed separate processors for the different stages in the graphics pipeline described in Chapters 1 and 14. In addition to pipeline parallelism, contemporary GPUs also employ multiple processors at each pipeline stage. The trend is toward increased per-stage parallelism in pipelines of fewer sequential stages.

Parallelism and other factors that contribute to GPU performance differentiation are explored further in Section 38.4.

- **Workload sufficiency:** Graphics workloads are huge. Interactive applications such as games render scenes comprising millions of triangles producing images, each comprising one or two million pixels, and do so 60 times per second or more. Current interactive rendering quality requires thousands of floating-point operations per pixel, and even this demand continues to increase. Thus, workloads of contemporary interactive graphics applications cannot be sustained by a single general-purpose CPU, or even by a small cluster of such processors.
- **Strong market demand:** Technical computing applications, in fields such as engineering and medicine, sustained a moderate market in special-purpose graphics accelerators during the 1980s and early 1990s. During this period companies such as Apollo Computer and Silicon Graphics developed much of the technological foundation that modern GPUs employ [Ake93]. But GPUs became a standard component of PC architecture only when demand for the computer games they enabled increased dramatically in the late ’90s.
- **Ubiquity:** Just as in the software and networking businesses, there is an inertia that inhibits utilization of uncommon, special-purpose add-ins, and that encourages and rewards their utilization when they have become ubiquitous. Strong market demand was a necessary condition for personal-computer GPUs to achieve ubiquity, but ubiquity also required that GPUs be interchangeable from an infrastructure standpoint. Two graphics interfaces—OpenGL [SA04] and Direct3D [Bly06]—have achieved industry-wide acceptance. Using these interfaces application programmers can treat GPUs from different manufacturers, and with different development dates, as being equivalent in all respects except performance. The abstraction that Direct3D defines is considered further in Section 38.3.

GPUs based on the classic graphics pipeline, exemplified by the NVIDIA GeForce 9800 GTX considered in the next section, have achieved a stable position in the architecture of personal computers. They are therefore well worth studying and understanding. The situation may change, however, as it has for so many other special-purpose functional units.

Parallelism, in the form of multiple processors (a.k.a. **cores**) on a single CPU, has replaced increasing clock frequency as the engine of exponential CPU performance improvement. The number of cores on a CPU is expected to increase exponentially, just as clock frequency has in the past. The resultant parallelism may

1. Indeed, graphics is sometimes referred to as embarrassingly parallel, because it offers so many opportunities for parallelism.

make general-purpose CPUs more performance-competitive with GPUs, thereby reducing the demand for special-purpose hardware.

Alternatively, the limitations of the classic-pipeline architecture discussed in this chapter may allow a contending architecture to overcome and replace current GPUs. Because GPUs have tended toward brute force algorithms, such as the z-buffer visible-surface algorithm described in Chapter 36, the workload argument is somewhat circular: GPUs amplify their own work requirements by implementing inefficient algorithms. Accelerated ray tracing is one candidate architecture to replace current GPUs.

Perhaps the most likely outcome is that GPU architecture will steadily evolve, as it has in the past decade, to accommodate new algorithms and architecture features.

The remainder of this chapter describes graphics architectures in part by using three exemplars: the OpenGL and Direct3D software architectures, and the NVIDIA GeForce 9800 GTX GPU, released in April 2008. While this GPU is already fading in popularity, the decisions that influenced its design will remain current for a long while.

This chapter defines many terms that you've already encountered earlier in the book. It does so for two reasons: First, the reader with some prior experience in graphics may profitably read this chapter *first,* even though it appears at the end of the book; second, some definitions in this chapter have been chosen to correspond to industry practice (so that, for instance, a "color" is an RGB triple rather than a percept in the human brain!), and we want to be certain that you understand exactly what's being said.

38.2 NVIDIA GeForce 9800 GTX

Figure 38.1 illustrates the key components and interconnections in a state-of-the-art personal computer (PC) system, as of early 2009. The Intel Core 2 Extreme QX9770 CPU, Intel X48 Express chipset, and NVIDIA GeForce 9800 GTX GPU were among the highest-performance components then available for desktop computing. This is the sort of system a serious computer gamer would have bought if the $5,000 price were within reach. With this context in mind, let's look at the performance and design of the 9800 GTX GPU.

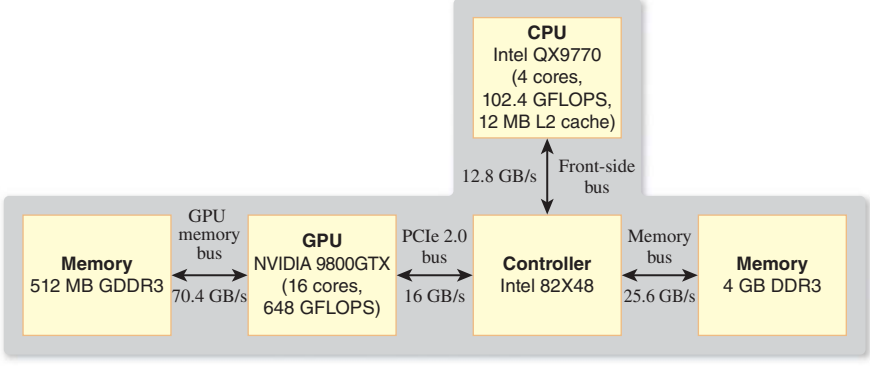


Figure 38.1: PC block diagram.

Figure 38.2: NVIDIA *GeForce 9800 GTX graphics card. (Courtesy of NVIDIA.)*

Like the Extreme CPU, the GTX GPU is a single component—a packaged silicon **chip**—that is mounted to a circuit board by hundreds of soldered circuit interconnections. The CPU, chipset, and CPU memory are mounted to the primary system circuit board, called the **motherboard,** along with most of the PC's remaining electronics. The GPU and GPU memory are mounted to a separate PCIe circuit board that is connected to the motherboard through a multiconnector socket (see Figure 38.2).

Perhaps the most notable characteristic of the GeForce 9800 GTX is its extremely high performance. The GTX can render 340 million small triangles per second and, when rendering much larger triangles, a peak of almost 11 billion pixels per second. Executing application-specified code to compute the shaded colors of rendered pixels, the GTX can perform 576 billion floating-point operations per second (GFLOPS). This is more GFLOPS than the most powerful supercomputer available just over a decade ago, and is more than five times the 102.4 GFLOPS that the Intel Core 2 Extreme QX9770 CPU can sustain. Memory bandwidth—the rate data is moved between a processor chip and its external random access memory—is an equally important metric of performance. The Core 2 Extreme QX9770 CPU accesses external memory through the X48 Express chipset. While the chipset supports memory transfers at up to 25.6 GB/s, the Front Side Bus that connects the CPU to the chipset limits CPU-memory transfers to a maximum of 12.8 GB/s. The GPU accesses its memory directly and achieves a peak transfer rate of 70.4 GB/s, more than five times that available to the CPU.

In the same way that interest compounds on invested money, CPU and GPU performance have reached their current state through steady exponential increase. Underlying this exponential increase is Moore's Law, Gordon Moore's 1965 prediction that the number of transistors on an economically optimum integrated circuit would continue to increase exponentially, as it had since Intel had begun producing integrated circuits a few years earlier [Moo65]. Over the succeeding four decades the actual increase held steady at about 50% per year, resulting in the near-iconic status of Moore's prediction. Compounding at high rates quickly results in huge gains. A decade of 50% annual compounding, for example, yields an increase of $1.5^{10} = 57.67$ times. And a decade of 100% annual compounding yields an increase of $2^{10} = 1{,}024$ times, so quantities with the same initial value that grow at these two rates differ by a factor of almost 20 after a decade. Driven by Moore's Law, the storage capacity of integrated circuit memory, which is proportional to transistor count, has increased by a factor greater than ten million since its first commercial availability in the early 1960s.

Increased transistor count allows increased circuit complexity, which engineers parlay into increased performance through techniques such as **parallelism** (performing multiple operations simultaneously) and **caching** (keeping frequently used data elements in small, high-speed memory near computational units). Increases in transistor count are driven primarily by the steady reduction in the dimensions of transistors and interconnections on silicon integrated circuits. (The interconnections on the Intel Core 2 Extreme QX9770 CPU have a **drawn width** of just 45 nm, $1/2{,}000$th the width of a human hair, and about one-tenth the wavelength of blue light.) Smaller transistors change state more quickly, and shorter interconnections introduce less delay, so circuits can be run at higher speeds. Increases in transistor count and circuit speed compound, allowing the

performance of well-engineered components to increase at rates approaching 100% per year.

GPU designers have made excellent use of increases in both transistor count and circuit speed. Over the ten years between 2001 and 2010 performance metrics of NVIDIA GPUs, such as triangles drawn per second and pixels drawn per second, have increased by 70% to 90% per year. Memory bandwidth has increased by only 50% per year. But data compression techniques, themselves enabled by increased circuit complexity, have allowed an increase in *effective* memory bandwidth of 80% per year, supporting the correspondingly large increases in drawing performance.

CPU designers have taken full advantage of increased circuit speed, but they have been less successful converting increased transistor count into performance. CPU performance has historically increased by 50% per year, an amazing achievement, but still significantly lower than the 70% to 90% annual increases in GPU performance. Today's relative performance advantage of GPUs over CPUs is the direct result of compounding performance at unequal rates.

Shortly after the year 2000, CPUs reached a power dissipation somewhat over 100 watts, near the maximum that can be dissipated by a single component in a personal computer cabinet. Because circuit power is directly proportional to circuit speed, the annual 20% increase in CPU clock speed, which had been a major driver of CPU performance increases for two decades, dropped suddenly to essentially zero. This event has motivated CPU designers to incorporate more parallelism into their circuits, an approach that has been very successful for GPUs. The Intel Core 2 Extreme QX9770 CPU is a quad-core design, meaning that it contains four microprocessor cores in a single component package. Dual-core Intel CPUs were introduced in 2005, and quad-core designs are now available. Each core has four floating-point Arithmetic and Logic Units (ALUs), each ALU including an addition unit and a multiplication unit, for a total of 32 floating-point units in the Core 2 Extreme QX9770. By comparison, the NVIDIA GeForce 9800 GTX GPU has 16 cores, each with eight floating-point ALUs, and each ALU with two multiplication units and one addition unit, for a total of 384 floating-point units. While the CPU cores are clocked at roughly twice the rate of the GPU cores (3.2 GHz to 1.5 Ghz), the sheer number of floating-point units in the GPU gives it a greater than five-to-one GFLOPS advantage (576 to 102).

In summary, as of 2009 GPUs sustained significantly higher performance than CPUs because they were doing more calculations in parallel, and they did this by devoting a greater percentage of their silicon area to computation than do CPUs. As of 2013, there's no sign of this trend slowing down. GPU parallelism is considered further in the following sections.

38.3 Architecture and Implementation

When you write code that uses a GPU for graphics, you do not directly manipulate the hardware circuits of the GPU. Instead, you code to an abstraction, which is implemented by a combination of the GPU hardware, GPU firmware (code that runs on the GPU), and a device driver (code that runs on the CPU). The firmware and the device driver are implemented and maintained by the GPU's manufacturer, which is NVIDIA in the case of the GeForce 9800 GTX. They are as fundamental to the implementation of the abstraction as the GPU hardware itself.

The distinction between an abstraction and its implementation, and the value conferred by this distinction, are familiar concepts in software development. Information hiding was introduced by Parnas in 1972 [Par72]. Today C++ developers isolate implementation details behind *interfaces* constructed using abstract classes. Prior to the development of C++, C programmers were trained to use functions, header files, and isolated code files to achieve many of the benefits that C++ classes came to provide.

The importance of separating interface and implementation was identified even earlier by computer hardware developers. Gerrit Blaauw and Fred Brooks, who with Gene Amdahl and others brought computing into the modern age with the creation of the IBM System/360 in 1964 [ABB64], define the **architecture** of a system as "the system's functional appearance to its immediate user, its conceptual structure and functional behavior as seen by one who programs in machine language" [BJ97]. They use the terms **implementation** and **realization** to refer to the logical organization and physical embodiment of a system. Careful distinction between architecture and implementation[2] allowed the System/360 to be implemented as a family of computers, with differing implementations and realizations (and correspondingly different costs and performances), but a single interface exposed to programmers. Code written for one member of the family was guaranteed to run correctly on all other family members.

By analogy, Direct3D, which was introduced in Chapter 16 and further discussed in Section 15.7, and OpenGL specify the architecture of modern GPUs. They enable code portability just as the System/360 architecture did. But the analogy runs deeper. Both GPU architectures and CPU architectures do the following.

- They allow for differences in configuration, which for CPUs includes memory size, disk storage capacity, and I/O peripherals, and for GPUs includes framebuffer size, texture memory capacity, and specific color codings available in each.
- They make no specification of absolute performance, allowing implementations to cover a wide gamut of cost and optimization.
- They tightly specify the semantics of all inputs, both valid and invalid, to further ensure code compatibility.

An important way that GPU and CPU architectures differ is in level: Direct3D and OpenGL are specified as libraries of function calls, which are compiled into application code, while CPUs are described by instruction set architectures, or ISAs, for which instructions are generated by a compiler. The relatively higher level of abstraction of GPU architecture has given GPU implementors more room to innovate, and it is probably a factor in the historically greater rate of GPU performance increase discussed in the previous section. Both Direct3D and OpenGL were carefully designed to allow highly parallel implementations, for example.

38.3.1 GPU Architecture

While both Direct3D and OpenGL specify their abstractions in exacting detail, as befits an architectural specification, here we do not completely define either

2. Modern usage of the term "architecture" sometimes includes implementation, but we will maintain the distinction carefully in this chapter.

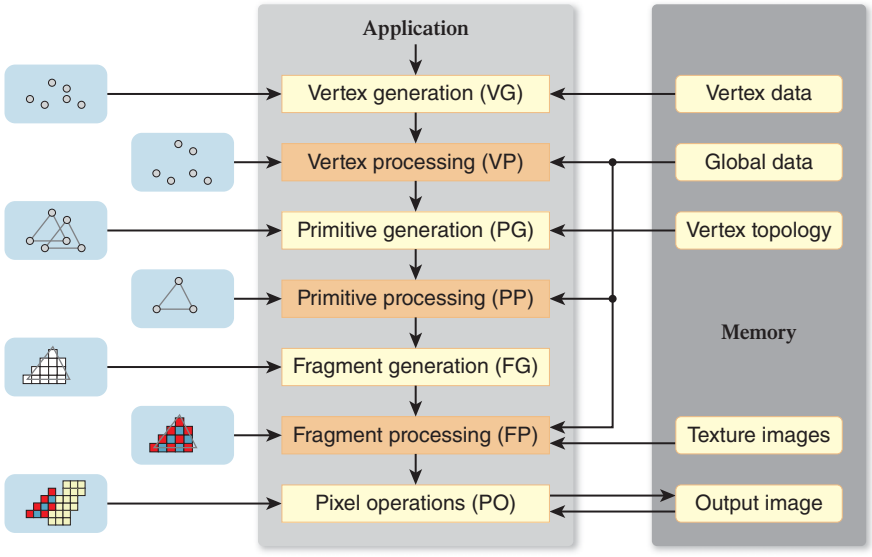


Figure 38.3: Graphics pipeline. This matches both Direct3D and OpenGL at this level of detail. Arrows indicate data flow while drawing.

interface. Instead, we develop a simplified pipeline model that matches both architectures at the chosen level of detail. Figure 38.3 is a block diagram of this architecture.

As you've seen in earlier chapters, graphics architectures perform operations on aggregate data types—vertices, primitives (e.g., triangles), pixel fragments (often just **fragments**), and pixels—and on multidimensional arrays of pixels (e.g., 2D images and 1D, 2D, and 3D textures). Pixels in a texture are called **texels.** Operations are performed in the fixed sequence illustrated in Figure 38.3. The sequence cannot generally be modified by the application, although some stages (e.g., primitive processing) may be omitted.

We briefly review the operation of the graphics pipeline by considering the processing of a single triangle (see Figure 38.3, left column). Prompted by the application, the vertex generation stage creates three vertices from geometric and attribute data (e.g., coordinates and color) stored in memory. The vertices are passed to the vertex processing stage, where operations such as coordinate-space transformations are performed, resulting in homogeneous clip-coordinate vertices. The primitive generation stage assembles the clip-coordinate vertices into a single triangle, possibly accessing topology information in memory to do so. The triangle is passed to the primitive processing stage, where it may be culled, replaced by a finite set of related primitives (e.g., subdivided into four smaller triangles), or left unchanged. After reaching the fragment generation stage, the triangle is clipped against the viewing-frustum boundaries, projected to screen coordinates, then *rasterized* such that a **fragment** (a piece of geometry so small that it contributes to only a single pixel) is generated for each pixel in the output image that geometrically intersects the triangle. The fragment processing stage colors each fragment by performing lighting calculations and accessing texture images. Finally, the colored fragment is merged into the corresponding pixel in the output image. Operations performed in the pixel operation stage, such as z-comparisons and simple

color arithmetic, determine whether and how the fragment values affect or replace corresponding pixel values.

Three of the pipeline stages—the vertex, primitive, and fragment processing stages—are application-programmable. A (typically) simple program, specified by the application using a C-like language, is assigned to the fragment processing stage, where it is executed on each fragment. Likewise, programs are specified and assigned to the vertex and primitive processing stages, where they operate on individual vertices and primitives. Floating-point arithmetic, logical operations, and conditional flow control are supported by each programmable processing stage, as are indexing of global data and filtered sampling (i.e., interpolation) of texture images, both stored in memory. The remaining stages—the vertex, primitive, and fragment generation stages and the pixel operations stage—are referred to as fixed-function stages, because they cannot be programmed by the application. (They can, however, be configured by various modal specifications.)

Modern graphics architectures support two types of commands, those that specify state (the majority) and those that cause rendering to occur. Thus, drawing is a two-step process: 1) Set up all the required state, and then 2) run the pipeline to cause drawing to happen.[3] Vertex data and topology, texture images, and the application-specified programs for the processing stages are all large components of pipeline state. In addition, modal state associated with each fixed-function stage determines the details of its operation (e.g., aliased or antialiased rasterization at the fragment generation stage). Figure 38.3 emphasizes *drawing,* rather than state setup, both by omitting modal state and by indicating read-only access to bulk memory state (e.g., texture images).

Certain properties of the graphics pipeline architecture have great significance for both graphics applications and GPU implementations. One such property is read-only access, during drawing operation, of all bulk memory state other than the output image. (Writing greatly complicates **coherence,** the requirement that memory state values appear consistent to all the processors in a parallel system [see Section 38.7.2].) Perhaps even more significant is the requirement for in-order operation. This applies both to drawing (fragments must reach the output image in the order that their corresponding triangles were specified) and to state changes (drawing activated prior to a state change cannot be affected by that change, and drawing specified after must be). GPU implementors struggle mightily with in-order drawing, while read-only bulk state (state such as texture memory that cannot be modified during rendering operation) simplifies their task. Conversely, application developers are constrained by read-only state semantics, but they are supported by in-order drawing semantics. The art of architecture design includes identifying such conflicts and making the best tradeoffs. If either constituency is entirely happy with the architecture, it probably isn't right. This is so important that we embody it in a principle.

✓ **THE DESIGN TRADEOFF PRINCIPLE:** The art of architecture design includes identifying conflicts between the interests of implementors and users, and making the best tradeoffs.

3. Early OpenGL interfaces blurred this distinction by providing commands that both specified vertex state and resulted in drawing. This is what is usually meant by the term "immediate-mode" rendering, as discussed in Chapter 16.

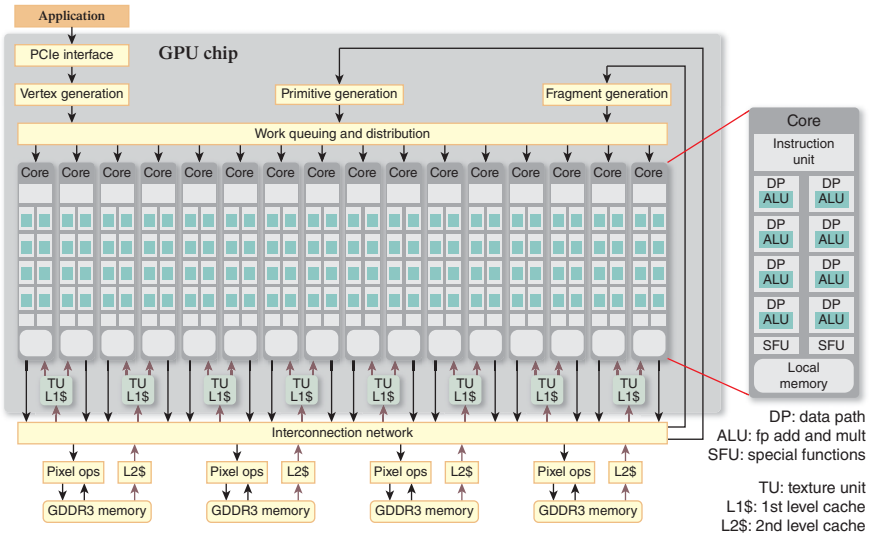


Figure 38.4: NVIDIA *GeForce 9800 GTX block diagram.*

38.3.2 GPU Implementation

Now we turn our attention to an implementation of the graphics pipeline architecture—NVIDIA's GeForce 9800 GTX. Figure 38.4 is a block-diagram depiction of this implementation. Block names have been chosen to best illustrate the correspondence to the blocks on the pipeline block diagram shown in Figure 38.3. There is a one-to-one correspondence between the vertex, primitive, and fragment generation stages in the pipeline and the identically named blocks in the implementation diagram. Conversely, all three programmable stages in the pipeline correspond to the aggregation of 16 cores, eight texture units (TUs), and the block titled "Work queuing and distribution." This highly parallel, application-programmable complex of computing cores and fixed-function hardware is central to the GTX implementation—we'll have much more to say about it. Completing the correspondence, the pixel operation stage of the pipeline corresponds to the four **pixel ops** blocks in the implementation diagram, and the large memory block adjacent to the pipeline corresponds to the aggregation of the eight L1$, four L2$, and four GDDR3 memory blocks in the implementation. The **PCIe interface** and **interconnection network** implementation blocks represent a significant mechanism that has no counterpart in the architectural diagram. A few other significant blocks, such as display refresh (regularly transferring pixels from the output image to the display monitor) and memory controller logic (a complex and highly optimized circuit), have been omitted from the implementation diagram.

38.4 Parallelism

As we have seen, several decades of exponential increase in transistor count and performance have given computer system designers ample resources to design and build high-performance systems. Fundamentally, computing involves performing *operations* on *data*. **Parallelism,** the organization and orchestration of simultaneous operation, is the key to achieving high-performance *operation*. It is the

subject of this section. We defer the discussion of *data*-related performance to Section 38.6.

We are all familiar with doing more than one thing at the same time. For example, you may think about computer graphics architecture while driving your car or brushing your teeth. In computing we say that things are done in parallel if they are in process at the same time. In this chapter, we'll further distinguish between *true* and *virtual* parallelism. **True parallelism** employs separate mechanisms in physically concurrent operation. **Virtual parallelism** employs a single mechanism that is switched rapidly from one sequential task to another, creating the appearance of concurrent operation.

Most parallelism that we are aware of when using a computer is virtual. For example, the motion of a cursor and of the scroll bar it is dragging appear concurrent, but each is computed separately on a single processing unit. By allowing computing resources to be shared, and indeed by allocating resources in proportion to requirements, virtual parallelism facilitates efficiency in computing. But it cannot scale performance beyond the peak that can be delivered by a single computing element.

All computing hardware therefore employs true parallelism to *increase* computing performance. For example, even a so-called scalar processor, which executes a single instruction at a time, is in fact highly parallel in its hardware implementation, employing separate specialized circuits concurrently for address translation, instruction decoding, arithmetic operation, program-counter advancement, and many other operations. At an even finer level of detail, both address translation and arithmetic operations utilize binary-addition circuits that employ per-bit full adders and "fast-carry" networks that all operate in parallel, allowing the result to be computed in the period of a single instruction. In a modern, high-performance integrated circuit the longest sequential path typically employs no more than 20 transistors, yet billions of transistors are employed overall. The circuit *must* be massively parallel.

Because true hardware parallelism is an implementation artifact, it cannot be specified architecturally. Instead, an architecture specifies parallelism that can be implemented either virtually (by sharing hardware circuits) or truly (with separate hardware circuits), or, typically, through a combination of the two. To better understand these alternatives, and to illustrate that parallelism has long been central to computing performance, we will briefly consider the architecture and implementation of a decades-old system: the CRAY-1 supercomputer.

The CRAY-1 was developed by Cray Research, Inc., primarily to satisfy the computing needs of the U.S. Department of Defense (see Figure 38.5). When introduced in 1976, it was the fastest scalar processor in the world: Its cycle time of 12.5 ns supported the execution of 80 million instructions per second.[4] Key to its peak performance of 250 million floating-point operations per second (MFLOPS), however, were special instructions that specified arithmetic operations on vectors of up to 64 operands. Data **vectors** were gathered from memory, stored in 64-location vector registers, and operated on by arithmetic **vector instructions** (e.g., the vector sum or vector per-element product of two vector operands could be computed), and the results returned to main memory.

Figure 38.5: The CRAY-1 supercomputer was the fastest system available when it was introduced in 1976. (Courtesy of Clemens Pfeiffer. Original image at http://upload.wikimedia.org/wikipedia/commons/f/f7/Cray-1-deutsches-museum.jpg.)

4. The analysis in this discussion is slightly simplified, because even in scalar operation the CRAY-1 could in some cases execute two instructions per cycle.

Because these per-element operations had no dependencies (i.e., the sum of any pair of vector elements is independent of the values and sums at all other pairs of vector elements) the 64 operations specified by a vector instruction *could* all be computed at the same time. While such an implementation was possible in principle, it was impractical given the circuit densities that could be achieved using the emitter-coupled transistor logic (ECL) that was employed. Indeed, had such an implementation been possible, its peak performance would have approached $64 \times 80{,}000{,}000 = 5{,}120$ MFLOPS, more than 20 times the peak performance that was actually achieved. Instead, the architectural parallelism of equivalent operations on 64 data pairs was implemented as virtual parallelism—the 64 operations were computed sequentially using a single arithmetic circuit.

The roughly 3*x* improvement in peak arithmetic performance over scalar performance was instead achieved using a parallelism technique called **pipelining.** Two specific circuit approaches were employed. First, because floating-point operations are too complex to be computed by a single ECL circuit in 12.5 ns, the floating-point circuits were divided into stages: 6 for addition, 7 for multiplication, and 14 for reciprocation. Each stage performed a portion of the operation in a single cycle, then forwarded the partial result to the next stage. Thus, floating-point operations were organized into pipelines of sequential stages (e.g., align operands, check for overflow), with the stages operating in parallel and the final stage of each unit producing a result each cycle. The reduced complexity of the individual stages allowed the 12.5 ns cycle time to be achieved, and therefore enabled sustained scalar performance of 80 MFLOPS. Second, a specialized pipelining mechanism called **chaining** allowed the results of one vector instruction to be used as input to a second vector instruction immediately, as they were computed, rather than waiting for the 64 operations of the first vector instruction to be completed. As illustrated in Figure 38.6, this allowed small compound operations on vectors, such as $a \times (b + c)$, to be computed in little more time than was required for a single vector operation. In the best case, with all three floating-point processors active, the combination of stage pipelining and operation chaining allows a performance of 250 MFLOPS to be sustained.

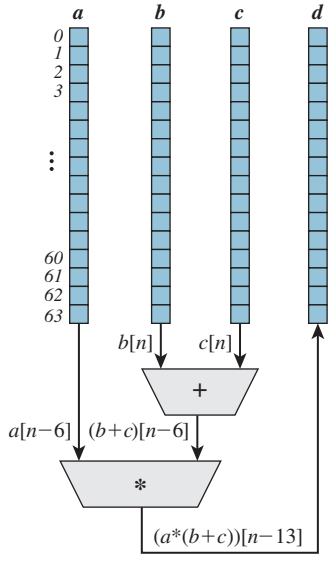


Figure 38.6: Chained evaluation of the vector expression $d = a \times (b + c)$ on the CRAY-1 supercomputer. The floating-point addition unit takes six pipelined steps to compute its result; the multiplication unit takes seven.

Another important characterization of parallelism is into *task* and *data* parallelism. **Data parallelism** is the special case of performing the same operation on equivalently structured, but distinct, data elements. The CRAY-1 vector instructions specify data-parallel operation: The same operation is performed on up to 64 floating-point operand pairs. **Task parallelism** is the general case of performing two or more distinct operations on individual data sets. Pipeline parallelism, such as the CRAY-1 floating-point circuit stages and operation chaining, is a specific organization of task parallelism. Other examples of task parallelism include multiple threads in a concurrent program, multiple processes running on an operating system, and, indeed, multiple operating systems running on a virtual machine.

GPU parallelism can also be characterized using these distinctions. GPU architecture (see Figure 38.3) is a task-parallel pipeline. The GeForce 9800 GTX implementation of this pipeline is a combination of true pipeline parallelism and virtual pipeline parallelism. Fixed-function vertex, primitive, and fragment generation stages are implemented with separate circuits—they are examples of true parallelism. Programmable vertex, primitive, and fragment processing stages are implemented with a single computation engine that is shared among these distinct tasks. Virtualization allows this expensive computation engine, which occupies a significant fraction of the GPU's circuitry, to be allocated dynamically, based on

the instantaneous requirements of the running application. A true-parallel implementation would assign fixed amounts of circuitry to these pipeline tasks, and thus would be efficient only for applications that tuned their vertex, primitive, and fragment workloads to the static, circuit-specified allocations. Such a static allocation *is* forced for the fixed-function stages, but these circuits occupy a small fraction of the overall GPU, so they can be overprovisioned without significantly increasing the cost of the GPU.

Task parallelism, in the form of pipeline stages, is only the top level of a hierarchy of parallelism in the GeForce 9800 GTX implementation, which continues down to small groups of transistors. The next lower layers of parallelism are the 16 task-parallel computing cores, each of which is implemented as a data-parallel, 16-wide vector processor. Unlike the CRAY-1, whose vector implementation was virtual parallel, the GeForce 9800 GTX vector implementation is a hybrid of virtual and true parallel—there is separate circuitry for eight vector data paths, each of which is used twice to operate on a 16-wide vector. (This parallel circuitry is represented by the eight data paths [DPs] in each of the 16 cores of Figure 38.4.) A true-parallel vector implementation is sometimes referred to as SIMD (Single Instruction Multiple Data) because a single decoded instruction is applied to each vector element. SIMD cores are desirable because they yield much more computation (GFLOPS) per unit area of silicon than scalar (SISD) cores do. And compute rate is a high priority for GPU implementations.

It is important to understand that SIMD vector implementation of the GPU cores is not revealed directly in the GPU architecture (as it is in the CRAY-1 vector instructions). While the GPU programming model executes the same program for each element (e.g., each vertex), that vertex program may contain branches, and the branches may be taken differently for each vertex. The implementation includes extra circuitry that implements **predication.** When all 16 vector data elements take the same branch path, operation continues at full efficiency. A single branch taken differently splits the vector data elements into two groups: those that take one branch path and those that take the other. Both paths are executed, one after the other, with execution suppressed for elements that don't belong to the path being executed. (Figure 38.9 in Section 38.7.3 illustrates diverging and nondiverging predicated execution.) Nested branches may further split the groups. In the limit separate execution could be required for each element, but this limit is rarely reached. Thus, a Single *Program* Multiple Data (SPMD) architecture is implemented with predicated SIMD computation cores.

38.5 Programmability

We begin our discussion of programmability by examining a simple coding example. Listing 38.1 is a complete fragment processing program written in the Direct3D High-Level Shading Language (HLSL). This program specifies the operations that are applied to each pixel fragment[5] resulting from the rasterization

5. Use of the term "fragment" to distinguish the data structures generated by rasterization (and operated upon by the fragment processing pipeline stage) from the data structures stored in the framebuffer (pixels) was established by OpenGL in 1992 and has since become an accepted industry standard. Microsoft Direct3D and HLSL blur this distinction by referring to both data structures as *pixels*. The confusion is compounded by the HLSL use of "fragment" to refer to a separable piece of HLSL code. The reader is warned.

of a primitive. While more complex operations are often specified, in this case the operation is a simple one: The red, green, blue, and alpha components of the fragment's color are attenuated by the floating-point variable `brightness`. The term **shader,** taken from Pixar's RenderMan Shading Language, is often used to refer to fragment processing programs (and also to vertex processing and primitive processing programs) although they have far more functionality than simply computing color.

Listing 38.1: A simple HLSL fragment processing program (Microsoft refers to this as a "pixel shader"). Fragment color is attenuated by the floating-point variable `brightness`.

```
1  float brightness = 0.5;
2
3  struct v2f {
4      float4 Position : POSITION;
5      float4 Color : COLOR0;
6  };
7
8  struct f2p {
9      float Depth : DEPTH0;
10     float4 Color : COLOR0;
11 }
12
13 // attenuate fragment brightness
14 void main(in v2f Input, out f2p Output) {
15     Output.Depth = Input.Position.z;
16     Output.Color = brightness * Input.Color;
17 }
```

HLSL is designed to be familiar to C and C++ programmers. As it would in C++, the first line of the shader declares `brightness` as a global, floating-point variable with initial value 0.5. Again, as in C++, because `brightness` is a global variable, its value can be changed by nonlocal code. In HLSL such nonlocal code includes the code of the Direct3D application that is driving the GPU pipeline. For example, the application can change the value of `brightness` to 0.75 by calling the Direct3D function.

```
SetFloat("brightness",0.75);
```

Structures `v2f` and `f2p` are defined much as they would be in C++. There are two important additions: vector data types and semantics. **Vector data types,** such as `float4`, are a shorthand notation that is similar to a C++ array typedef (in this case, `typedef float[4] float4;`), but with additional capabilities. This notation can be used for vectors of lengths two, three, and four. It is also available for square matrices (e.g., `float4x4`) and for all HLSL data types (i.e., for `half`, `float`, `double`, `int`, and `bool`). **Semantics** are the predefined tokens, such as `POSITION` and `COLOR0`, that are separated by colons from the variable names (such as `Position` and `Color`). These tokens associate HLSL-program variables with mechanisms in the fixed-function stages of the pipeline—they are the glue that connects application-specified shaders to architecture-specified fixed-function mechanisms.

Operation of the fragment shader is specified by the `main` function. Input to `main` is through structure variable `Input` of type `v2f`, and output is through

structure variable `Output` of type `f2p`. In the body of `main` the output depth value is copied from the input depth value, while the output color is set to the attenuated input color. Note that HLSL defines multiplication of a vector value (`Input.Color`) by a scalar value (`brightness`) in the mathematical sense: The result is a vector of the same dimension, with each component multiplied by the scalar value.

Beginning with Direct3D 10 (2006) HLSL is the only way that programmers can specify a vertex, primitive, or fragment shader using Direct3D. Earlier versions of Direct3D included assembly-language–like interfaces that were deprecated in Direct3D 9 and are unavailable in Direct3D 10. Thus, HLSL *is* the portion of the Direct3D architecture through which programmers specify the operation of a shader. While it is not the intention of this section to provide a coding tutorial for Direct3D shaders, this simple example includes many of the key ideas.

A useful metaphor for the operation of this simple shader is that of a heater operating on a stream of flowing water. Pixel fragments arrive in an ordered sequence, like water flowing through a pipe. A simple operation is applied to each pixel fragment, just as the heater warms each unit of water. Finally, the pixel fragments are sent along for further processing, just as water flows out of the heater through the exit pipe. Indeed, this metaphor is so apt that GPU processing is often referred to as **stream processing.** That's how we treat GPU programmability in this section, although the following section (Section 38.6) will consider an important exception with significant implications for GPU implementation.

Writing highly parallel code on a general-purpose CPU is difficult—only a small subset of computer programmers is thought to be able to achieve reliable operation and scalable performance of parallel code.[6] Yet writing shaders is a straightforward task: Even novice programmers achieve correct and high-performance results. The difference in difficulty is the result of differences between the general-purpose architectures of CPUs and the special-purpose architectures of GPUs. The Single Program Multiple Data (SPMD) abstraction employed by GPUs allows shader writers to think of only a single vertex, primitive, or fragment while they code—the implementation takes care of all the details required to execute the shaders in the correct order, on the correct data, while efficiently utilizing the data-parallel circuitry. CPU programmers, on the other hand, must carefully consider parallelism while they code, because their code specifies the details of any thread-level parallelism that is to be supported.

While GPU programming is experienced as a recent and exciting development, the ability to program GPUs is as old as GPUs themselves. The Ikonas graphics system [Eng86], introduced in 1978, exposed a fully programmable architecture to application developers. Ten years later Trancept introduced the TAAC-1 graphics processor, which included a C-language microcode compiler to simplify the development of application-specific code. The architectures of these early GPUs were akin to extended CPU architectures; they bore little similarity to the more specialized pipeline architecture that is the subject of this chapter, and that has coevolved with mainstream GPU implementations since the early 1980s. Unlike these CPU-like architectures, the pipeline architecture was without support for application-specified programs (shaders) until both OpenGL and Direct3D were

6. It's easy to write code that exploits the circuit-level parallelism of the CPU, and operating systems make it easy to exploit virtual parallelism by concurrently executing multiple programs. The parallelism that is difficult to exploit is at the intermediate level: multiple-thread task parallelism within a single program.

extended in 2001. Since that time support for shaders has become a defining feature of GPUs.

Despite the delay in exposing programmability to application programmers, from the start essentially all *implementations* of OpenGL and Direct3D were programmed by their developers. For example, the programmable logic array (PLA) that specified the behavior of the Geometry Engine, which formed the core of Silicon Graphics' graphics business in the early '80s, was programmed prior to circuit fabrication using a Stanford-developed microcode assembler. Subsequent Silicon Graphics GPUs included microcoded compute engines, which could be reprogrammed after delivery as a software update. But application programmers were not allowed to specify these program changes. Instead, they were limited to specifying modes that determined GPU operation within the constraints of the seemingly "hardwired" pipeline architecture.

There are several reasons that the programmability of mainstream GPUs remained hidden behind modal architectural interfaces throughout the '80s and '90s. Directly exposing implementation programming models, which differed substantially from one GPU to another, would have forced applications to be recoded as technology evolved. This would have broken forward compatibility, a key tenet of any computing architecture. (Programmers reasonably expect their code to run without change, albeit faster, on future systems.) Today's GPU drivers solve this problem by cross-compiling high-level, implementation-*independent* shaders into low-level, implementation-*dependent* microcode. But these software technologies were just maturing during the '80s and '90s; they would have executed too slowly on CPUs of that era.

These reasons and others can be summarized in the context of exponential performance and complexity advances.

- **Need:** Increasing GPU performance created the need for application-specified programmability, both because more complex operations *could* be supported at interactive rates, and because modal specification of these operations became prohibitively complex.
- **GPU hardware capability:** High-performance GPUs of the '80s and '90s were implemented using multiple components sourced from various vendors. But increased transistor count allows modern GPUs to be implemented as single integrated circuits. Single-chip implementations give implementors much more control, simplifying cross-compilation from a high-level language to the GPU microcode by minimizing implementation differences from one product generation to the next.
- **CPU hardware capability:** Increasing CPU performance allowed GPU driver software to perform the (simplified) cross-compilation from high-level language to hardware microcode with adequate performance (both of the compilation and of the resultant microcode).

38.6 Texture, Memory, and Latency

Thus far our discussions of parallelism and programmability have treated the graphics pipeline as a **stream processor.** Such a processor operates on individual, predefined, and (typically) small data units, such as the vertices, primitives, and fragments of the graphics pipeline, without accessing data from a larger external memory. But the graphics pipeline is in reality not so limited. Instead, as can

be seen in Figure 38.3, most pipeline stages have access to a generalized memory system, in addition to the data arriving from the previous stage. In this section we consider both the opportunities and the performance challenges of such memory access, using texture mapping as an archetypal example.

38.6.1 Texture Mapping

Recall from Chapter 20 that texture mapping maps image data onto a primitive. In the graphics pipeline this is implemented in two steps: *correspondence* and *evaluation*.

Correspondence specifies a mapping from the geometric coordinates of the primitive to the image-space coordinates of the texture image. In the modern graphics pipeline (see Figure 38.3) correspondence is specified by assigning texture image coordinates to the vertices of primitives. The assignment may be done directly by the application, using interfaces such as OpenGL's `TexCoord*` commands, or indirectly through calculations specified by the vertex shader (i.e., the application-specified program executing on the vertex processing pipeline stage can generate texture coordinates, or modify application-specified texture coordinates). To determine the correspondence at sample locations within individual pixel fragments, the texture coordinates specified at vertices are linearly interpolated to the sample's corresponding position within the primitive. Texture coordinate interpolation to the centers of pixel fragments is implemented as a part of the rasterization process in the fragment generation pipeline stage. Because interpolation is to be linear in the primitive's coordinates, but it is implemented in the warped, post-projection coordinates of the framebuffer, the mathematics require a high-precision division for each correspondence calculation (typically, one division for each pixel fragment that is generated by rasterization). While this division is inexpensive to implement in modern GPUs, it was a nearly insurmountable burden prior to the '80s, and it is a significant reason why texture mapping was available only in expensive, specialized graphics systems such as flight training simulators prior to that.

Evaluation is also an interpolation. But unlike the interpolation of texture coordinates, which involves accessing only the few coordinates assigned to the vertices of a primitive, evaluation accesses multiple pixels (called **texels**) within a texture image. And texture images can be very large—Direct3D 10 implementations must support $4{,}096 \times 4{,}096$-texel images, which require a minimum of 8 million bytes of storage. Thus, large-memory access, which is typically not required for texture correspondence, is fundamental to texture evaluation.

Images are equivalent to 2D tables of colors, and table-driven interpolation—constructing new data points within a regularly spaced range of discrete data points by computing a weighted average of nearby data points—is a fundamental mathematical operation that is available in many nongraphical systems. The heavily used `interp1`, `interp2`, and `interp3` commands in MATLAB are good examples: They construct new points within ranges specified as 1D, 2D, or 3D arrays. Texture image interpolation has always been a part of OpenGL and Direct3D, but prior to application programmability it was hidden within their fixed-function texture-mapping mechanisms.

Now, like MATLAB, both the Direct3D and OpenGL shading languages expose 1D, 2D, and 3D versions of table interpolation, which operate on 1D, 2D, and 3D texture images. Critically, these image-interpolation functions accept

parameters that specify the image position to be evaluated. (Regardless of the image size, parameter values between 0 and 1 specify positions from one edge of the image to the other. Parameter values outside $[0, 1]$ may wrap, reflect, or clamp, as specified ahead of time.) Thus, a shader can specify coordinates that have been interpolated across the primitive during rasterization (i.e., they can implement traditional texture correspondence) or they can choose to specify coordinates that have been computed arbitrarily, perhaps as functions of the results of other texture image interpolations. Such *dependent* texture image interpolation is a very powerful feature of modern GPUs, but, as we will see in Section 38.7.2, it complicates their implementation.

Listing 38.2 replaces the constant-based color attenuation of the example fragment shader in Listing 38.1 with attenuation defined by a 1D texture image: `brightness_table`. Like the global variable `brightness`, `brightness_table` is specified by the application prior to rendering and remains constant throughout rendering. Unlike `brightness`, however, which was stored in a GPU register (e.g., within a *core* in Figure 38.4), `brightness_table` is bulk memory state (see Section 38.3), stored in the off-chip GPU memory system (e.g., *GDDR3 memory* in Figure 38.4). Sampler `s` is a simple data structure that specifies a texture image (`brightness_table`) and the technique to be used to evaluate it (linear interpolation between texels).

Listing 38.2: Another simple HLSL fragment shader.

```
texture1D brightness_table;

sampler1D s = sampler_State {
    texture = brightness_table;
    filter = LINEAR;
};

struct v2f {
    float4 Position : POSITION;
    float4 Color : COLOR0;
    float TexCoord : TEXCOORD0;
};

struct f2p {
    float Depth : DEPTH0;
    float4 Color : COLOR0;
}

// attenuate fragment brightness with a 1-D texture
void main(in v2f Input, out f2p Output) {
    Output.Depth = Input.Position.z;
    Output.Color = tex1D(s, Input.TexCoord) * Input.Color;
}
```

The input structure `v2f` is extended with a third component, `TexCoord`, with semantic `TEXCOORD0`, which specifies the texture location to be evaluated. Within `main`, the 1D texture interpolation function `tex1D` is called with sampler `s` (specifying that `brightness_table` is to be interpolated linearly) and coordinate `Input.TexCoord` (indicating where `brightness_table` is to be evaluated), and it returns the floating-point result. The four-component vector `Input.Color` is scaled by this value, and the result, an attenuated, four-component color vector, is assigned to `Output.Color`.

The process of texture image interpolation can be further partitioned into three steps.

1. **Address calculation:** Based on the specified interpolation coordinates, the memory addresses of the nearby texels are computed.
2. **Data access:** The texels are fetched from memory.
3. **Weighted summation:** Based on the specified interpolation coordinates, an individual fractional weight is computed and applied to each accessed texel, and the sum of the weighted texels is computed and returned.

GPUs such as the GeForce 9800 GTX implement these sequential steps using true, hardware-instantiated parallelism: The steps are implemented as a sequential pipeline of data-parallel operations. Among these steps weighted summation parallelizes particularly well: The 1D interpolation in our example sums only two weighted values, but interpolations of 2D and 3D images sum four and eight values, respectively.

Image interpolation samples images accurately, but high-quality image reconstruction requires both accuracy *and* sufficiency of image samples. Reconstruction from an insufficient number of samples results in aliasing: low-frequency artifacts in the reconstructed image corresponding to high-frequency components in the sampled image. These artifacts are objectionable in still images (see Figures 18.10 and 18.11) but are particularly annoying in dynamic 3D graphics because they introduce false motions (e.g., shimmering and flickering) that strongly and incorrectly direct the viewer's attention. Modern graphics systems are designed to minimize aliasing artifacts.

Typically the image is reconstructed in the framebuffer, with each pixel's color specified by a pixel-fragment shader that takes a single sample of the texture image (as does the example shader in Listing 38.1). In this case adequate image sampling is ensured only when the texture-image samples corresponding to two adjacent framebuffer pixels are separated in texture coordinates by no more than a single texel.[7] Texture-coordinate separations corresponding to unit-pixel separations are a complex function of model geometry, transformation, and projection, and cannot in general be predicted prior to rasterization. So a fragment shader risks undersampling and aliasing unless it is written to detect and, when necessary, compensate for undersampling. One reason that writing such shaders is undesirable is that, in the worst case (when the entire texture corresponds to a single framebuffer pixel), compensation would require that a single fragment shader access, weight, and sum every texel in the texture image!

GPUs solve this problem by separating the process into two parts: a prerendering phase that is executed only once, and additional rendering semantics that enable interpolation instructions such as `tex1D` to detect undersampling, and, when necessary, compensate for it *in constant time*. This two-phase approach, called **MIP mapping,** was first published by Lance Williams in 1983 [Wil83].

During the prerendering phase a MIP map, comprising the original texture image and a sequence of reduced-size versions of it, is computed (see Figure 38.7). The dimensions of the original image must all be powers of two, though they need not be equal. A half-size version of this image is computed, ideally using high-quality filtering to avoid aliasing. This process is repeated (quarter-size,

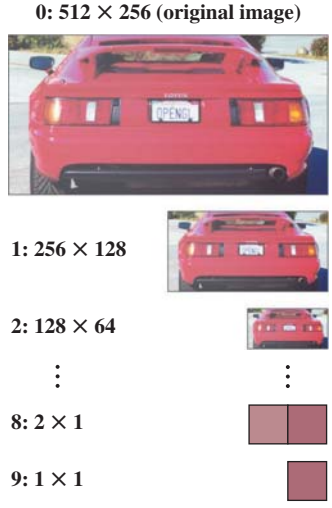


Figure 38.7: A MIP map includes the original image and repeated half-size reductions of it. The smallest image is a single pixel.

7. Recall that Chapter 18 discusses sampling, reconstruction, and aliasing.

eighth-size, etc.) until an image with all dimensions equal to one results. If the dimensions of the original image differ, smaller dimensions reach unit length early—they remain unit length until the algorithm terminates.

Triangle rasterization generates pixel fragments in groups of four, called **quad fragments,** with each group corresponding to a 2×2-pixel framebuffer region. The sole purpose of rasterizing to quad fragments is to allow sample-to-sample separations to be computed reliably and inexpensively.[8] Separations in 1D textures are simple differences. Separations in 2D and 3D textures are correctly computed as square roots of summed squares of differences, but conservative approximations such as sum of differences are sometimes employed to reduce computation. Both left-to-right and top-to-bottom separations are computed. From these separations a single representative separation ρ is computed. If $\rho \leq 1$, the texture image is sampled sufficiently—texture image interpolation is performed on the original texture image and the resultant color is returned.

If $\rho > 1$, interpolation of the original texture image may result in aliasing. But interpolation of the first MIP-map image (half-size) will alias only if $\rho > 2$, and interpolation of the second MIP-map image (quarter-size) will alias only if $\rho > 4$. Thus, aliasing is always avoided when interpolation is performed on the nth MIP-map image, where $n = \lceil \log_2 \rho \rceil$.

While interpolating the nth MIP-map image in isolation avoids aliasing, it introduces artifacts of its own, especially in dynamic 3D graphics, because screen-adjacent pixels for which different values of n are computed will be colored by different MIP-map images, and these discontinuities will move with object and camera movements. An additional interpolation is required to avoid this.[9] Both the nth and the $(n-1)$th MIP-map images are interpolated ($n = 0$ implies the original texture image). Then these two interpolated values are themselves interpolated, using $\lceil \log_2 \rho \rceil - \log_2 \rho$ as the weighting factor. The term **trilinear MIP mapping** is sometimes used to describe this common MIP-mapping algorithm, when it is applied with 2D texture images, because interpolation is done in three dimensions: two spatial and one between images. But this term is imprecise and should be avoided because, for example, a 3D texture sampled at only one MIP-map level is also trilinearly interpolated.

Texture mapping algorithms in modern GPUs are immensely complicated—many important details have been simplified or even ignored in this short discussion. For a concise but fully detailed description of texture mapping, or indeed of any particular of GPU architecture, refer to the OpenGL specification. The latest version of this specification is always available at www.opengl.org.

38.6.2 Memory Basics

To understand how and why memory access complicates the implementation of high-performance GPUs, and in particular their texture-mapping capability, we now detour into the study of memory itself.

8. There are costs associated with quad-fragment rasterization too. See Fatahalian et al. [FBH+10] for an example.
9. This problem and its solution are a recurring theme in dynamic 3D graphics: Any algorithm that can introduce frame-to-frame discontinuities is interpolated to avoid them.

Memory is state. In this abstract, theoretical sense, there is no difference between a bit stored in a memory chip, a bit stored in a register, and a bit that is part of a finite-state machine. Each exists in one of two conditions—true or false—and each can have its value queried or specified. In this sense, all bits are created equal.

But all bits are not created equal. The practical difference is their location; specifically, the distance between a bit's physical position and the position of the circuit that depends on (or changes) its value. Ideally, all bits would be located immediately adjacent to their related circuits. But in practice only a few bits can be so located, for many reasons, but fundamentally because bit storage implementations must be physically distinct.[10] Thus, it is capacity that distinguishes what we call **memory** (large aggregations of bits that are necessarily distant from their related circuits) from simple **state** (small aggregations of bits that are intermingled with their related circuits).

Why does distance matter? Fundamentally, because information (the value of a bit) cannot travel faster than $1/3$ of a meter per nanosecond. In our daily experience this limit—the speed of light in a vacuum—is insignificant; we are unaware that the propagation of light takes any time at all. But in high-performance systems this limit is crucial and very apparent. For example, a bit can travel no more than 10 cm in the 0.31 ns clock period of the (3.2 GHz) Intel Core 2 Extreme QX9770 CPU (see Figure 38.1). And this is in the best case, which is never achieved! In practice, signal propagation never exceeds $1/2$ the speed of light, and easily falls to $1/10$ that speed or less in dense circuitry. That's a thumbnail width or less in a single clock cycle. To quote Richard Russell, a designer of the CRAY-1 supercomputer, "In computer design it is a truism that smaller means faster." Indeed, the love-seat-like shape of the CRAY-1 (see Figure 38.5) was chosen to minimize the lengths of its signal wires [Rus78].

In practice, distance not only increases the time required for a single bit to propagate to or from memory, it also reduces **bandwidth:** the rate that bits can be propagated. Individual state bits can be wired one-to-one with their related circuits, so *state* can be propagated in parallel, and there is effectively no limit to its propagation rate. But larger memories cannot be wired one-to-one, because the cost of the wiring is too great: Too many wires are required, and their individual costs increase with their length. In particular, wires that connect one integrated circuit to another are much more expensive (and, to compound the problem, consume far more power and are much slower) than wires within an individual integrated circuit. But large memories are optimized with specialized fabrication techniques that differ from the techniques used for logic and state, so they are typically implemented as separate integrated circuits. For example, the four GDDR3 memory blocks in Figure 38.4 are separate integrated circuits, each storing one billion bits. But they are connected to the GeForce 9800 GTX GPU with only 256 wires—a ratio of 16 million bits per wire.

Running fewer wires between memory and related circuits obviously reduces bandwidth. It also increases delay. To query or modify a memory bit a circuit must, of course, specify which bit. We refer to such specification as **addressing** the memory. Because bits are mapped many-to-one with wires, addressing must be implemented (at least partially) on the *memory* end of the wires. Thus, a circuit

10. Quantum storage approaches, such as holography, may ease this limitation in the future, but they are not used in today's computing systems.

that queries memory must endure two wire-propagation delays: one for the address to travel to the memory, and a second for the addressed data to be returned. The time required for these two propagations, plus the time required for the memory circuit itself to be queried, is referred to as memory **latency.**[11] Together, latency and bandwidth are the two most important memory-related constraints that system implementors must contend with.

With this somewhat theoretical background in hand, let's consider the important practical example of dynamic random-access memory (DRAM). DRAM is important because its combination of tremendous storage capacity (in 2009 individual DRAM chips stored four billion bits) and high read-write performance make it the best choice for most large-scale computer memory systems. For example, both the DDR3-based CPU memory and the GDDR3-based GPU memory in the PC block diagram of Figure 38.1 are implemented with DRAM technology.

Because modern integrated circuit technology is a planar technology, DRAM is organized as a 2D array of 1-bit memory cells (see Figure 38.8). Perhaps surprisingly, DRAM cells cannot be read or written individually. Instead, individual bit operations specified at the interface of the DRAM are implemented internally as operations on **blocks** of memory cells. (Each row in the memory array in Figure 38.8 is a single block.) When a bit is read, for example, all the bits in the block that contains the specified bit are transferred to a block buffer at the edge of the 2D array. Then the specified bit is taken from the block buffer and delivered to the requesting circuit. Writing a bit is a three-step process: All the bits in the specified block are transferred from the array to the block buffer; the specified bit in the block buffer is changed to its new value; then the contents of the (modified) block buffer are written back to the memory array.

While early DRAMs hid this complexity behind a simple interface protocol, modern DRAMs expose their internal resources and operations.[12] The GDDR3 DRAM used by the GeForce 9800 GTX, for example, implements four separate block buffers, each of which can be loaded from the memory array, modified, and written back to the array as individually specified, and in some cases concurrent, operations. Thus, the circuit connected to the GDDR3 DRAM does more than just read or write bits—it manages the memory as a complex, optimizable subsystem. Indeed, the memory control circuits of modern GPUs are large, carefully engineered subsystems that contribute greatly to overall system performance.

But what performance is optimized? In practice, the DRAM memory controller cannot simultaneously maximize *bandwidth* and minimize *latency*. For example, sorting requests so that operations affecting the same block can be aggregated minimizes transfers between the block buffers and the array, thereby optimizing bandwidth at some expense to latency. Because the performance of modern GPUs is frequently limited by the available memory bandwidth, their optimization is skewed toward bandwidth. The result is that total memory latency—through the memory controller, to the DRAM, within the DRAM, back to the GPU, and again through the memory controller—can and does reach *hundreds* of computation cycles. This observation is our second hardware principle.

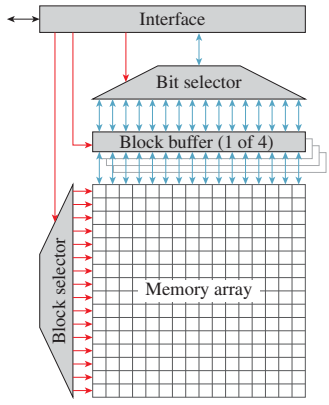


Figure 38.8: Block diagram of a simplified GDDR3 memory circuit. For increased clarity, the true storage capacity (one billion bits) is reduced to 256 bits, implemented as an array of sixteen 16-bit blocks (a.k.a. rows). The red arrows arriving at the left edges of blocks indicate control paths, while the blue ones meeting the tops and bottoms of blocks are data paths.

11. Note that latency affects only memory reads. It can be hidden during writes with pipeline parallelism, as discussed in Section 38.4.
12. Using the terminology of this chapter, DRAM architecture has evolved to more closely match DRAM implementation.

✓ **THE MEMORY PRINCIPLE:** The primary challenge of memory is coping with access latency and limited bandwidth. Capacity is a secondary concern.

The way that GPUs such as the GeForce 9800 GTX handle this (sometimes) huge memory latency is the subject of the next subsection.

38.6.3 Coping with Latency

Consider again the fragment shader of Listing 38.2. Ignoring execution of the `tex1D` instruction, this shader makes five floating-point assignments and performs four floating-point multiplications, for a total of nine operations. Again, ignoring texture interpolation, we would expect its execution to require approximately ten clock cycles, perhaps fewer if the GPU's data path implementation supports hardware parallelism for short vector operations. (The GeForce 9800 GTX data path does not.) Unfortunately, even if the GPU implementation provides separate hardware for the computations required by image interpolation (the GeForce 9800 GTX, like most modern GPUs, does) the execution time of the texture-based fragment shader could increase to hundreds or even thousands of cycles due to the latency of the memory reads required to gather the values of the texels. Thus, the performance of a naive implementation of this shader could be reduced by a factor of ten to one hundred, or more, due to memory latency.

There are three potentially legitimate responses to this situation: 1) accept it; 2) take further steps to reduce memory latency; or 3) arrange for the system to do something else while waiting on memory. Of course, options (2) and (3) may be combined.

The engineering option of accepting a nonoptimal situation must always be considered. Just as code optimization is best directed by thorough performance profiling, hardware optimization is justified only by a significant improvement in dynamic (i.e., real-world) performance. If texture interpolation were extremely rare, its low performance would have little real-world effect. In fact, texture interpolation is ubiquitous in modern shaders, so its optimization is of paramount concern to GPU implementors. Something must be done.

Once the memory controller has been optimized, further reduction of memory latency may be achieved by **caching.** Briefly, caching augments a homogeneous, large (and therefore distant and high-latency) memory with a hierarchy of variously sized memories, the smallest placed nearest the requesting circuitry, and the largest most distant from it. All modern GPUs implement caching for texture image interpolation. However, unlike CPUs such as the Intel Core 2 Extreme QX9770, which depend primarily on their large (four-level) cache systems for both memory bandwidth *and* latency optimization, GPU caches are large enough to ensure that available memory bandwidth is utilized efficiently, but they are too small to adequately reduce memory latency. We leave further discussion of the important topic of caching to Section 38.7.2.

Because options (1) and (2) do not adequately address latency concerns, the performance of GPUs such as the GeForce 9800 GTX depends heavily on their implementation of option (3): arranging for the GPU to do something else while waiting on memory. The technique they employ is called **multithreading.** A thread is the dynamic, nonmemory state (such as the program counter and register

contents) that is modified during the execution of a task. The idea of multithreading will not be new to you—it's related to ideas already discussed in Section 38.4. There we learned that the (architecturally specified) programmable vertex, primitive, and fragment processing stages are implemented with a single computation engine that is shared between these distinct tasks. Multithreading is the name of such a virtual-parallel implementation. When multiple tasks share a single processor, the thread of the currently executing task is stored (and modified) in the program counter and registers of the processor itself, while the threads of the remaining tasks are saved unchanged in **thread store.** Changing which task is executing on the processor involves swapping two threads: The thread of the currently active task is copied from the processor into thread store, and then the thread of the next-to-execute task is copied from thread store into the processor.

Multithreading implementations are distinguished by their scheduling techniques. Scheduling determines two things: when to swap threads, and which inactive thread should become active. Interleaved scheduling cycles through threads in a regular sequence, allocating a fixed (though not necessarily equal) number of execution cycles to each thread in turn. Block scheduling executes the active thread until it cannot be advanced, because it is waiting on an external dependency such as a memory read operation, or an internal dependency such as a multicycle ALU operation, and then swaps this thread for a thread that is runnable. Two queues of thread IDs are maintained: a queue of blocked threads and a queue of runnable threads. When a thread becomes blocked its ID is appended to the blocked queue. IDs of threads that become unblocked are moved from the blocked queue to the run queue as their status changes.

GPUs such as the GeForce 9800 GTX implement multithreading with a hierarchical combination of interleaved and block scheduling. The GeForce 9800 GTX hardware enables zero-cycle replacement of blocked threads: No processor cycles are lost during swaps. Because there is no performance penalty for swapping threads, the GeForce 9800 GTX implements a simple static load balancing by swapping every cycle, looping through the threads in the run queue. Load balancing between tasks of different types—vertex, primitive, and fragment—is implemented by including different proportions of these tasks in the mix of threads that is executed on a single core. This per-thread-group load balancing (the mix can't be changed during the execution of a group of such threads) is adjusted from thread group to thread group based on queue depths for the various task types.

Threads are small by the standards of DRAM capacity—on the order of 2,000 bytes each (roughly 128 bytes per vector element). This suggests that thread stores would contain large numbers of threads, but in fact they do not. The GeForce 9800 GTX, for example, stores a maximum of 48 threads per processing core in expensive, on-chip memory. Once again memory latency is the culprit. To support zero-cycle thread swaps, there must be near-immediate access to threads on the run queue. Thus, thread store must have low latency, and is necessarily small and local. (In fact, the GeForce 9800 GTX stores all threads in a single register file. Threads are not swapped in and out at all; instead, the addressing of the register is offset based on which thread is being executed.) More generally, because multithreading compensates for DRAM latency, it is impractical for its implementation to experience (and be complicated by) that same latency. Thus, thread store is an expensive and scarce resource.

Because thread store is a scarce resource whose capacity has a significant effect on performance (the processor remains stalled while the run queue is empty) optimizations that reduce thread storage requirements are vigorously pursued. We discuss two such optimizations, both of which are implemented by the GeForce 9800 GTX.

First we consider thread size. Because register contents constitute a significant fraction of a thread's state, the size of a thread can be meaningfully reduced by saving and restoring only the contents of "active" registers. In principle, register activity could be tracked throughout shader execution so that thread-store usage varied depending on the program counter of the blocked thread. In practice, thread size is fixed, throughout the execution of a shader, to accommodate the maximum number of registers that will be in use at any point during its execution. Because the shader compiler is part of the GPU implementation (recall that shader architecture is specified as a high-level language interface) the implementation not only knows the peak register usage, but also can influence it as a compilation optimization.

Thread size matters because the finite thread store holds more small threads than large ones, decreasing the chances of the run queue becoming empty and the processor stalling. While this relationship is easily understood, programmers are sometimes surprised when their attempt to optimize shader performance by minimizing execution length (the number of instructions executed) reduces performance rather than increasing it. Typically (compiled) shader length and register usage trade off against each other—that is, shorter, heavily optimized programs use more registers than longer, less optimized ones—hence, the counterintuitive tuning results. Modern GPU shader compilers include heuristics to optimize this tradeoff, but even experienced coders are sometimes confounded.

Performance may also be optimized by running threads longer, thereby keeping more threads in the run queue. A naive scheduler might immediately block a thread on a memory read (or an instruction such as `tex1D` that is known to read data from memory) because it is rightly confident that the requested data will not be available in the next cycle. But the requested data may not be *required* during the next cycle—perhaps the thread will execute several instructions that do not depend on the requested data before executing an instruction that does. A hardware technique known as **score boarding** detects dependencies when they actually occur, allowing thread execution to continue until a dependency is reached, thereby avoiding stalls by keeping more threads in the run queue. It is good programming practice to sequence source code such that dependencies are pushed forward in the code as far as possible, but shader compilers are optimized to find such re-ordering opportunities regardless of the code's structure.

While hiding memory latency with multithreaded processor cores is a defining trait of modern GPUs, the practice has a long history in CPU organization. The CRAY-1 did not use multithreading, but the CDC 6600, an early-1960s Seymour Cray design that preceded the CRAY-1, did [Tho61]. It included a **register barrel** that implemented a combination of pipeline parallelism and multithreading, rotating through ten threads, each at a different stage in the execution of its ten-clock instruction cycle. The Stellar GS 1000, a graphics supercomputer built in the late '80s, executed four threads in a round-robin order on its vector-processing main CPU, which also accelerated graphics operations [ABM88]. Most Intel processors in the IA-32 family implement "hyperthreading," Intel's branded version of multithreaded execution.

38.7 Locality

A common rule of thumb is that programs spend 90% of their time running only 10% of their program code. The 90/10 rule is an approximation—for example, code percentages varied between six and 57 in tests reported by John Hennessy and David Patterson [HP96]—but it suggests an essential truth, which is that the outcomes of computing *events,* such as generating an address or taking a branch, are not evenly distributed. Instead, these events, when treated as random variables, have probability distributions that are unfair (meaning that outcomes have different probabilities) and dependent (meaning that the probability distributions themselves change with the history of recent events).

An unfair coin is undesirable, but unfair and dependent probability distributions in computing are an important opportunity—they allow system designers to make optimizations that dramatically improve performance and efficiency. In computer science the term **locality** describes the nature of the unfair, dependent probability distributions that are observed in functional computing systems. In this section we define various forms of locality and study how system designers take advantage of them.

38.7.1 Locality of Reference

In treating memory abstractly as an ordered array of items, two observations have been made about the patterns of access to these items during the execution of a program. First, an item that has been accessed recently has an increased probability of being accessed again. This program property is called **temporal locality.** Second, items whose addresses are similar tend to be accessed close together in time. This program property, though it is also time-dependent, is called **spatial locality** to emphasize its application to multiple items. Together, temporal and spatial locality constitute **locality of reference.**

While locality of reference is observed in all computing systems, design decisions significantly affect how much is observed. Consider the sequence in which instructions are fetched. A computer could be designed such that each instruction specified the address of the subsequent instruction, allowing execution to skip arbitrarily through the code. But this approach is never taken.[13] Instead, designers universally choose to execute instructions sequentially, except in the special case of branching. This choice directly increases spatial locality in an obvious way. Because most branches cause short sequences of instructions to be repeated (i.e., they *loop*), sequential instruction fetching also increases temporal locality. Indeed, this single decision frequently ensures that locality of reference is greater for instruction accesses than it is for data accesses, despite designers' attempts to increase the latter.

Because GPU shaders are typically much smaller than the data structures they access, GPU designers are more concerned with data locality than they are with code locality. An example of a design decision that greatly affects GPU data locality is the way that 2D texel coordinates, which specify a single texel in a 2D texture image, are mapped to the (1D) memory addresses of the texel-data storage locations. When fragments generated by the rasterization of a small triangle are

13. To this author's knowledge. But partially randomized execution sequences are now employed as a security measure, to thwart the attempts of computer "hackers."

textured, their 2D texel coordinates typically cluster into a small region in the texture image. Data locality will be greater if the 1D memory addresses of the texels that are accessed also cluster into a small band of locations.

Let (x, y) be the 2D integer coordinates of a texel in a square texture image of dimension w, and a_0 be the base address of the texture data in memory. Then an obvious mapping would be **raster order,** specified as

$$a = a_0 + x + w \cdot y. \tag{38.1}$$

Unfortunately Equation 38.1 results in a single tight cluster of memory addresses only if y is single-valued. If the patch of texels is not on a single scanline of the texture image, different values of y produce smaller clusters of addresses, themselves separated by intervals of texture dimension w. Because w can be large (e.g., 1,024 or even 4,096) the overall clustering is not tight at all.

Spatial locality can be greatly improved by replacing the raster-order mapping of Equation 38.1 with **tiled** mapping. The texture image is logically divided into smaller squares that tile the image, meaning that they cover the image with no gaps and no overlaps. Tiled mapping is hierarchical: first raster order among tiles, then raster order within the selected tile. Given a tile dimension of w_t, the mapping is specified as

$$a = a_0 + {w_t}^2 \left((x \div w_t) + \frac{w}{w_t} (y \div w_t) \right) + (x \oslash w_t) + w_t (y \oslash w_t), \tag{38.2}$$

where $\div$ indicates truncated integer division (e.g., $7 \div 4 = 1$) and $\oslash$ indicates modulo division yielding the remainder (e.g., $7 \oslash 4 = 3$). If the accessed texels fall within a single tile (i.e., among all the address mappings, only the last two terms of the equation differ) then spatial locality is improved by a factor of w/w_t, because the small clusters of memory addresses are separated by w_t (which multiplies y in the final expression in Equation 38.2) rather than w (which multiplies y in Equation 38.1). Decreasing the tile size increases the magnitude of the improvement: 8×8-texel tiling of a $2{,}048 \times 2{,}048$-texture image improves spatial locality by a factor of 256! But smaller tiles also increase the likelihood that the cluster of texels will straddle multiple tiles, which drives locality back down, potentially below its raster-mapped value, if vertically adjacent tiles are straddled and the 2D cluster of texels is small. Finding the best balance among such tradeoffs is central to the art of system design. One clever solution increases the depth of the hierarchy by implementing tiles within tiles, or even tiles within tiles within tiles. Of course, this approach runs into limits of complexity as the depth of the hierarchy is increased, introducing yet another tradeoff.

We've just seen how the implementation of a GPU, through tiled texel mapping, can improve spatial locality. GPU architectures—that is, their programming interfaces—can also be designed to allow improved locality of reference. For example, the OpenGL programming interface couples each texture image with its texture reconstruction-filtering mode, allowing the GPU driver to select image-tiling parameters based on details such as linear versus cubic filtering. The Direct3D interface allows a single texture image to be used with several texture interpolation modes. This choice gives programmers more flexibility (they can use a single texture image for multiple purposes that require different interpolations)

but constrains optimizations by the GPU driver (e.g., a compromise tiling of the texture image may be chosen).[14]

38.7.2 Cache Memory

We've discussed what data locality is, and seen some examples of how GPU architecture and implementation are designed to increase data locality. Now we'll see how data locality can be exploited by system designers to greatly improve the performance of computing systems.

Recall from Section 38.6.2 that latency and bandwidth, the two greatest concerns of a memory-system designer, are inversely related to memory capacity: Smaller memories have lower access latency and higher access bandwidth, that is, they are faster than larger memories. Because locality of reference makes it likely that, for a given short window of time, most memory accesses are to a small number of physical memory blocks, we can improve performance by storing these blocks in a smaller, faster memory. There are two ways to expose this local memory: explicitly and implicitly.

The explicit approach directly exposes local memory in the architecture, giving programmers complete control of its use. Address fields of explicit local memory (which is small) require fewer bits than addresses for the main memory system (which is large), so their use reduces the bit rate of the processor instruction stream, as well as benefitting from reduced latency and increased bandwidth to the specified data. Registers are an extreme form of explicit local memory: They require very small addresses, and they exhibit negligible latency and huge bandwidth. The **local memory** blocks in Figure 38.3 are a more typical form of explicit local memory. Because these memories are not visible in either the OpenGL or the Direct3D pipeline models, we defer discussion to Section 38.9.

Although registers are explicit in the Direct3D architecture, their allocation and use is managed by the shader compiler rather than by programmers. In principle, shader compilers could also manage and optimize the use of other local memory, but in practice this is left to human programmers. Programmer-specified management of local memory is powerful, but it is also complex, time-consuming, and error-prone. Thus, it is desirable to provide a form of local storage that is managed implicitly and automatically by low-level (typically hardware-implemented) mechanisms within the architecture. We refer to such local memory as **cache** (pronounced "cash") memory.

Cache memory intercepts all accesses to main memory. If the requested data item is already present in the cache, it is either returned with low latency (in the case of a read request) or modified in place (in the case of a write request). If the requested item is not present in the cache, some previously cached item is **evicted,** the requested item is read from main memory into the cache, and then either returned with *high* latency (in the case of a read request) or modified in place (in the case of a write request). All of this happens implicitly, without programmer intervention, so there is no opportunity for programmer error. But programmers can significantly influence performance by coding to maximize data locality, since accesses of main memory are so costly.

14. OpenGL versions 3.3 and later have adopted the Direct3D approach of decoupling texture images and interpolation modes.

Read requests that are fulfilled by cached data (cache **hits**) have dramatically lower latency than those fulfilled from main memory (cache **misses**). Taking main-memory latency as the benchmark, this disparity is desirable: If most memory requests hit, latency is dramatically reduced. But it is tempting instead to take the cache's hit latency as the benchmark, because this performance is achieved asymptotically as the cache-miss rate goes to zero. Unfortunately, the large disparity in latencies is undesirable from this viewpoint, because even a few misses dramatically increase average latency. For example, the average latency of a cache with a miss penalty of $100x$ is doubled by a miss rate of only 1%. In practice, only very large cache memories achieve average read latencies that approach this hit latency.

Cache memory is organized into equal-size units called **lines,** which are typically much larger than a single data item. Transfers between the processor and cache memory operate at the granularity of individual data items—a word is read from a line in the cache and returned to the processor, or a byte is written from the processor into the appropriate cache line. But transfers between cache and main memory operate at cache-line granularity—entire cache lines are either read from or written to main memory. Cache-line size is chosen so that these transfers make efficient use of main-memory bandwidth. For example, cache lines may be as large as the blocks in main memory, or at least a substantial fraction of this size. When a cache read miss forces a line to be loaded from main memory, spatial locality ensures that most if not all of the data items in that line will be accessed before the line is overwritten by another. And caches can be designed to transfer lines back to main memory infrequently (**write-back cache**) rather than immediately after the processor writes a data item to the cache (**write-through cache**), minimizing the main-memory bandwidth consumed by writing, and thereby maximizing the main-memory bandwidth available for reading.

From the standpoint of the processor, cache memory addresses both of the key concerns of main memory: Apparent memory latency is reduced, and apparent memory bandwidth is increased. If cache memory size could be made arbitrarily large, both apparent latency and apparent bandwidth could in principle be driven to the point of diminishing return (i.e., to the point where further improvement would not increase processor performance). In practice, cache size is limited to a small fraction of the size of main memory, after which cache performance slows to that of main memory. Because apparent latency increases quickly even for very low miss rates, GPU implementations are typically tuned to achieve performance that is unconstrained by memory bandwidth (assuming typical graphics loading) with caches that are far too small to ensure the required latency. The (otherwise unacceptable) apparent memory latency is hidden by multithreading, as described in Section 38.6.3, rather than by outsized cache memories.

It is still possible for shader programmers to get in trouble, though, by demanding more memory bandwidth than is available. For example, GPU texture interpolation performance is typically balanced assuming high data locality. If this assumption is disrupted—if, for example, texture sample addresses specify disjoint, widely separated clusters of texels—then an excessively large number of memory blocks may be transferred from main memory to cache memory, and shader performance can plummet. Undersampling a texture is one way to create this situation. Thus, texture aliasing not only destroys image quality, it can also destroy GPU performance! Dependent texture reads, meaning calls to `tex1D`, `tex2D`, or `tex3D`, with a parameter that is not directly derived from the

triangle-interpolated texture coordinates, can also disrupt or destroy locality. The resultant complications contributed to the delay of support for dependent texture lookup, which was introduced to GPUs years after shader programmability was first supported.

Thus far we have considered a single cache memory, but modern GPUs often have two levels of cache, and CPUs even more (three, sometimes four). By convention these are named L1 cache, L2 cache, … L*n* cache, counting outward from the processor toward main memory.[15] L1 cache has the least capacity, but also the least latency, and is optimized to interact well with the processor. L*n* cache has the greatest capacity and the highest latency, and is optimized to interact well with main memory. Worst-case latency may actually increase as levels of memory hierarchy are added, due to the summation of multiple miss penalties, but overall performance is improved.

In systems with multiple processor cores, caching is typically also parallelized. The GeForce 9800 GTX GPU, for example, implements a separate L1 cache for each pair of cores, and a separate L2 cache for each bank of memory (see Figure 38.4). Multiple L1 caches allow each to be tightly coupled with only two processor cores, reducing latency by improving locality (each cache is physically closer to its cores) and by reducing access conflicts (each cache receives requests from fewer cores). Pairing an L2 cache with each memory bank allows each cache to aggregate accesses that map to its portion of main memory. Explicit local memory is also parallelized—the GeForce 9800 GTX implements a separate local memory for each core.

Recall that a key advantage of implicit local memory is the simplicity and reliability of its programming model. Adding hierarchy does not compromise this model: Although a single physical memory location may now be cached at multiple levels of the memory hierarchy, memory requests from multiple cores "see" a consistent value because the requests are handled consistently. Parallel caches (such as multiple L1 caches) potentially break the model, however, because they are accessed and updated independently, so replications of a single physical memory location can become inconsistent. If this happens, the programmer's model of the memory system has become *incoherent,* and the likelihood of programming errors skyrockets.

Architects of parallel systems handle the cache coherence problem in one of three ways.

1. **Coherent memory:** A coherent view of memory is enforced by adding complexity to the memory hierarchy implementation. Cache-coherent protocols ensure that changes made to one data replica are broadcast or otherwise transferred to other replicas, either immediately or as required. This solution is expensive, both in implementation complexity and in the inevitable reduction in performance.
2. **Incoherent memory:** An incoherent view of memory is accepted—programmers must contend with the additional complexity this entails. This solution is frugal in system implementation, but it is expensive due to the likely reduction in programmer efficiency.

15. Abbreviations L1$, L2$, … L*n*$ are sometimes used informally, such as in figures.

3. **Constrained access:** A coherent view of memory is enforced by constraining how memory is accessed, rather than by adding complexity to the memory hierarchy implementation. This is the approach taken by GPUs as they are exposed by the Direct3D and OpenGL interfaces. Memory that is shared, such as texture images, is read-only, so no inconsistencies are possible. Memory that is written, such as framebuffer memory, is write-only from the viewpoint of the parallel cores—read-modify-write framebuffer operations, such as depth buffering and blending, are implemented by dedicated "pixel ops" circuits that operate directly on framebuffer physical memory (see Figure 38.4). So again, no inconsistency is possible.

Cache memory is a central concern in computer architecture, and we have only scratched the surface in this short section. Rich topics, such as eviction policy and set associativity, have not been covered at all. Interested readers are encouraged to peruse the list of suggested readings at the end of this chapter. Be warned, however, that most of the literature is written from the perspective of CPU architects, whose experience and consequent world views differ from those of GPU architects.

38.7.3 Divergence

We learned in Section 38.4 that GPUs such as the GeForce 9800 GTX implement a SPMD programming model with SIMD processing cores. The single-*program,* multiple-data programming model allows shaders to be written as though each was executed individually, greatly simplifying the programmer's job. The single-*instruction,* multiple-data implementation collects elements (e.g., vertices, primitives, or pixel fragments) into short vectors, which are executed in parallel by data paths that share a single instruction sequence unit. (The GeForce 9800 GTX has 16 cores, each with an effective vector length of 16 elements.)

The motivation for SIMD implementation is efficiency: Sharing a single instruction stream among multiple data paths allows more data paths to be implemented in the same silicon area and reduces instruction-fetch bandwidth per data path. For example, if a core's instruction sequence unit occupies the same silicon area as one data path, then a 16-wide true-parallel SIMD core occupies just over half (17/32) the area required by 16 SISD cores, almost doubling peak performance per unit silicon area. But this efficiency is achieved only when the elements assigned to a vector require the same sequence of instructions. When different sequences are required—that is, when the instruction sequences **diverge**—efficiency is lost.

When GPU shader programming was first exposed in OpenGL and Direct3D, shaders had no conditional branch instructions. Each instance of such a shader executed the same sequence of instructions, regardless of the element data being operated on, so divergence was limited to vectors that straddled a change made to a shader. Then, as now, GPU architectures encouraged operations on large blocks of data (e.g., Direct3D vertex buffers and OpenGL vertex arrays), during which no changes can be made to shaders. And GPU implementations typically packed SIMD vectors first-come-first-served, just as skiers are loaded onto lift chairs. So vectors that straddled changes in shaders were unusual, and divergence was not a significant problem.

Modern GPUs *do* support conditional branching in shaders, however, and its use by programmers does increase divergence. In the worst case, when each vector

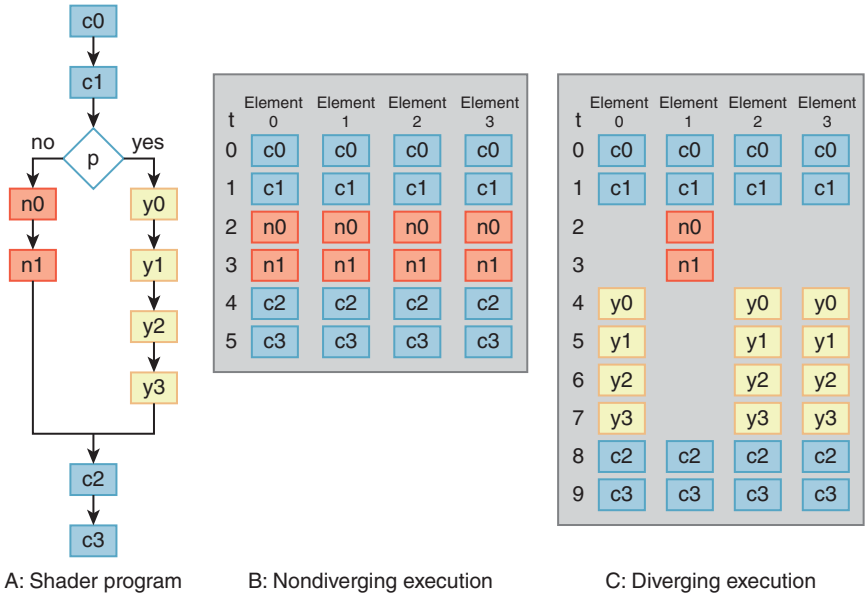


Figure 38.9: Diverging and nondiverging execution on a four-element predicated vector core. Each element executes the ten-operation shader A that branches on predicate p. In case B, all four elements take the no branch, there is no divergence, and only six execution steps are required. In case C, element one takes the no branch, but the other three elements take the yes branch. Predication handles this divergence by executing the no and yes operations separately, so all ten execution steps are required.

element executes a completely different program section (e.g., the shader is equivalent to a C++ switch statement with selection driven by unique element indices) divergence is complete and all parallelism is lost. More typically, elements share portions of code, which the *predicated* SIMD core executes in parallel, so parallelism is merely reduced.

Figure 38.9 illustrates such a typical situation, using a simplified four-element vector core. Shader A includes a single conditional branch that selects either the two-operation *no* path or the four-operation *yes* path. Another four operations, two ahead of the branch and two after, are common to both paths: They are executed regardless of which path is taken. In the first example, B, all four elements take the *no* branch, so there is no divergence. Predication handles this case by executing each common *no*-path operation in parallel across the four-wide vector. Because no elements require the *yes*-path operations, they are never executed and no cycles are lost to them. Thus, the entire shader executes in only six cycles.

In the second example, C, element one takes the *no* branch, but the other elements take the *yes* branch. Because both branches are taken, execution diverges. Predication handles this divergence by first executing each *no* operation on the single element that requires it, then executing each *yes* operation in parallel across the three elements that require it. Because both *no* and *yes* operations are executed, shader execution requires the full ten cycles to complete.

Obviously, divergence reduces the efficiency of computation on a predicated SIMD core. We quantify this loss by computing **utilization:** the ratio of the useful work that is done to the number of operation slots made available for that

work. Let n be the vector's length, i_{pred} be the number of predicated instruction steps required to execute the vector to completion, and i_{seq} be the total number of instruction steps that would be required to execute the element's shaders sequentially, one at a time, as though on a single processor. Then the utilization of this vector's execution (u_{vec}) is the ratio of useful work done (i_{seq}) to the number of slots available for that work ($n \cdot i_{pred}$):

$$u_{vec} = \frac{i_{seq}}{n \cdot i_{pred}}. \tag{38.3}$$

In example B, the nondiverging case,

$$u_{vec} = \frac{6+6+6+6}{4 \times 6} = 1.0, \tag{38.4}$$

which is the maximum possible value, indicating full utilization. In example C, the diverging case,

$$u_{vec} = \frac{8+6+8+8}{4 \times 10} = 0.75, \tag{38.5}$$

indicating partial utilization. Predication ensures that an operation is executed for at least one element during each cycle, so the worst possible utilization for an n-wide vector core is $1/n$. Minimum utilization is achieved in the switch-statement situation described above; it is approached asymptotically when a single element executes a path that is much longer than the paths executed by the other elements.

Utilization directly scales performance—the 0.75 utilization achieved in example C corresponds to 75% of peak performance, or 33% additional running time (when aggregated across many elements). Because poor utilization is the direct result of divergence, it is useful to understand the likelihood of divergence, perhaps as a first step to minimizing it.

Again consider a shader with a single conditional branch. Let p be the branch's probability of taking the *yes* path, and $1-p$ be its probability of taking the *no* path. Then, if p is evaluated independently for each element, that is, if evaluations of p had no locality, then divergence outcome probabilities for an n-wide vector core are:

$$\begin{array}{rl} p^n & \text{no divergence, all } \textit{yes} \text{ outcomes} \\ (1-p)^n & \text{no divergence, all } \textit{no} \text{ outcomes} \\ 1-\left(p^n+(1-p)^n\right) & \text{divergence, various utilizations.} \end{array} \tag{38.6}$$

Unless p is either very near to zero or very near to one, the probability of diverging increases rapidly as vector length n increases. For example, the probability of divergence with $p = 0.1$ is 34% for $n = 4$, but it increases to 81% for $n = 16$, 97% for $n = 32$, and 99.9% for $n = 64$. Even with $p = 0.01$, a seemingly low probability, divergence occurs almost half the time (47%) for a vector length of 64. These odds might dissuade GPU architects from implementing wide vector units if they were correct, but in general they are not.

In fact, evaluations of p are not independent—they tend to cluster into *yes* groups and *no* groups. Temporal locality predicts this: Clusters of repeated references suggest that the same code branch is executed repeatedly. The geometric nature of computer graphics often strengthens the effect. Consider the typical case of a predicate p that is *true* in shadow and *false* otherwise. Some triangles will be

partially shadowed, but unless tessellation is very coarse, most will be either fully lighted or fully in shadow. The stream of fragments resulting from the rasterization of these triangles will therefore have long clusters of same-valued predicates. And SIMD vectors loaded with these fragments will experience low divergence.

Just as GPU designers manipulate the texel-to-memory mapping to improve spatial locality during texture mapping (see Section 38.7.1), they also manipulate the sequence in which fragments are generated during rasterization. Rather than generating the fragment stream from a raster path, a more complex 2D path (akin to a **space-filling curve**) is often employed (see Figure 38.10). For triangles that project to a small region in the framebuffer this ordering makes little difference. But for larger triangles it substantially improves the 2D locality in the pixel coordinates of the generated fragments. This locality reduces SIMD divergence in the same way that rasterizing small triangles does. It has the added benefit of improving the spatial locality of texture coordinates, and therefore of the memory accesses required to perform texture mapping.

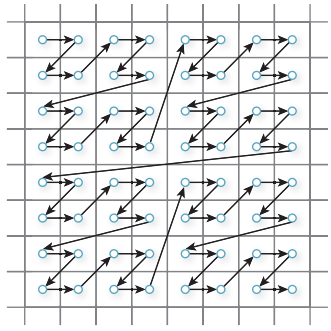


Figure 38.10: A portion of a rasterization path that exhibits high 2D spatial locality.

38.8 Organizational Alternatives

GPU architectures are specified at the relatively high levels of Direct3D and OpenGL, leaving implementors plenty of room for innovation. But not endless room—there exist attractive design alternatives that test and sometimes exceed this limit. Three such alternatives are considered in this chapter: deferred shading, binned rendering, and CPU/GPU-hybrid implementation. All have been implemented in earlier research or production systems, some are employed in current embedded or mobile systems, and all have desirable properties that make them likely candidates for future systems.

38.8.1 Deferred Shading

The goal of deferred shading is to shade only *visible* samples, thereby minimizing shading costs.[16] Because Direct3D and OpenGL determine visibility in the framebuffer, using the *z*-buffer algorithm, visibility is fully resolved only after the entire scene has been rendered. So shading must be deferred until this time, and rendering becomes a two-phase process: Render the entire scene to the framebuffer, and then compute shading for each pixel in the framebuffer. Furthermore, because shading requires access to various parameters that are interpolated during rasterization—such as surface normals, material parameters, and texture names—these values must be rendered into and stored in the framebuffer.

In addition to reducing the number of shading calculations, deferred shading also alters the *locality* of the shading calculations. Consider an implementation that executes shaders on screen-aligned tiles of pixels. Individual triangles are often split across tile boundaries, so per-tile deferred shading reduces the inherent locality of triangle calculations. But the set of triangles rendered to a tile have a locality in 3D space that can allow lighting to be optimized. Specifically, many lights in a highly lighted scene can be culled, because they are too far from the

16. Shading as it is treated here considers only light that reaches surface samples directly, not light that is reflected from other surfaces. This is a typical simplification in high-performance graphics rendering.

3D region corresponding to the tile. In scenes with thousands of lights the performance balance can tilt toward deferred shading.

Difficulties with deferred shading include the following.

- **Excess storage and bandwidth:** Burdening each pixel in the framebuffer with the information required for its shading is a significant expense in both storage and bandwidth. Indexing helps: Textures can be specified by reference; even better, the entire shader can be specified by reference. But parameters such as surface normals and interpolated texture coordinates are specific to each pixel, so they require storage by value. Variability in storage requirements complicates the situation, because maximum per-pixel storage requirement is not easily inferred at the start of rendering, and neither Direct3D nor OpenGL requires that it be specified.
- **Incompatibility with multi-sample anti-aliasing (MSAA):** Multi-sample anti-aliasing, the currently preferred approach to reducing edge artifacts in full-scene rendering, requires storage for multiple color and depth samples at each pixel. For example, multi-sample anti-aliasing with four samples per pixel increases framebuffer storage requirements by a factor of four. In combination with deferred shading, multi-sample anti-aliasing increases already burdensome framebuffer storage requirements by this same factor. Shading calculations are also increased by this factor, undoing the central optimization of multi-sample anti-aliasing, which is limiting shading calculations to one per pixel. The real possibility of a net *increase* in shading calculations, and the certainty of increased storage, make deferred shading incompatible with multi-sample anti-aliasing.[17]
- **No shader-specified visibility:** High-performance rendering sometimes approximates the visibility of complex geometry, such as foliage, with an alpha matte rendered as a texture. This optimization is inconsistent with deferred shading, whose goal is full determination of visibility *prior* to shading.

Against this seemingly bleak background, the story of deferred shading has a surprisingly happy ending. All modern GPUs, including the GeForce 9800 GTX, implement an optimization called **early *z*-cull.** An outline of the algorithm follows. As the frame is rendered, the GPU builds a hierarchical structure of z-values in dedicated local memory. Rasterized fragments are tested against this z-pyramid and are either culled (prior to shading) if they are not visible, or delivered to the framebuffer (and added to the z-pyramid) if they are visible. Little extra storage, and no main-memory bandwidth, are required, yet much of the performance gain of deferred shading is achieved, especially when programmers deliver scene data in an order that approximates front to back (i.e., rendering objects that are progressively farther from the viewpoint). While exact front-to-back ordering is a huge burden to programmers, approximating this ordering is often straightforward, and the penalty for partial failure is low (just a slight decrease in performance). Alternatively, application programmers sometimes choose to render the entire scene twice—first with shading *disabled* to create the full z-pyramid, and then again with shading *enabled* to shade exactly the visible fragments—to fully achieve

17. At the time of this writing, researchers are actively investigating deferred-shading algorithms that are compatible with multi-sample anti-aliasing.

deferred shading. Of course, early z-cull doesn't optimize computational locality for scenes with thousands of lights, so GPU architects will continue to study approaches to true deferred shading.

Inline Exercise 38.1: Suppose that the depth complexity of a pixel is n (i.e., a ray traced through the pixel would intersect n different surfaces in the scene, only the nearest being visible), and that these surfaces are processed in random order, but with early z-cull, that is, a fragment is shaded only if it's in front of all other fragments we've encountered so far. What's the expected number of fragments that are shaded during the course of rendering this pixel?

38.8.2 Binned Rendering

We usually define rasterization as the process that converts screen-coordinate geometric primitives directly to pixel fragments. But rasterization to larger screen areas, such as to $n \times n$-pixel tiles, is also possible. The GeForce 9800 GTX rasterizer is a case in point—it outputs 2×2 **quad fragments** to simplify texture-mapping calculations, as described in Section 38.6.1. Binned rendering splits rasterization into two phases: a first phase that outputs medium-size **tile fragments,** each corresponding to a (typically) 8×8-, 16×16-, or 32×32-pixel grid in screen coordinates, followed by a second phase that reduces each tile fragment to pixel fragments. Of course, tile fragments include information derived from the screen-coordinate primitive so that second-phase rasterization can produce the correct pixel fragments.

Binned rendering actually splits the *entire* rendering process into two phases, corresponding to the two phases of rasterization. During the first phase the scene is processed through tiled rasterization, and the resultant tile fragments are sorted into bins, one bin corresponding to each screen tile. Only after the first phase is completed (i.e., after tile fragments for the entire scene have been generated and sorted into bins) does the second phase begin. During the second phase each bin is processed individually to completion, yielding an $n \times n$ tile of pixels that is deposited in the framebuffer.

Binned rendering has several attractive properties.

- **Local memory:** The absolute guarantee of framebuffer data coherence—only pixels within the tile are accessed—allows pixels to be processed in local memory, rather than cached from main memory. Both power and main-memory cycles are conserved, making binned rendering an attractive solution for mobile devices.
- **Full-scene anti-aliasing:** Recall that multi-sample anti-aliasing requires storage for multiple color and depth samples at each pixel. As quality is improved by increasing the sample count, both storage and bandwidth become prohibitively expensive when rendering is to the entire framebuffer, but they remain economical when rendering is limited to a small tile of pixels. Even more advanced rendering algorithms, such as order-independent rendering of transparent surfaces, can be supported with clever use of local memory.
- **Deferred shading:** Limiting rendering to a small tile of pixels addresses the key limitations of deferred shading: its requirements of excessive

memory storage and bandwidth, and its incompatibility with multi-sample anti-aliasing.

The advantages of binned rendering are compelling, yet no current PC-class GPU implements it.[18] The fundamental reason is that binned rendering differs too much from the pipeline Direct3D and OpenGL architectures—we say that the **abstraction distance** is too large. Typically, excess abstraction distance results in products with confounding performance characteristics (operations expected to be fast are slow; those expected to be slow are fast) or subtle deviations from specified operation. Practical problems that are encountered include the following.

- **Excess latency:** Previous binned rendering systems, such as the PixelPlanes 5 system developed at the University of North Carolina, Chapel Hill, have added a full frame time of latency.
- **Poor multipass operation:** Direct3D and OpenGL encourage advanced multipass rendering techniques that, in a binned implementation, require multiple two-pass operations per final frame. An example is rendering reflection from a curved surface by 1) rendering the scene that will be visible in the reflection, 2) loading this image as a texture, and 3) rendering the curved surface with the appropriately warped texture image. Some binned rendering systems failed to support such operations; others supported them, but performed poorly.
- **Unbounded memory requirements:** While binned rendering bounds *pixel storage* to that required for a single tile, the memory required by the bins themselves grows with scene complexity. Neither OpenGL nor Direct3D has scene-complexity limits, so a fully confirming implementation requires infinite memory (an obvious impossibility) or must introduce complexity to handle cases for which finite bin storage is inadequate.

These complications have been sufficient to keep binned rendering out of mainstream PC GPUs. But recent implementation trends, in particular the use of a time-shared single compute engine to implement all pipeline shading stages, may overcome some of the difficulties.

38.8.3 Larrabee: A CPU/GPU Hybrid

In 2008, Intel published a technical paper [SCS+08] describing a forthcoming GPU, code-named Larrabee. While this product never shipped, for reasons we will discuss shortly, it was a serious attempt to combine ideas from Intel CPUs and competitive GPUs into a single compelling product. In this short section we will analyze this hybrid GPU, first by comparing its implementation and architecture with those of the NVIDIA GeForce 9800 GTX GPU and the Intel Core 2 Extreme QX9770 CPU, and then by considering the resultant strengths and weaknesses. We begin with implementation, referring to the Larrabee block diagram in Figure 38.11, which is drawn to be consistent (where possible) with the NVIDIA GeForce 9800 GTX block diagram in Figure 38.4.

In several key respects the Larrabee implementation more closely resembles the implementation of the NVIDIA GeForce 9800 GTX *GPU* than it does that of the Intel Core 2 Extreme QX9770 *CPU*.

18. Inexpensive Intel GPUs have implemented binned rendering in the past, but none do now.

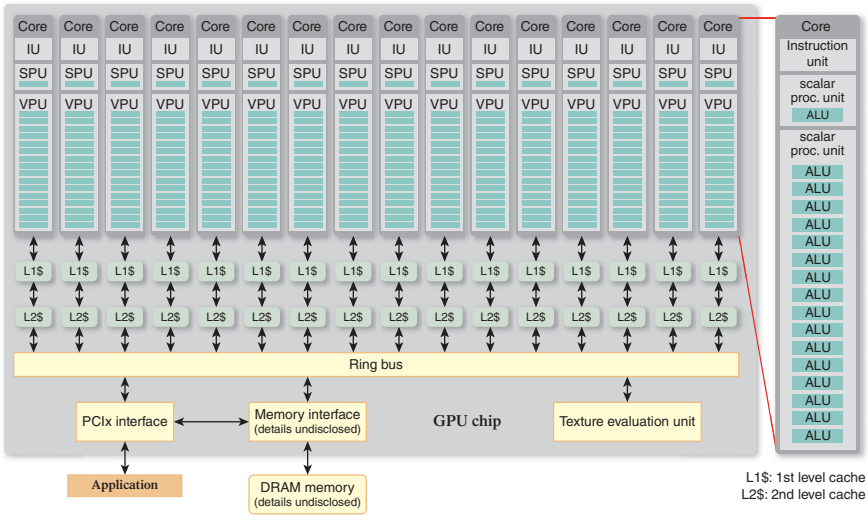


Figure 38.11: Intel Larrabee GPU block diagram. Compare this with the NVIDIA *GeForce 9800 GTX GPU block diagram in Figure 38.4.*

- **Many cores:** The Larrabee and GeForce 9800 GTX block diagrams are both dominated by multiple processing cores, as are the implementations of the GPUs themselves. While the number of cores in the Larrabee GPU was never announced, it is known to be at least 16, matching or exceeding the GeForce 9800 GTX, and greatly exceeding the four cores of the Core 2 Extreme QX9770 CPU. Furthermore, like the GeForce 9800 GTX cores, the Larrabee cores are designed to maximize performance per unit of silicon area (and therefore overall GPU performance), whereas the Core 2 Extreme QX9770 CPU cores are designed to maximize performance per core (at the expense of overall performance).
- **Wide vectors:** The Larrabee and GeForce 9800 GTX cores both include wide SIMD units: $n = 8$ for the GeForce 9800 GTX (virtualized to $n = 16$) and $n = 16$ for Larrabee. Both GPU cores provide hardware support for predication, and both provide separate address circuits for each element, allowing data to be efficiently **gathered** into a vector, then **scattered** back to memory. Conversely, the Core 2 Extreme QX9770 CPU implements narrow vectors ($n = 4$) with no support for efficient scatter/gather or predication.
- **Texture evaluation:** Both GPUs support texture evaluation with dedicated, fixed-function units; the CPU provides no support.

Balanced against these important similarities, the Larrabee and GeForce 9800 GTX implementations also differ in ways that betray Larrabee's CPU heritage. These include the following.

- **Specialized, fixed-function hardware:** Except for its texture evaluation unit, Larrabee omits support for the many GeForce 9800 GTX fixed-function units, including vertex generation, primitive generation, fragment generation (rasterization), work queueing and distribution, and pixel operations.

- **Latency hiding:** Like the Core 2 Extreme QX9770 CPU, Larrabee relies more heavily on its large, hierarchical caches to hide memory latency.[19] Each Larrabee core supports only four threads, against the 48 threads that implement the GeForce 9800 GTX's block multithreading.[20]
- **Cache coherence:** Larrabee's cache hierarchy maintains full coherence, just as the Core 2 Extreme QX9770 CPU's caches do (though at somewhat higher cost, since Larrabee has at least four times as many caches at each level of the hierarchy). Cache coherence is not implemented by the GeForce 9800 GTX.

Larrabee's strong implementation parallels to the GeForce 9800 GTX GPU are not matched in terms of architecture, where parallels to the Core 2 Extreme QX9770 CPU dominate.

- **ISA:** The Larrabee and Core 2 Extreme QX9770 cores share Intel's IA-32 instruction set architecture (ISA), though Larrabee's ISA is augmented with vector instructions. While the GeForce 9800 GTX cores are programmable, their ISA is hidden beneath the OpenGL and Direct3D shader languages. Both Larrabee and Core 2 Extreme QX9770 can be **programmed** to support OpenGL and Direct3D, and in fact Intel engineers ported both APIs to Larrabee (using binned rendering!), but these APIs are not architectural. Indeed, there is no evidence of a pipeline model in either CPU-like architecture.
- **SPMD:** Like the Core 2 Extreme QX9770, Larrabee cores are exposed in the ISA as scalar with vector instructions. Each core has 17 ALUs: one scalar and 16 vector. Programmers use a combination of scalar and vector instructions to explicitly manage control flow (a single program counter for the entire core) and vector operations on data. Predication is also explicit—programmers manipulate a mask to control which vector elements are committed. The GeForce 9800 GTX hides all this complexity behind its SPMD architecture, allowing programmers to treat each vector element as an individual execution unit with full flow control (its own program counter) and data execution capability.

Larrabee's CPU-like architecture gives it important advantages over the GeForce 9800 GTX's SPMD architecture.

- **Flexibility:** Larrabee executes arbitrary graphics algorithms with equal efficiency, because it isn't optimized for a particular algorithm, as the GeForce 9800 GTX is for the "graphics pipeline."
- **Generality:** Larrabee executes arbitrary *non*graphics algorithms efficiently too.
- **Capability:** Like the Core 2 Extreme QX9770, Larrabee can run system-level programs up to and including operating systems. The GeForce 9800 GTX cannot.

19. Larrabee's cache sizes are 64 Kbytes $L1 and 256 Kbytes $L2. While NVIDIA does not publish figures for the GeForce 9800 GTX, they are almost certainly much lower.
20. A technique called **software fibers**—explicitly coding a multithread instruction sequence into a single thread—is also used to hide latency. It lacks the GPU capability of block scheduling, however, and is therefore less effective.

But Larrabee's flexibility, generality, and capability come at a cost: It is substantially more difficult to program than traditional GPUs such as the GeForce 9800 GTX. And, even with the heroic programming efforts of Intel's experts, the resultant OpenGL and Direct3D implementations performed significantly less well than competitive, traditional GPUs. Larrabee's more general architecture (ISA versus GeForce 9800 GTX's SPMD) contributes to this situation, of course, but the relative lack of graphics specialization in Larrabee's implementation is the root cause. While all graphics algorithms *can* be run on general-purpose vector cores, for some algorithms this is a very inefficient approach. Traditional GPUs such as the GeForce 9800 GTX are optimized to account for this: Independent, per-element operations with rich, application-programmable semantics such as those on vertices, primitives, and fragments are implemented on the data-parallel vector cores, while pipeline-specific algorithms with inherently rigid semantics such as rasterization (fragment generation) are implemented with specialized, fixed-function units.

Fixed-function units improve efficiency in several ways.

- **Efficient parallelization:** Specialized hardware can efficiently parallelize algorithms that are not inherently data-parallel.
- **Correct provisioning:** Algorithmic parameters such as numeric representation and precision can be optimized when the algorithm is implemented in task-dedicated hardware. Eight-bit integer multiplication, for example, requires less than one-tenth the hardware of 32-bit floating-point multiplication.
- **Sequence optimization:** When an algorithm is cast into dedicated hardware, each "step" is implemented with exactly the required capability (e.g., addition of two values) rather than effectively consuming the full capability of a core's ALU (addition, subtraction, multiplication, division, etc.). Furthermore, the sequence of steps is managed with dedicated hardware (e.g., a simple finite-state machine) rather than expressed as a program running on a core's instruction unit (whose stored-program model consumes expensive memory bandwidth and cache hierarchy).

All together, these advantages can yield impressive savings. For example, all modern GPUs include specialized hardware for decoding video streams.[21] While video decoding can be implemented using the data-parallel cores, it reportedly consumes 1/100th the power when run in a purpose-designed unit, allowing laptop computers to display movies without quickly draining their batteries. Similar savings ratios may be achieved by fixed-function implementations of graphics pipeline stages.

Larrabee's designers were not unaware of these advantages—they chose, for example, to implement texture evaluation with a purpose-built, fixed-function unit. But overall they shifted the traditional GPU implementation balance from specialization toward generalization, trading the resultant loss of performance on existing applications (OpenGL and Direct3D) for the opportunity to achieve improved performance in new areas, such as alternative graphics pipelines, and nongraphical algorithms. In a competitive market, performing better on new applications is a valuable differentiation, but performing well on existing applications is

21. This hardware is not otherwise discussed in this chapter.

crucial. Faced with introducing an uncompetitive GPU, Intel chose instead to cancel Larrabee.

38.9 GPUs as Compute Engines

As we've now seen, modern GPUs such as the GeForce 9800 GTX utilize massive parallelism, exposed as a pipeline of fixed-function and application-programmable stages, to apply hundreds of GFLOPS to the rendering of 3D graphics. Because the peak performance of GPUs is so much higher than that of CPUs (see Section 38.2), and because the SPMD architecture of GPU-programmable stages makes exploiting that performance straightforward (see Section 38.5), programmers are highly motivated to speed up their nongraphical programs by porting them from CPUs to GPUs. We conclude this chapter with a short discussion of these efforts.

Creative programmers have probably been porting nongraphical algorithms to GPUs since they came into existence. Under the rubric of **GPGPU** (general-purpose computing on GPUs), these efforts became an important trend in the late '90s. The primary enabler of this trend was the availability of programmable shaders in ubiquitous, single-chip GPUs such as NVIDIA's predecessors to the GeForce 9800 GTX.

Algorithms with substantial data parallelism (see Section 38.4) were ported to GPUs by implementing their **kernels** as shaders. The kernel of an image-processing filter, for example, computes the value of a single output pixel as the weighted sum of the values of nearby pixels. To run the shaders (i.e., to execute the algorithm) initial data was loaded as a 2D texture and a 2D rectangle fitted to the texture was rendered, causing the results to be deposited into the framebuffer. Over time, researchers identified data-parallel representations for algorithms, such as sorting, that aren't typically thought of as data-parallel. This allowed GPGPU to apply to a broader range of problems.

As GPGPU became more prevalent, new architectures were developed to better expose the general-purpose computing capabilities of GPUs. Examples include OpenCL, Microsoft's Direct Compute, and NVIDIA's CUDA. These architectures all maintain the SPMD programming model (i.e., the shaders) of the traditional pipeline architectures, implemented using the same multithreaded SIMD cores. All, however, dispense with the graphics pipeline and much of its fixed-function implementation, exposing instead a single compute stage. Compute-appropriate mechanisms, such as execution commands (it is no longer necessary to rasterize a rectangle to execute the shader-implemented kernels) and explicit local memory (as described in Section 38.7.2), are also added. The general-purpose computing architectures are alternatives to, not replacements for, OpenGL and Direct3D. Some in fact support interoperation, allowing a single GPU to compute *and* display data without transferring intermediate data from GPU to CPU and back.

GPGPU has come a long way during its decade of development. Today the fastest supercomputers in the world use GPUs as their primary computing engines, as do applications ranging from hedge-fund management to quantum physics. GPUs will never *replace* CPUs, but it seems increasingly likely that a new computing architecture, derived from GPU and CPU technology and perhaps resembling Intel's Larrabee prototype, will define the future of computer architecture.

38.10 Discussion and Further Reading

The design of graphics hardware is a special, but important, case of computer architecture design. Anyone interested in studying the subject more closely should first read the classic book by Hennessy and Patterson [HP96], where tradeoffs of the kind discussed in this chapter are covered in great detail.

Even if you don't want to design the next great GPU, you should be familiar with some aspects of architecture so that you can better understand how they influence your use of the GPU you've got. A great starting point is Ulrich Drepper's "What Every Programmer Should Know About Memory" [Dre07].

For years, computers have improved—processor cores are faster, bandwidth is greater, networks are faster, memory is larger and less expensive—and many of these improvements have followed exponential patterns, the classic being Moore's Law. But the observation that everything is improving exponentially masks the critical point that the constants in the exponents are different: GPUs have improved faster than CPUs, for instance. Even within a single device like a GPU, differing constants shift the landscape. If you want to understand how GPUs will change in the future, these differing constants (and the eventual leveling off from exponential growth) will be important predictors.

38.11 Exercises

Exercise 38.1: Recall that annual growth at rate r for y years results in $t = r^y$ compound growth (see Section 38.2). For example, $1.5x$ annual growth for five years results in $7.59 = 1.5^5$ compound growth. Derive the formula that gives r as a function of y and t. Using this formula, compute the annual growth rates for ten-year compound growths of $10x$, $100x$, and $1{,}000x$.

Exercise 38.2: Accurate performance measurement is a great way to understand the implementation of an architecture, and it's a prerequisite to tuning your own graphics programs. Using either Direct3D or OpenGL, write a program that initializes graphics state, performs a simple rendering task repeatedly for a specified number of times, and then returns the time required per iteration. Use this program to measure the rate that triangles of various sizes are rendered. Is there a size below which there is no significant change in performance? What might explain this?

Here are a few tips for accurate performance measurement.

- Neither Direct3D nor OpenGL provides timers, so you'll have to learn your operating system's commands for this.
- To get accurate timings you must flush the GPU before starting the timer and before stopping it. OpenGL provides the `glFlush()` command for this purpose. Why is this important?
- Don't swap buffers during a timing test.
- Don't run other applications during a timing test.
- Run your test repeatedly to test for consistency.

Think about other things that might affect your measurements.

Exercise 38.3: Look for performance dependencies on data coherence, using the framework developed in the preceding exercise. Using linear image

interpolation, map a $1{,}024 \times 1{,}024$-texture image to a 256×256 triangle. Compare the performance of 1-to-1, 2-to-1, and 4-to-1 texel-to-pixel mappings. If there is no performance difference, increase the demand for memory bandwidth by utilizing multiple textures until a difference is found. Make an estimate of peak memory bandwidth based on your results. If possible, compare this value to the one that is advertised for the GPU you are using.

Exercise 38.4: In Section 38.8.2 it is suggested that the recent trend toward implementing all shading pipeline stages with a single, time-shared compute engine may overcome some difficulties. Which difficulty in particular might this trend overcome? How is the situation improved?

Exercise 38.5: One difficulty with nonbinned deferred shading is the large amount of memory required to store interpolated parameters such as normals and texture coordinates in the framebuffer (see Section 38.8.1). Could these large, per-pixel data blocks be replaced with references, as was described for texture images and shaders? Propose a solution. Compare its memory requirements to per-pixel storage, making reasonable assumptions about parameters such as framebuffer dimensions, numeric representation, triangles per frame, and average projected triangle size.

List of Principles

Here we collect the principles that we explicitly espoused throughout the book. Like koans, these are perhaps best used to reflect on what you have learned, as new apertures through which to view those topics, and as a concise way of discussing the patterns that underlie computer graphics. Simply studying them outside of their context would not be effective.

We view this book as a series of case studies of concrete techniques—the *practice* that shows how to apply the principles to image synthesis and other problems. Our hope is that after reading many of the chapters you will agree that there is indeed a compact set of ideas and many ways of applying them.

As computer graphics scientists and engineers, we explore a vast sea of knowledge into which mathematics, engineering, physics, biology, psychology, and art flow. These principles chart the many courses of "computer graphics" that allow us to appreciate those disciplines and make forward progress in our own. For example, dynamics simulation of a gear system and estimating light transport are two of the many destinations presented in the field, but you can reach them with the same mathematical vehicles by following parallel courses of numerical integration techniques.

Computer graphics practice vertically integrates techniques from concrete, low-level engineering to abstract, high-level mathematics. A rendering engineer at a game company must not only understand mathematics and physics, but also must have reasonable computer organization skills to understand a graphics processor, cache, and bandwidth. That same engineer must also follow good software engineering methodology to work in a team of other programmers and artists, and nearly every day relies on calculus, geometry, optics, and color theory. Our principles vary in scope accordingly, from implementation strategies to modes of thought.

Know Your Problem principle: Know what problem you are solving.

Approximate the Solution principle: Approximate the solution, not the problem.

Wise Modeling principle: When modeling a phenomenon, understand the phenomenon you're modeling and your goal in modeling it, *then* choose a rich-enough abstraction, and *then* choose adequate representations to capture

your abstraction within the bounds of your resources, and finally, *test* to verify that your abstraction was appropriate.

Visual System Impact principle: Consider the impact of the human visual system on your problem and its models.

Coordinate-System/Basis principle: Always choose a coordinate system or basis in which your work is most convenient, and use transformations to relate different coordinate systems or bases.

First Pixel principle: The first pixel is the hardest.

Visual Debugging principle: Use visual displays to help you debug and understand your graphics programs.

Hierarchical Modeling principle: Whenever possible, construct models hierarchically. Try to make the modeling hierarchy correspond to a functional hierarchy for ease of animation.

Implementation principle: If you understand a mathematical process well enough, you can write a program that executes it.

Parametric/Implicit Duality principle: There's a duality between parametric and implicit forms for shapes: In general, it's easy to find an intersection between shapes where one's described implicitly and the other parametrically, and harder when either both are implicit or both parametric.

Tilting principle: If T' is an oriented triangle in plane P' with normal $\mathbf{n}'$, and T is its projection to plane P, the projection being along the unit normal $\mathbf{n}$ to P, then the signed area of T is $\mathbf{n}' \cdot \mathbf{n}$ times the signed area of T'.

Division of Modeling principle: Separate the mathematical and/or physical model of a phenomenon from the numerical model used to represent it.

Meaning principle: For every number that appears in a graphics program, you need to know the semantics, the **meaning,** of that number.

Transformation Uniqueness principle: For each class of transformations—linear, affine, projective—and any corresponding coordinate frame, and any set of corresponding target elements, there's a unique transformation mapping the frame elements to the correponding elements in the target frame. If the target elements themselves constitute a frame, then the transformation is invertible.

High-Level Design principle: Start from the broadest possible view. Elements of a graphics system don't separate as cleanly as we might like; you can't design the ideal representation for an emitter without considering its impact on light transport. Investing time at the high level lets us avoid the drawbacks of committing, even if it defers gratification.

Noncommutativity principle: The order of operations often matters in graphics. Swapping the order of operations can introduce both efficiencies in computations and errors in results. You should be sure that you know when you're doing so.

API principle: Design APIs from the perspective of the programmer who will use them, not that of the programmer who will implement them or from the mathematical notation used in their derivation. For example, a single BSDF $f(\omega_i, \omega_o)$ mapped to a function API `Color3 bsdf(Vector3 wi, Vector3 wo)` is easy to implement but hard to use in a real renderer.

Early Optimization principle: It's worth optimizing early if it makes the difference between an interactive program and one that takes several minutes to execute. Shortening the debugging cycle and supporting interactive testing are worth the extra effort.

Level of Detail principle: Level of detail is important for both efficiency *and* correctness.

Average Height principle: The average height of a point on the upper hemisphere of the unit sphere is $\frac{1}{2}$. Thus, for any unit vector $\mathbf{n}$, the integral

$$\int_{\{\omega \in \mathbf{S}^2 : \omega \cdot \mathbf{n} \geq 0\}} \omega \cdot \mathbf{n} \, d\omega = \pi.$$

Structure principle: Treat surprising structural symmetries and asymmetries as both clues about underlying structure and as warnings to check the robustness of any plan. For example, if otherwise similar elements differ by orders of magnitude or demand different parameters, as is the case for the fundamental forces of nature, something interesting is going on that can either lead to insight if followed, or bite you if ignored.

Culling principle: It is often efficient to approach a problem with one or more fast and conservative solutions that narrow the space by culling obviously incorrect values, and a slow but exact solution that then needs only to consider the fewer remaining possibilities.

Design Tradeoff principle: The art of architecture design includes identifying conflicts between the interests of implementors and users, and making the best tradeoffs.

Memory principle: The primary challenge of memory is coping with access latency and limited bandwidth. Capacity is a secondary concern.

This page intentionally left blank

Bibliography

[ABB64] G. M. Amdahl, G. A. Blaauw, and Frederick P. Brooks. Architecture of the IBM System/360. In *IBM Journal of Research and Development*, pages 87–101, April 1964.

[ABM88] Brian Apgar, Bret Bersack, and Abraham Mammen. A display system for the Stellar graphics supercomputer model GS1000. *SIGGRAPH Comput. Graph.*, 22(4):255–262, 1988.

[Ado08] Adobe. Document management—Portable document format—Part 1: PDF 1.7, 2008.

[AdSP07] Yeuhi Abe, Marco da Silva, and Jovan Popović. Multiobjective control with frictional contacts. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '07, pages 249–258, 2007. Eurographics Association.

[AFO05] Okan Arikan, David A. Forsyth, and James F. O'Brien. Pushing people around. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '05, pages 59–66, New York, NY, USA, 2005. ACM.

[AGCA08] Prekshu Ajmera, Rhushabh Goradia, Sharat Chandran, and Srinivas Aluru. Fast, parallel, GPU-based construction of space filling curves and octrees. *Poster at the 2008 Symposium on Interactive 3D Graphics and Games*, I3D '08, page 10:1, New York, NY, USA, 2008. ACM.

[AGW86] Phil Amburn, Eric Grant, and Turner Whitted. Managing geometric complexity with enhanced procedural models. *SIGGRAPH Comput. Graph.*, 20(4):189–195, August 1986.

[Air90] John Milligan Airey. *Increasing update rates in the building walkthrough system with automatic model-space subdivision and potentially visible set calculations*. PhD thesis, The University of North Carolina at Chapel Hill, USA, 1990.

[AK87] James Arvo and David Kirk. Fast ray tracing by ray classification. *SIGGRAPH Comput. Graph.*, 21(4):55–64, August 1987.

[Ake93] Kurt Akeley. RealityEngine graphics. In *SIGGRAPH '93: Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, pages 109–116, New York, NY, USA, 1993. ACM.

[Ale02] Marc Alexa. Linear combination of transformations. *ACM Trans. Graph.*, 21(3):380–387, July 2002.

[AMD12] AMD. RenderMonkey Toolsuite. `http://developer.amd.com/resources/archive/archived-tools/gpu-tools-archive/rendermonkey-toolsuite/`, 2012.

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering (3rd Edition)*. A K Peters, Ltd., Natick, MA, USA, 2008.

[AMMH07] Tomas Akenine-Möller, Jacob Munkberg, and Jon Hasselgren. Stochastic rasterization using time-continuous triangles. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/Eurographics Symposium on Graphics Hardware*, pages 7–16, 2007. Eurographics Association.

[App67] Arthur Appel. The Notion of Quantitative Invisibility and the Machine Rendering of Solids. *Proc. ACM Natl. Mtg.*, page 387, 1967.

[App68] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.

[ARS79] Arthur Appel, F. James Rohlf, and Arthur J. Stein. The haloed line effect for hidden line elimination. In *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '79, pages 151–157, New York, NY, USA, 1979. ACM.

[Arv95] James Richard Arvo. *Analytic methods for simulated light transport*. PhD thesis, Yale University, New Haven, CT, USA, 1995.

[AS95] David Adalsteinsson and James A. Sethian. A fast level set method for propagating interfaces. *J. Comput. Phys.*, 118(2):269–277, May 1995.

[AS06] Kurt Akeley and Jonathan Su. Minimum triangle separation for correct z-buffer occlusion. In *Proceedings of the 21st ACM SIGGRAPH/Eurographics Symposium on Graphics Hardware*, pages 27–30, New York, NY, USA, 2006. ACM.

[Ash01] Michael Ashikhmin. Synthesizing natural textures. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, I3D '01, pages 217–226, New York, NY, USA, 2001. ACM.

[ASP07] Paul Asente, Mike Schuster, and Teri Pettit. Dynamic planar map illustration. In *ACM Trans. Graph.*, 26(3), 30:1–30:10, July 2007.

[AVF04] C. Andujar, P. Vazquez, and M. Fairén. Way-Finder: Guided tours through complex walkthrough models. *Computer Graphics Forum*, 23(3):499–508, 2004.

[AW87] J. Amanatides and A. Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Proc. Eurographics '87, Amsterdam, The Netherlands*, pages 1–10, August 1987.

[AZ97] Johnny Accot and Shumin Zhai. Beyond Fitts' law: Models for trajectory-based HCI tasks. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, CHI '97, pages 295–302, New York, NY, USA, 1997. ACM.

[Bac] Michael Bach. Visual Phenomena and Optical Illusions. `http://www.michaelbach.de/ot/`.

[Ban65] Thomas F. Banchoff. Tightly Embedded Two-Dimensional Polyhedral Manifolds. *American Journal of Mathematics*, 87:465–472, 1965.

[Ban74] Thomas F. Banchoff. Triple Points and Surgery of Immersed Surfaces. *Proceedings of the American Mathematical Society*, 45(3): 407–413, December 1974.

[Bar92] R. Barzel. *Physically-Based Modeling for Computer Graphics: A Structured Approach*. Academic Press, 1992.

[Bau72] Bruce G. Baumgart. Winged edge polyhedron representation. Technical report, Stanford University, Stanford, CA, USA, 1972.

[BBD+99] C. Betrisey, J. F. Blinn, B. Dresevic, B. Hill, G. Hitchcock, B. Keely, D. P. Mitchell, J. C. Platt, and T. Whitted. Displaced filtering for patterned displays. In *Society for Information Display 1999 Digest of Technical Papers*, pages 296–299, 1999.

[BBO+09] Bernd Bickel, Moritz Bächer, Miguel A. Otaduy, Wojciech Matusik, Hanspeter Pfister, and Markus Gross. Capture and modeling of non-linear heterogeneous soft tissue. *ACM Trans. Graph.*, 28(3):89:1–89:9, July 2009.

[BCfPRD61] H. P. Bishop, M. N. Crook, Tufts Inst. for Psychological Research, and United States Wright Air Development Division. *Absolute Identification of Color for Targets Presented Against White and Colored Backgrounds*. WADD technical report. Behavioral Sciences Laboratory, Aerospace Medical Laboratory, Wright Air Development Division, Air Research and Development Command, United States Air Force, 1961.

[BCL+07] Louis Bavoil, Steven P. Callahan, Aaron Lefohn, João L. D. Comba, and Cláudio T. Silva. Multi-fragment effects on the GPU using the k-buffer. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 97–104, New York, NY, USA, 2007. ACM.

[BCWG09] Mirela Ben-Chen, Ofir Weber, and Craig Gotsman. Spatial deformation transfer. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '09, pages 67–74, New York, NY, USA, 2009. ACM.

[BD80] J. K. Bowmaker and H. J. Dartnall. Visual pigments of rods and cones in a human retina. *The Journal of Physiology*, 298:501–511, January 1980.

[BD02] David Benson and Joel Davis. Octree textures. *ACM Trans. Graph.*, 21(3):785–790, July 2002.

[BEG98] B. Wyvill, E. Galin, and A. Guy. The Blob Tree, Warping, Blending and Boolean Operations in an Implicit Surface Modeling System. In *Implicit Surfaces '98*, June 1998.

[BF01] Samuel R. Buss and Jay P. Fillmore. Spherical averages and applications to spherical splines and interpolation. *ACM Trans. Graph.*, 20(2):95–126, April 2001.

[BHW96] Ronen Barzel, John F. Hughes, and Daniel N. Wood. Plausible motion simulation for computer graphics animation. In *Proceedings of the Eurographics Workshop on Computer Animation and Simulation '96*, pages 183–197, New York, NY, USA, 1996. Springer-Verlag New York, Inc.

[Bir61] Faber Birren. *Creative color*. Reinhold Pub. Corp., 1961.

[BJ97] Gerrit A. Blaauw and Frederick P. Brooks Jr. *Computer Architecture: Concepts and Evolution*. Addison-Wesley, 1997.

[BJ01] Y. Y. Boykov and M. P. Jolly. Interactive graph cuts for optimal boundary & region segmentation of objects in N-D images. In *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on*, pages 105–112, 2001.

[BKLP04] D. Bowman, E. Kruijff, J. LaViola, and I Poupyrev. *3D User Interfaces: Theory and Practice*. Addison-Wesley, 2004.

[BL04] William V. Baxter and Ming C. Lin. A Versatile Interactive 3D Brush Model. In *Proceedings of the 12th Pacific Conference on Computer Graphics and Applications.* PG '04, pages 319–328, Washington, DC, USA, 2004. IEEE Computer Society.

[BLB+08] Bernd Bickel, Manuel Lang, Mario Botsch, Miguel A. Otaduy, and Markus Gross. Pose-space animation and transfer of facial details. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '08, pages 57–66, 2008. Eurographics Association.

[Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, July 1977.

[Bli78] James F. Blinn. Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph.*, 12(3):286–292, August 1978.

[Bli82a] James F. Blinn. A Generalization of Algebraic Surface Drawing. *ACM Trans. Graph.*, 1:235–256, July 1982.

[Bli82b] James F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. *SIGGRAPH Comput. Graph.*, 16(3):21–29, July 1982.

[Bli93] James F. Blinn. A Trip Down the Graphics Pipeline: The Homogeneous Perspective Transform. *IEEE Comput. Graph. Appl.*, 13(3):75–80, 1993.

[Bly06] David Blythe. The Direct3D 10 System. *ACM Trans. Graph.*, 25(3), 2006.

[BN76] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, October 1976.

[BOT79] N. I. Badler, J. O'Rourke, and H. Toltzis. A spherical representation of a human body for visualizing movement. *Proceedings of the IEEE*, 67(10):1397–1403, October 1979.

[Bou70] W. Jack Bouknight. A procedure for generation of three-dimensional half-toned computer graphics presentations. *Commun. ACM*, 13(9):527–536, September 1970.

[Boy79] R. M. Boynton. *Human color vision*. Holt, Rinehart and Winston, 1979.

[Bra99] Ronald Bracewell. *The Fourier Transform and Its Applications (3rd Edition)*. McGraw-Hill Science/Engineering/Math, Columbus, OH, USA, 1999.

[Bre65] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Syst. J.*, 4(1):25–30, 1965.

[Bri07] Robert Bridson. Fast Poisson disk sampling in arbitrary dimensions. In *ACM SIGGRAPH 2007 sketches*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[BS81] F. W. Billmeyer and M. Saltzman. *Principles of color technology*. Wiley-Interscience publication. Wiley, 1981.

[BS05] Tamy Boubekeur and Christophe Schlick. Generic Mesh Refinement On GPU. In *ACM SIGGRAPH/Eurographics Graphics Hardware*, 2005.

[BS08] A. I. Bobenko and Y. B. Suris. *Discrete Differential Geometry: Integrable Structure*. Graduate Studies in Mathematics. American Mathematical Society, 2008.

[BSFG09] Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B. Goldman. PatchMatch: A randomized correspondence algorithm for structural image editing. *ACM Trans. Graph.*, 28(3):24:1–24:11, July 2009.

[BTS05] Pascal Barla, Joëlle Thollot, and François Sillion. Geometric Clustering for Line Drawing Simplification. In *Proceedings of the Eurographics Symposium on Rendering*, 2005.

[BTT07] Pascal Barla, Joëlle Thollot, and Gwenola Thomas. Rendu expressif. In D. Bechmann and B. Péroche (Eds.) *Informatique graphique et rendu, Traitement du signal et de l'image*, Chapter 11. Hermes–Lavoisier, 2007.

[Bun05] Michael Bunnell. Dynamic Ambient Occlusion and Indirect Lighting. In Matt Pharr and Randima Fernando (Eds.) *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley, 2005.

[BVGP09] Ilya Baran, Daniel Vlasic, Eitan Grinspun, and Jovan Popović. Semantic deformation transfer. *ACM Trans. Graph.*, 28(3):36:1–36:6, July 2009.

[BW90] Jules Bloomenthal and Brian Wyvill. Interactive techniques for implicit modeling. In *Proceedings of the 1990 Symposium on Interactive 3D Graphics, I3D '90*, pages 109–116, 1990.

[BW97a] David Baraff and Andrew Witkin. Physically Based Modeling: Principles and Practice. SIGGRAPH '97 course notes, available online at `http://www.cs.cmu.edu/~baraff/sigcourse/notesa.pdf`, 1997.

[BW97b] Jules Bloomenthal and Brian Wyvill (Eds.). *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[BWR+05] Eric Burns, Mary C. Whitton, Sharif Razzaque, Matthew R. Mccallus, Abigail T. Panter, and Frederick P. Brooks. The hand is slower than the eye: A quantitative exploration of visual dominance over proprioception. *IEEE Virtual Reality 2005 Conference Proceedings*, pages 3–10, 2005.

[Car84] Loren Carpenter. The A-buffer, an antialiased hidden surface method. *SIGGRAPH Comput. Graph.*, 18(3):103–108, January 1984.

[CAS+97] Cassidy J. Curtis, Sean E. Anderson, Joshua E. Seims, Kurt W. Fleischer, and David H. Salesin. Computer-generated watercolor. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 421–430, New York, NY, USA, 1997. ACM Press/Addison-Wesley.

[Cat74] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, USA, December 1974.

[CC98] E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. In *Seminal Graphics*, pages 183–188. ACM, New York, NY, USA, 1998.

[CCC87] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes image rendering architecture. *SIGGRAPH Comput. Graph.*, 21(4):95–102, August 1987.

[CD05a] Eric Chan and Fredo Durand. Fast Prefiltered Lines. In Matt Pharr and Randima Fernando (Eds.), *GPU Gems 2*, Chapter 22. NVIDIA, 2005.

[CD05b] Robert L. Cook and Tony DeRose. Wavelet noise. *ACM Trans. Graph.*, 24(3):803–811, July 2005.

[CF00] Stephen Chenney and D. A. Forsyth. Sampling plausible solutions to multi-body constraint problems. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 219–228, New York, NY, USA, 2000. ACM Press/Addison-Wesley.

[CGL+12] Forrester Cole, Aleksey Golovinskiy, Alex Limpaecher, Heather Stoddart Barros, Adam Finkelstein, Thomas Funkhouser, and Szymon Rusinkiewicz. Where Do People Draw Lines? *Communications of the ACM*, 55(1):107–115, January 2012.

[Che46] C. Chevalley. *Theory of Lie Groups*. Number 1 in Princeton Mathematical Series. Princeton University Press, 1946.

[CK96] G. C. H. Chuang and C. C. J. Kuo. Wavelet descriptor of planar curves: Theory and applications. *IEEE Transactions on Image Processing,* 5(1):56–70, 1996.

[CKS98] Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. Directed edges–A scalable representation for triangle meshes. *J. Graph. Tools*, 3(4):1–11, 1998.

[Cla76] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10):547–554, October 1976.

[CM83] William S. Cleveland and Robert McGill. A Color-Caused Optical Illusion on a Statistical Graph. *The American Statistician*, 37(2):101–105, 1983.

[Con12] International Color Consortium. Introduction to the ICC profile format. `http://www.color.org/iccprofile.xalter`, 2012.

[Coo67] S. A. Coons. Surfaces for Computer-Aided Design of Space Forms. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1967.

[Coo84] Robert L. Cook. Shade trees. *SIGGRAPH Comput. Graph.*, 18(3):223–231, January 1984.

[Coo86] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Trans. Graph.*, 5(1):51–72, January 1986.

[Coo10] Robert L. Cook. Personal communication, 2010.

[Cow83] William B. Cowan. An inexpensive scheme for calibration of a colour monitor in terms of CIE standard coordinates. In *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '83, pages 315–321, New York, NY, USA, 1983. ACM.

[CP98] Shek Ling Chan and Enrico O. Purisima. A new tetrahedral tesselation scheme for isosurface generation. *Computers and Graphics*, 22(1):83–90, 1998.

[CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18(3):137–145, 1984.

[CPMV+09] E. Chorro, E. Perales, F. M. Martánez-Verdú, J. Campos, and A. Pons. Colorimetric and spectral evaluation of the optical anisotropy of metallic and pearlescent samples. *Journal of Modern Optics*, 56(13):1457–1465, 2009.

[CPS11] Keenan Crane, Ulrich Pinkall, and Peter Schröder. Spin transformations of discrete surfaces. *ACM Trans. Graph.*, 30(4):104:1–104:10, July 2011.

[Cra68] F. S. Crawford. *Waves*. Berkeley Physics Course. McGraw-Hill, 1968.

[Cro77] Franklin C. Crow. The aliasing problem in computer-generated shaded images. *Commun. ACM*, 20(11):799–805, November 1977.

[Cro84] Gary A. Crocker. Invisibility coherence for faster scan-line hidden surface algorithms. *SIGGRAPH Comput. Graph.*, 18(3):95–102, January 1984.

[CSKK99] Baoquan Chen, J. Edward Swan II, Eddy Kuo, and Arie Kaufman. LOD-sprite technique for accelerated terrain rendering. In *Proceedings of the Conference on Visualization '99: Celebrating Ten Years*, VIS '99, pages 291–298, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[CT82] R. L. Cook and K. E. Torrance. A Reflectance Model for Computer Graphics. *ACM Trans. Graph.*, 1(1):7–24, January 1982.

[CTP+03] Matthieu Cunzi, Joëlle Thollot, Sylvain Paris, Gilles Debunne, Jean-Dominique Gascuel, and Frédo Durand. Dynamic Canvas for Immersive Non-Photorealistic Walkthroughs. In *Proc. Graphics Interface*. A K Peters, Ltd., June 2003.

[CWH93] Michael F. Cohen, John Wallace, and Pat Hanrahan. *Radiosity and realistic image synthesis*. Academic Press Professional, Inc., San Diego, CA, USA, 1993.

[dC76] M. P. do Carmo. *Differential geometry of curves and surfaces*. Prentice-Hall, 1976.

[DDSD03] Xavier Décoret, Frédo Durand, François X. Sillion, and Julie Dorsey. Billboard clouds for extreme model simplification. *ACM Trans. Graph.*, 22:689–696, July 2003.

[Deb06] Paul Debevec. The Story of Reflection Mapping. `http://www.pauldebevec.com/ReflectionMapping/`, 2006.

[Dee05] Michael F. Deering. A photon accurate model of the human eye. *ACM Trans. Graph.*, 24(3):649–658, July 2005.

[DFM07] Leo Dorst, Daniel Fontijne, and Stephen Mann. *Geometric Algebra for Computer Science: An Object-Oriented Approach to Geometry*. The Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann, 2007.

[DFRS03] Doug DeCarlo, Adam Finkelstein, Szymon Rusinkiewicz, and Anthony Santella. Suggestive contours for conveying shape. *ACM Trans. Graph.*, 22(3):848–855, July 2003.

[dGBOD12] Fernando de Goes, Katherine Breeden, Victor Ostromoukhov, and Mathieu Desbrun. Blue noise through optimal transport. *ACM Trans. Graph.*, 31(6):171:1–171:11, November 2012.

[DGPR02] David (grue) DeBry, Jonathan Gibbs, Devorah DeLeon Petty, and Nate Robins. Painting and rendering textures on unparameterized models. *ACM Trans. Graph.*, 21(3):763–768, July 2002.

[DHOO05] Simon Dobbyn, John Hamill, Keith O'Conor, and Carol O'Sullivan. Geopostors: A real-time geometry/impostor crowd rendering system. *ACM Trans. Graph.*, 24(3), 933–940, July 2005.

[DHS+05] Frédo Durand, Nicolas Holzschuch, Cyril Soler, Eric Chan, and François X. Sillion. A frequency analysis of light transport. *ACM Trans. Graph.*, 24(3):1115–1126, July 2005.

[dLE07] Eugene d'Eon, David Luebke, and Eric Enderton. Efficient rendering of human skin. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, EGSR '07, pages 147–157, 2007. Eurographics Association.

[DM85] H. Dym and H. P. McKean. *Fourier Series and Integrals.* Academic Press, San Diego, CA, USA, 1985.

[DMSB99] Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H. Barr. Implicit fairing of irregular meshes using diffusion and curvature flow. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 317–324, New York, NY, USA, 1999. ACM Press/Addison-Wesley.

[Dre07] Ulrich Drepper. What Every Programmer Should Know About Memory, `http://www.akkadia.org/drepper/cpumemory.pdf`, 2007.

[DRS08] Julie Dorsey, Holly Rushmeier, and François Sillion. *Digital Modeling of Material Appearance*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[Dru71] C. G. Drury. Movements with lateral constraint. *Ergonomics*, 14:293–305, 1971.

[DS02] Douglas DeCarlo and Anthony Santella. Stylization and abstraction of photographs. *ACM Trans. Graph.*, 21(3), pages 769–776, July 2002.

[Duf85] Tom Duff. Compositing 3-D rendered images. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, pages 41–44, New York, NY, USA, 1985. ACM.

[DvGNK99] Kristin J. Dana, Bram van Ginneken, Shree K. Nayar, and Jan J. Koenderink. Reflectance and texture of real-world surfaces. *ACM Trans. Graph.*, 18(1):1–34, January 1999.

[DWS+97] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing terrain: Real-time optimally adapting meshes. In *Proceedings of the 8th Conference on Visualization '97*, VIS '97, pages 81–88, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[DYN02] Yoshinori Dobashi, Tsuyoshi Yamamoto, and Tomoyuki Nishita. Interactive rendering of atmospheric scattering effects using graphics hardware. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, HWWS '02, pages 99–107, 2002. Eurographics Association.

[EL99] Alexei A. Efros and Thomas K. Leung. Texture Synthesis by Non-Parametric Sampling. In *Proceedings of the International Conference on Computer Vision-Volume 2*, ICCV '99, pages 1033–1038, Washington, DC, USA, 1999. IEEE Computer Society.

[Eld99] James H. Elder. Are Edges Incomplete? *Int. J. Comput. Vision*, 34 (2-3):97–122, October 1999.

[EMF02] Douglas Enright, Stephen Marschner, and Ronald Fedkiw. Animation and rendering of complex water surfaces. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 736–744, New York, NY, USA, 2002. ACM.

[EMX02] Regina Estkowski, Joseph S. B. Mitchell, and Xinyu Xiang. Optimal decomposition of polygonal models into triangle strips. In *Proceedings of the 18th Annual Symposium on Computational Geometry*, SCG '02, pages 254–263, New York, NY, USA, 2002. ACM.

[Eng86] Nick England. A Graphics System Architecture for Interactive Application-Specific Display Functions. *IEEE Computer Graphics and Applications*, 6(1):60–70, 1986.

[ESSL10] Eric Enderton, Erik Sintorn, Peter Shirley, and David Luebke. Stochastic Transparency. In *I3D '10: Proceedings of the 2010 Symposium on Interactive 3D Graphics and Games*, pages 157–164, New York, NY, USA, 2010.

[ESSL11] Eric Enderton, Erik Sintorn, Peter Shirley, and David P. Luebke. Stochastic Transparency. *IEEE TVCG*, 17(8):1036–1047, 2011.

[ETH+09] Kevin Egan, Yu-Ting Tseng, Nicolas Holzschuch, Frédo Durand, and Ravi Ramamoorthi. Frequency analysis and sheared reconstruction for rendering motion blur. In *ACM Trans. Graph.*, 28(3), pages 93:1–93:13, July 2009.

[Fat11] Raanan Fattal. Blue-Noise Point Sampling using Kernel Density Model. *ACM Trans. Graph.*, 30(4), pages 48:1–48:12, July 2011.

[FBH+10] Kayvon Fatahalian, Solomon Boulos, James Hegarty, Kurt Akeley, William R. Mark, Henry Moreton, and Pat Hanrahan. Reducing shading on GPUs using quad-fragment merging. *ACM Trans. Graph.*, 29(4):1–8, 2010.

[Fel68] William Feller. *An Introduction to Probability Theory and Its Applications, Vol. 1, 3rd Edition*. Wiley, January 1968.

[FFR83] Eugene Fiume, Alain Fournier, and Larry Rudolph. A parallel scan conversion algorithm with anti-aliasing for a general-purpose ultracomputer. In *SIGGRAPH '83: Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques*, pages 141–150, New York, NY, USA, 1983. ACM.

[Fi76] Munsell Color (Firm). *Munsell Book of Color*. Munsell Color, 1976.

[Fit54] P. M. Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology*, 47:381–391, 1954.

[FKN80] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On Visible Surface Generation by a Priori Tree Structures. *Computer Graphics (SIGGRAPH '80 Proceedings),* 14(3):124–133, July 1980.

[FLB+09] Kayvon Fatahalian, Edward Luong, Solomon Boulos, Kurt Akeley, William R. Mark, and Pat Hanrahan. Data-parallel rasterization of micropolygons with defocus and motion blur. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 59–68, New York, NY, USA, 2009. ACM.

[FLW02] Raanan Fattal, Dani Lischinski, and Michael Werman. Gradient domain high dynamic range compression. *ACM Trans. Graph.*, 21(3):249–256, July 2002.

[FMM+08] George Fitzmaurice, Justin Matejka, Igor Mordatch, Azam Khan, and Gordon Kurtenbach. Safe 3D navigation. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, I3D '08, pages 7–15, New York, NY, USA, 2008. ACM.

[Fro84] Francine S. Frome. Improving Color CAD Systems for Users: Some Suggestions from Human Factors Studies. *Design Test of Computers, IEEE*, 1(1):18 –27, February 1984.

[FS94] Adam Finkelstein and David H. Salesin. Multiresolution Curves. In *Proceedings of SIGGRAPH '94*, pages 261–268, July 1994. ACM.

[FWSB07] Clifton Forlines, Daniel Wigdor, Chia Shen, and Ravin Balakrishnan. Direct-touch vs. mouse input for tabletop displays. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 647–656, New York, NY, USA, 2007. ACM.

[Gar70] M. Gardner. The fantastic combinations of John Conway's new solitaire game "life." *Scientific American*, 223:120–123, October 1970.

[Gar85] Geoffrey Y. Gardner. Visual simulation of clouds. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, pages 297–304, New York, NY, USA, 1985. ACM.

[GBF03] Eran Guendelman, Robert Bridson, and Ronald Fedkiw. Nonconvex rigid bodies with stacking. *ACM Trans. Graph.*, 22(3):871–878, July 2003.

[GCA+09] Michael Glueck, Keenan Crane, Sean Anderson, Andres Rutnik, and Azam Khan. Multiscale 3D reference visualization. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 225–232, New York, NY, USA, 2009. ACM.

[GH81] Marvin J. Greenberg and John R. Harper. *Algebraic Topology: A First Course (Mathematics Lecture Note Series)*. Westview Press, Boulder, CO, USA, 1981.

[Gib77] J. J. Gibson. The Theory of Affordances. In R. Shaw and J. Bransford (Eds.), *Perceiving, Acting, and Knowing: Toward an Ecological Psychology*. Lawrence Erlbaum, 1977.

[GKB07] Tovi Grossman, Nicholas Kong, and Ravin Balakrishnan. Modeling pointing at targets of arbitrary shapes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 463–472, New York, NY, USA, 2007. ACM.

[GKM93] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical Z-buffer visibility. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 231–238, New York, NY, USA, 1993. ACM.

[Gla88] Andrew S. Glassner. Spacetime Ray Tracing for Animation. *IEEE Comput. Graph. Appl.*, 8:60–70, March 1988.

[Gla94] Andrew S. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[GM85] Michael Girard and A. A. Maciejewski. Computational modeling for the computer animation of legged figures. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, pages 263–270, New York, NY, USA, 1985. ACM.

[GP10] V. Guillemin and V. G. A. Pollack. *Differential Topology*. AMS Chelsea Publishing Series. AMS Chelsea Pub., 2010.

[Gra47] H. Grassmann. *Geometrische Analyse*. Weidmann'sche Buchhandlung, 1847.

[Gre96] Ned Greene. Hierarchical polygon tiling with coverage masks. In *SIGGRAPH '96: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 65–74, New York, NY, USA, 1996. ACM.

[Gre97] R. L. Gregory. *Eye and Brain: The Psychology of Seeing*. Princeton Science Library. Princeton University Press, 1997.

[Gre99] Donald P. Greenberg. A framework for realistic image synthesis. *Commun. ACM*, 42(8):44–53, 1999.

[GS89] IEEE Computer Graphics and Applications Staff. Return of the Jaggy. *IEEE Comput. Graph. Appl.*, 9(2):82–89, March 1989.

[GSCH93] Steven J. Gortler, Peter Schröder, Michael F. Cohen, and Pat Hanrahan. Wavelet radiosity. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 221–230, New York, NY, USA, 1993. ACM.

[GTR+06] Jinwei Gu, Chien-I Tu, Ravi Ramamoorthi, Peter Belhumeur, Wojciech Matusik, and Shree Nayar. Time-varying surface appearance: Acquisition, modeling and rendering. *ACM Trans. Graph.*, 25(3):762–771, July 2006.

[HA79] J. A. Hartigan and M. A. Wong. A K-Means Clustering Algorithm. *J. Royal Statistical Society (Applied Statistics)*, 28:100–108, 1979.

[Hae76] M. Haeusing. Color Coding of Information on Electronic Displays. In *Proc. 6th Congress Int'l. Ergonomics Assoc.*, pages 210–217, 1976.

[Hal12] R. Hall. *Illumination and Color in Computer Generated Imagery*. Monographs in Visual Communication. Springer, 2012.

[Ham53] William Rowan Hamilton. *Lectures on Quaternions*. Hodges and Smith, Dublin, Ireland, 1853.

[HAM06] Jon Hasselgren and Tomas Akenine-Möller. Efficient depth buffer compression. In *Proceedings of the 21st ACM SIGGRAPH/Eurographics Symposium on Graphics Hardware*, GH '06, pages 103–110, New York, NY, USA, 2006. ACM.

[Har09] Robin Hartshorne. *Foundations of Projective Geometry*. Ishi Press, 2009.

[HAT+00] Steven Haker, Sigurd Angenent, Allen Tannenbaum, Ron Kikinis, Guillermo Sapiro, and Michael Halle. Conformal Surface Parameterization for Texture Mapping. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):181–189, April 2000.

[Hav00] Havran Vlastimil. *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[HCS96] Li-wei He, Michael F. Cohen, and David H. Salesin. The virtual cinematographer: A paradigm for automatic real-time camera control and directing. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 217–224, New York, NY, USA, 1996. ACM.

[HD01] I. Herman and D. J. Duke. Minimal Graphics. *IEEE Computer Graphics and Applications*, 21:18–21, 2001.

[HDD+93] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 19–26, New York, NY, USA, 1993. ACM.

[HDD+94] Hugues Hoppe, Tony DeRose, Tom Duchamp, Mark Halstead, Hubert Jin, John McDonald, Jean Schweitzer, and Werner Stuetzle. Piecewise smooth surface reconstruction. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 295–302, New York, NY, USA, 1994. ACM.

[He93] Xiao Dong He. *Physically-based models for the reflection, transmission and subsurface scattering of light by smooth and rough surfaces, with applications to realistic image synthesis*. PhD thesis, Cornell University, Ithaca, NY, USA, 1993.

[Hec90] Paul S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. *SIGGRAPH Comput. Graph.*, 24(4):145–154, September 1990.

[HG97] Paul S. Heckbert and Michael Garland. Survey of Polygonal Surface Simplification Algorithms. Technical report, Carnegie-Mellon University, USA, 1997.

[Hil91] David Hilbert. Ueber die stetige Abbildung einer Line auf ein Flächenstück. *Mathematische Annalen*, 38(3):459–460, 1891.

[HKD93] Mark Halstead, Michael Kass, and Tony DeRose. Efficient, fair interpolation using Catmull-Clark surfaces. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 35–44, New York, NY, USA, 1993. ACM.

[HKS+97] M. Hostetter, D. Kranz, C. Seed, C. Terman, and S. Ward. Curl: A Gentle Slope Language for the Web. Technical report, MIT, Cambridge, MA, USA, 1997.

[HL01] Mark J. Harris and Anselmo Lastra. Real-Time Cloud Rendering. In *Computer Graphics Forum*, pages 76–84. Blackwell Publishing, 2001.

[HLW93] S. C. Hsu, I. H. H. Lee, and N. E. Wiseman. Skeletal strokes. In *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology*, UIST '93, pages 197–206, New York, NY, USA, 1993. ACM.

[Hof00] Donald D. Hoffman. *Visual Intelligence: How We Create What We See*. W. W. Norton & Company, February 2000.

[HOJ08] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. Progressive photon mapping. *ACM Trans. Graph.*, 27(5):130:1–130:8, December 2008.

[Hop96] Hugues Hoppe. Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 99–108, New York, NY, USA, 1996. ACM.

[Hop98] Hugues Hoppe. Efficient implementation of progressive meshes. *Computers and Graphics*, 22(1):27–36, 1998.

[Hop99] Hugues Hoppe. Optimization of mesh locality for transparent vertex caching. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 269–276, New York, NY, USA, 1999. ACM Press/Addison-Wesley.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[HP03] Naty Hoffman and Arcot J. Preetham. *Real-time light-atmosphere interactions for outdoor scenes*, in *Graphics Programming Methods*, pages 337–352. Charles River Media, Inc., Rockland, MA, USA, 2003.

[HPP05] Klaus Hildebrandt, Konrad Polthier, and Eike Preuss. Evolution of 3d Curves under Strict Spatial Constraints. In *Proceedings of the Ninth International Conference on Computer Aided Design and Computer Graphics*, CAD-CG '05, pages 40–45, Washington, DC, USA, 2005. IEEE Computer Society.

[HS84] David Hestenes and G. Sobczyk. *Clifford Algebra to Geometric Calculus*. Springer Verlag, 1984.

[Hub95] Philip M. Hubbard. Collision Detection for Interactive Graphics Applications. *IEEE Transactions on Visualization and Computer Graphics*, 1: 218–230, 1995.

[Hun05] R. W. G. Hunt. *The Reproduction of Colour*. The Wiley-IS&T Series in Imaging Science and Technology. John Wiley & Sons, 2005.

[Hus93] D. Husemöller. *Fibre Bundles*. Graduate Texts in Mathematics. Springer, 1993.

[HW96] Eric Haines and Steven Worley. Fast, Low Memory Z-Buffering When Performing Medium-Quality Rendering. *Journal of Graphics, GPU and Game Tools*, 1(3):1–5, 1996.

[HZ00] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 517–526, New York, NY, USA, 2000. ACM Press/Addison-Wesley.

[HZR+92] Kenneth P. Herndon, Robert C. Zeleznik, Daniel C. Robbins, D. Brookshire Conner, Scott S. Snibbe, and Andries van Dam. Interactive shadows. In *Proceedings of the 5th Annual ACM Symposium on User Interface Software and Technology*, UIST '92, pages 1–6, New York, NY, USA, 1992. ACM.

[IC01] Takeo Igarashi and Dennis Cosgrove. Adaptive unwrapping for interactive texture painting. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, I3D '01, pages 209–216, New York, NY, USA, 2001. ACM.

[ICG86] David S. Immel, Michael F. Cohen, and Donald P. Greenberg. A radiosity method for non-diffuse environments. *SIGGRAPH Comput. Graph.*, 20(4):133–142, August 1986.

[IH01] Takeo Igarashi and John F. Hughes. A suggestive interface for 3D drawing. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology*, UIST '01, pages 173–181, New York, NY, USA, 2001. ACM.

[IMT99] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: A sketching interface for 3D freeform design. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 409–416, New York, NY, USA, 1999. ACM Press/Addison-Wesley.

[Int97] Victoria Interrante. Illustrating surface shape in volume data via principal direction-driven 3D line integral convolution. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 109–116, New York, NY, USA, 1997. ACM Press/Addison-Wesley.

[Int03] Commission Electrotechnique Internationale. Multimedia systems and equipment—Colour measurement and management—Part 9: Digital Cameras. `webstore.iec.ch/preview/info_iec61966-9Bed2.0Den.pdf`, 2003.

[JAF+01] Henrik Wann Jensen, Jim Arvo, Marcos Fajardo, Pat Hanrahan, Don Mitchell, Matt Pharr, and Peter Shirley. *State of the Art in Monte Carlo Ray Tracing for Realistic Image Synthesis*, ACM, New York, NY, USA, August 2001.

[Jak12] Wenzel Jakob. Mitsuba physically based renderer. `http://www.mitsuba-renderer.org/`, 2012.

[JB10] Jon Jansen and Louis Bavoil. Fourier opacity mapping. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, pages 165–172, New York, NY, USA, 2010. ACM.

[JDA07] Tilke Judd, Frédo Durand, and Edward H. Adelson. Apparent ridges for line drawing. *ACM Trans. Graph.*, 26(3):19, 2007.

[Jen01] Henrik Wann Jensen. *Realistic image synthesis using photon mapping*. A K Peters, Ltd., Natick, MA, USA, 2001.

[JLSW02] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. Dual contouring of Hermite data. *ACM Trans. Graph.*, 21(3):339–346, July 2002.

[JMLH01] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 511–518, New York, NY, USA, 2001. ACM.

[Jon71] C. B. Jones. A New Approach to the 'Hidden Line' Problem. *Computer Journal*, 14(3):232–237, August 1971.

[JP02] Doug L. James and Dinesh K. Pai. DyRT: Dynamic response textures for real time deformation simulation with graphics hardware. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 582–585, New York, NY, USA, 2002. ACM.

[JSW05] Tao Ju, Scott Schaefer, and Joe Warren. Mean value coordinates for closed triangular meshes. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 561–566, New York, NY, USA, 2005. ACM.

[Ju04] Tao Ju. Robust repair of polygonal models. *ACM Trans. Graph.*, 23(3):888–895, August 2004.

[Jud75] D. B. Judd. *Color in business, science, and industry*. Wiley, New York, 1975.

[Kaj86] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM.

[Kar72] R. M. Karp. Reducibility Among Combinatorial Problems. In R. E. Miller and J. W. Thatcher (Eds.), *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[KBSS01] Leif P. Kobbelt, Mario Botsch, Ulrich Schwanecke, and Hans-Peter Seidel. Feature sensitive surface extraction from volume data. In *SIGGRAPH '01: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pages 57–66, New York, NY, USA, 2001. ACM.

[KCODL06] Johannes Kopf, Daniel Cohen-Or, Oliver Deussen, and Dani Lischinski. Recursive Wang tiles for real-time blue noise. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 509–518, New York, NY, USA, 2006. ACM.

[KDC+08] Ladislav Kavan, Simon Dobbyn, Steven Collins, Jiri Zara, and Carol O'Sullivan. Polyposters: 2D polygonal impostors for 3D crowds. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 149–155, New York, NY, USA, 2008. ACM.

[Kel97] Alexander Keller. Instant radiosity. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 49–56, New York, NY, USA, 1997. ACM Press/Addison-Wesley.

[KH06] Olga A. Karpenko and John F. Hughes. SmoothSketch: 3D free-form shapes from complex sketches. *ACM Trans. Graph.*, 25(3):589–598, July 2006.

[Kil99] Mark J. Kilgard. Improving Shadows and Reflections via the Stencil Buffer. Technical report, NVIDIA Corporation, 1999. `http://developer.nvidia.com/attach/6641`.

[KK89] J. T. Kajiya and T. L. Kay. Rendering fur with three dimensional textures. *SIGGRAPH Comput. Graph.*, 23(3):271–280, July 1989.

[KKA05] Ryo Kondo, Takashi Kanai, and Ken-ichi Anjyo. Directable animation of elastic objects. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '05, pages 127–134, New York, NY, USA, 2005. ACM.

[KKS+05] Azam Khan, Ben Komalo, Jos Stam, George Fitzmaurice, and Gordon Kurtenbach. HoverCam: Interactive 3D navigation for proximal object

inspection. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, I3D '05, pages 73–80, New York, NY, USA, 2005. ACM.

[KLA04] Jan Kautz, Jaakko Lehtinen, and Timo Aila. Hemispherical Rasterization for Self-Shadowing of Dynamic Objects. In *Rendering Techniques*, pages 179–184, 2004.

[KMDZ09] Denis Kovacs, Jason Mitchell, Shanon Drone, and Denis Zorin. Real-time creased approximate subdivision surfaces. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 155–160, New York, NY, USA, 2009. ACM.

[KMF+08] Azam Khan, Igor Mordatch, George Fitzmaurice, Justin Matejka, and Gord Kurtenbach. ViewCube: A 3D Orientation Indicator and Controller. In *I3D 2008 Conference Proceedings: ACM Symposium on Interactive 3D Graphics*, pages 17–25, 2008.

[KO11] Max Kazakov and Eisaku Ohbuchi. Primitive processing and advanced shading architecture for embedded space. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 169–176, New York, NY, USA, 2011. ACM.

[Koe11] Jan Koenderink. Exploration of Pictorial Space. Talk presented at Brown University, USA, 2011.

[KSKAC02] Csaba Kelemen, László Szirmay-Kalos, György Antal, and Ferenc Csonka. A Simple and Robust Mutation Strategy for the Metropolis Light Transport Algorithm. In *Computer Graphics Forum*, pages 531–540, 2002.

[KSS02] Jan Kautz, Peter-Pike Sloan, and John Snyder. Fast, arbitrary BRDF shading for low-frequency lighting using spherical harmonics. In *Proceedings of the 13th Eurographics Workshop on Rendering*, EGRW '02, pages 291–296, 2002. Eurographics Association.

[KUB54] Paul Kubelka. New Contributions to the Optics of Intensely Light-Scattering Materials. Part II: Nonhomogeneous Layers. *J. Opt. Soc. Am.*, 44(4):330–334, April 1954.

[KVH84] James T. Kajiya and Brian P. Von Herzen. Ray tracing volume densities. *SIGGRAPH Comput. Graph.*, 18(3):165–174, January 1984.

[KW79] M. Krebs and J. Wolf. Design Principles for the Use of Color in Displays. *Proc. Soc. Information Display*, 20:10–15, 1979.

[Laf96] Eric Lafortune. *Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering*. PhD thesis, Katholieke Universiteit Leuven, 1996.

[Lan02] Hayden Landis. Production-Ready Global Illumination, SIGGRAPH Course Notes (2002), Volume 16, ACM.

[Law04] Jason Lawrence. Efficient BRDF Importance Sampling Using a Factored Representation. *ACM Trans. Graph.*, 23:496–505, 2004.

[Law06] Jason Lawrence. *Acquisition and Representation of Material Appearance for Editing and Rendering*. PhD thesis, Princeton University, USA, September 2006.

[LBJK09] Manfred Lau, Ziv Bar-Joseph, and James Kuffner. Modeling spatial and temporal variation in motion data. In *ACM SIGGRAPH Asia 2009*

papers, SIGGRAPH Asia '09, pages 171:1–171:10, New York, NY, USA, 2009. ACM.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, August 1987.

[LCW03] Beitao Li, Edward Chang, and Yi Wu. Discovery of a perceptual distance function for measuring image similarity. *Multimedia Systems*, 8:512–522, 2003.

[LD12] Gábor Liktor and Carsten Dachsbacher. Decoupled deferred shading for hardware rasterization. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 143–150, New York, NY, USA, 2012. ACM.

[Lee09] Jeffrey N. Lee. *Manifolds and Differential Geometry*. AMS, 2009.

[Lei10] Kefei Lei. Assessment of Microfacet Theory Based on First Principles and Actual Surface Microgeometry. Williams College Honors Thesis, 2010.

[Lev06] Adi Levin. Modified subdivision surfaces with continuous curvature. *ACM Trans. Graph.*, 25(3):1035–1040, July 2006.

[LFTG97] Eric P. F. Lafortune, Sing-Choong Foo, Kenneth E. Torrance, and Donald P. Greenberg. Non-linear approximation of reflectance functions. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 117–126, New York, NY, USA, 1997. ACM Press/Addison-Wesley.

[LG95] David Luebke and Chris Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, I3D '95, pages 105–107, New York, NY, USA, 1995. ACM.

[LGS+09] C. Lauterbach, M. Garl, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH construction on GPUs. *Computer Graphics Forum*, 28(2), pages 375–384, 2009.

[LHLW10] Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. FreePipe: A programmable parallel rendering architecture for efficient multifragment effects. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, pages 75–82, New York, NY, USA, 2010. ACM.

[Lis48] J. B. Listing. *Vorstudien zur Topologie*. Vandenhoeck und Ruprecht, 1848.

[LJ99] Eugene Lapidous and Guofang Jiao. Optimal depth buffer for low-cost graphics hardware. In *Proceedings of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, HWWS '99, pages 67–73, New York, NY, USA, 1999. ACM.

[LK10] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, pages 55–63, New York, NY, USA, 2010. ACM.

[LK11] Samuli Laine and Tero Karras. High-Performance Software Rasterization on GPUs. In *Proceedings of High-Performance Graphics 2011*, 2011.

[LKHW03] Aaron Lefohn, Joe Kniss, Charles Hansen, and Ross Whitaker. Interactive Deformation and Visualization of Level Set Surfaces Using Graphics Hardware. Technical report, School of Computing, University of Utah, USA, 2003.

[LKMK97] J. Lacouture, D. Kersten, P. Mamassian, and D. C. Knill. Moving cast shadows induce apparent motion in depth. *Perception*, 26, 1997.

[LMHB00] Adam Lake, Carl Marshall, Mark Harris, and Marc Blackstein. Stylized rendering techniques for scalable real-time 3D animation. In *Proceedings of the 1st International Symposium on Non-photorealistic Animation and Rendering*, NPAR '00, pages 13–20, New York, NY, USA, 2000. ACM.

[LMLH07] Yunjin Lee, Lee Markosian, Seungyong Lee, and John F. Hughes. Line drawings via abstracted shading. *ACM Trans. Graph.*, 26(3):18, 2007.

[Lö81] P. Lötstedt. Coulomb Friction in Two-Dimensional Rigid Body Systems. *ZAMM–Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik*, 61(12):605–615, 1981.

[Loo87] Charles T. Loop. Smooth Subdivision Surfaces Based on Triangles. Master's thesis, University of Utah, USA, August 1987.

[Lor07] Tristan Lorach. Soft Particles. Technical report, NVIDIA, 2007.

[LRR04] Jason Lawrence, Szymon Rusinkiewicz, and Ravi Ramamoorthi. Efficient BRDF importance sampling using a factored representation. *ACM Trans. Graph.*, 23(3), pages 496–505, August 2004.

[LS08] Charles Loop and Scott Schaefer. Approximating Catmull-Clark subdivision surfaces with bicubic patches. *ACM Trans. Graph.*, 27(1):8:1–8:11, March 2008.

[LTG92] Dani Lischinski, Filippo Tampieri, and Donald P. Greenberg. Discontinuity Meshing for Accurate Radiosity. *IEEE Comput. Graph. Appl.*, 12(6):25–39, November 1992.

[Lue01] David P. Luebke. A Developer's Survey of Polygonal Simplification Algorithms. *IEEE Comput. Graph. Appl.*, 21(3):24–35, May 2001.

[LV00] Tom Lokovic and Eric Veach. Deep shadow maps. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 385–392, New York, NY, USA, 2000. ACM Press/Addison-Wesley.

[LW93] Eric P. Lafortune and Yves D. Willems. Bi-Directional Path Tracing. In *Proceedings of the 3rd International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)*, pages 145–153, 1993.

[Mar82] Aaron Marcus. Color: A Tool for Computer Graphics Communication. In *The Computer Image*, pages 76–90, New York, NY, USA. Addison-Wesley, 1982.

[Mat] Math.net. Math.Net numerics. `http://numerics.mathdotnet.com/`.

[MB07] Kevin Myers and Louis Bavoil. Stencil routed A-Buffer. In *ACM SIGGRAPH 2007 sketches*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[McC94] Scott McCloud. *Understanding Comics: The Invisible Art.* Harper Collins, 1994.

[MCCH99] Lee Markosian, Jonathan M. Cohen, Thomas Crulli, and John F. Hughes. Skin: A constructive approach to modeling free-form shapes. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 393–400, New York, NY, USA, 1999. ACM Press/Addison-Wesley.

[McG] Morgan McGuire. The Brown Mesh Set. `graphics.cs.brown.edu/games/brown-mesh-set`, 2012.

[McG12] Morgan McGuire. The G3D Innovation Engine. `http://g3d.sf.net`, 2012.

[McT04] Gary McTaggart. Half-Life 2 / Valve Source Shading, March 2004.

[MDSB03] Mark Meyer, Mathieu Desbrun, Peter Schröder, and Alan H. Barr. Discrete Differential-Geometry Operators for Triangulated 2-Manifolds. In Hans-Christian Hege and Konrad Polthier (Eds.), *Visualization and Mathematics III*, pages 35–57. Springer-Verlag, Heidelberg, 2003.

[ME11] Morgan McGuire and Eric Enderton. Colored Stochastic Shadow Maps. In *Proceedings of the ACM Symposium on Interactive 3D Graphics and Games*, February 2011.

[Mei88] Barbara J. Meier. ACE: A color expert system for user interface design. In *Proceedings of the 1st Annual ACM SIGGRAPH Symposium on User Interface Software and Technology*, UIST '88, pages 117–128, New York, NY, USA, 1988. ACM.

[Mei96] Barbara J. Meier. Painterly rendering for animation. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 477–484, New York, NY, USA, 1996. ACM.

[Mei02] Erik Meijering. A chronology of interpolation from ancient astronomy to modern signal and image processing. *Proc. IEEE*, 90, 2002.

[MEP92] Steven Molnar, John Eyles, and John Poulton. PixelFlow: High-speed rendering using image composition. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '92, pages 231–240, New York, NY, USA, 1992. ACM.

[MESL10] Morgan McGuire, Eric Enderton, Peter Shirley, and David Luebke. Real-Time Stochastic Rasterization on Conventional GPU Architectures. In *Proceedings of High Performance Graphics 2010*, June 2010.

[Mil88] Gavin S. P. Miller. The motion dynamics of snakes and worms. *SIGGRAPH Comput. Graph.*, 22(4):169–173, June 1988.

[Mit87] Don P. Mitchell. Generating antialiased images at low sampling densities. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 65–72, New York, NY, USA, 1987. ACM.

[Mit96] Don P. Mitchell. Consequences of stratified sampling in graphics. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 277–280, New York, NY, USA, 1996. ACM.

[Mit07] Martin Mittring. Finding next gen: CryEngine 2. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, pages 97–121. ACM, New York, NY, USA, 2007.

[MKG+97] Lee Markosian, Michael A. Kowalski, Daniel Goldstein, Samuel J. Trychin, John F. Hughes, and Lubomir D. Bourdev. Real-time nonphotorealistic rendering. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 415–420, New York, NY, USA, 1997. ACM Press/Addison-Wesley.

[MKM89] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. *SIGGRAPH Comput. Graph.*, 23(3):41–50, July 1989.

[ML09] Morgan McGuire and David Luebke. Hardware-accelerated global illumination by image space photon mapping. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 77–89, New York, NY, USA, 2009. ACM.

[MLD97] Stephen Mann, Nathan Litke, and Tony Derose. A Coordinate Free Geometry ADT. Technical report, University of Waterloo, Research Report CS-97-15, 1997.

[MLW+99] Stephen R. Marschner, Stephen H. Westin, Eric P. F. Lafortune, and Kenneth E. Torrance. Image-based BRDF Measurement. *Applied Optics*, 39(16), 2000.

[MM02] Vincent C. H. Ma and Michael D. McCool. Low latency photon mapping using block hashing. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, HWWS '02, pages 89–99, 2002. Eurographics Association.

[MMK+00] Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Loring S. Holden, J. D. Northrup, and John F. Hughes. Art-based rendering with continuous levels of detail. In *Proceedings of the 1st International Symposium on Non-photorealistic Animation and Rendering*, NPAR '00, pages 59–66, New York, NY, USA, 2000. ACM.

[MMS06] Rafal Mantiuk, Karol Myszkowski, and Hans-Peter Seidel. A perceptual framework for contrast processing of high dynamic range images. *ACM Trans. Appl. Percept.*, 3(3):286–308, July 2006.

[MN88] Don P. Mitchell and Arun N. Netravali. Reconstruction filters in computer-graphics. *SIGGRAPH Comput. Graph.*, 22(4):221–228, June 1988.

[Moo65] Gordon Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[Moo85] David A. Moon. Architecture of the Symbolics 3600. *SIGARCH Comput. Archit. News*, 13(3):76–83, 1985.

[Mor66] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM, Ottawa, Canada, 1966.

[Mor70] M. Mori. Bukimi no tani (the uncanny valley). *Energy*, 7:33–35, 1970.

[MP77] R. S. Millman and G. D. Parker. *Elements of differential geometry*. Prentice-Hall, 1977.

[MP08] James McCann and Nancy S. Pollard. Real-time gradient-domain painting. *ACM Trans. Graph.*, 27(3):93:1–93:7, August 2008.

[MP09] James McCann and Nancy S. Pollard. Local layering. *ACM Trans. Graph.*, 28(3), 84:1–84:7, July 2009.

[MPBM03] Wojciech Matusik, Hanspeter Pfister, Matt Brand, and Leonard McMillan. A Data-Driven Reflectance Model. *ACM Trans. Graph.*, 22(3):759–769, July 2003.

[MS68] T. H. Myer and I. E. Sutherland. On the design of display processors. *Commun. ACM*, 11(6):410–414, 1968.

[MS74] J. W. Milnor and J. Stasheff. *Characteristic Classes*. Annals of Mathematics Studies, No. 76. Princeton University Press, 1974.

[MS95] Paulo W. C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, I3D '95, pages 95–102, New York, NY, USA, 1995. ACM.

[MS97] McGuire and Stone. Techniques for multi-resolution image registration in the presence of occlusions. In *Proceedings of the Image Registration Workshop*, X. NASA, 1997.

[MSK04] Barbara J. Meier, Anne Morgan Spalter, and David B. Karelitz. Interactive Color Palette Tools. *IEEE Comput. Graph. Appl.*, 24(3):64–72, May 2004.

[MSY07] Mark Micire, Martin Schedlbauer, and Holly A. Yanco. Horizontal Selection: An Evaluation of a Digital Tabletop Input Device. In John A. Hoxmeier and Stephen Hayn (Eds.), *AMCIS*, page 164. Association for Information Systems, 2007.

[MT97] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, 2(1):21–28, October 1997.

[MTPS04] Antoine McNamara, Adrien Treuille, Zoran Popović, and Jos Stam. Fluid control using the adjoint method. *ACM Trans. Graph.*, 23(3):449–456, August 2004.

[Mum94] David Mumford. Elastica and computer vision. In S. S. Abhyankar and C. Bajaj (Eds.), *Algebraic geometry and its applications: Collections of papers from Shreeram S. Abhyankar's 60th birthday conference*, pages 491–506. Springer-Verlag, 1994.

[Mum02] David Mumford. Pattern Theory: The Mathematics of Perception. ICM 2002, vol. 1, pages 401–422. Higher Education Press, 2002.

[Mur85] G. Murch. Using Color Effectively: Designing to Human Specifications. *Technical Comm*, pages 14–20., 4th Quarter 1985.

[Mye75] A. J. Myers. An Efficient Visible Surface Program. Technical Report to the National Science Foundation, Grant Number DCR 74-00768A01. Technical report, 1975.

[MZS09] Adriano Macchietto, Victor Zordan, and Christian R. Shelton. Momentum control for balance. *ACM Trans. Graph.*, 28(3):80:1–80:8, July 2009.

[Nay93] Bruce Naylor. Constructing Good Partitioning Trees. In *Proceedings of Graphics Interface '93*, pages 181–191, 1993.

[NBS06] Diego Nehab, Joshua Barczak, and Pedro V. Sander. Triangle order optimization for graphics hardware computation culling. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, pages 207–211, New York, NY, USA, 2006. ACM.

[NDM05] A. Ngan, F. Durand, and W. Matusik. Experimental analysis of BRDF models. In *Rendering Techniques (Proceedings of the Eurographics Symposium on Rendering)*, pages 117–226. Eurographics Association, 2005.

[NDN96] Tomoyuki Nishita, Yoshinori Dobashi, and Eihachiro Nakamae. Display of clouds taking into account multiple anisotropic scattering and sky light. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 379–386, New York, NY, USA, 1996. ACM.

[Net09] Netpbm. Documentation for Netpbm. `http://netpbm.sourceforge.net/doc/`, 2009.

[New18] I. Newton. *Opticks: or, A treatise of the reflections, refractions, inflections, and colours of light*. Printed for W. and J. Innys, 1718. Also reprinted by Dover Publications.

[NISA06] Andrew Nealen, Takeo Igarashi, Olga Sorkine, and Marc Alexa. Laplacian mesh optimization. In *Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia*, GRAPHITE '06, pages 381–389, New York, NY, USA, 2006. ACM.

[NKK91] S. K. Nayar, K. Ikeuchi, and T. Kanade. Surface Reflection: Physical and Geometrical Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(7):611–634, July 1991.

[NM00] J. D. Northrup and Lee Markosian. Artistic silhouettes: A hybrid approach. In *Proceedings of the 1st International Symposium on Nonphotorealistic Animation and Rendering*, NPAR '00, pages 31–37, New York, NY, USA, 2000. ACM.

[NMN87] Tomoyuki Nishita, Yasuhiro Miyawaki, and Eihachiro Nakamae. A shading model for atmospheric scattering considering luminous intensity distribution of light sources. *SIGGRAPH Comput. Graph.*, 21:303–310, August 1987.

[NN85] Tomoyuki Nishita and Eihachiro Nakamae. Continuous tone representation of three-dimensional objects taking account of shadows and interreflection. *SIGGRAPH Comput. Graph.*, 19(3):23–30, July 1985.

[NN94] Tomoyuki Nishita and Eihachiro Nakamae. Method of displaying optical effects within water using accumulation buffer. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 373–379, New York, NY, USA, 1994. ACM.

[NNS72] M. E. Newell, R. G. Newell, and T. L. Sancha. A solution to the hidden surface problem. In *Proceedings of the ACM Annual Conference, Volume 1*, ACM '72, pages 443–450, New York, NY, USA, 1972. ACM.

[OBW+08] Alexandrina Orzan, Adrien Bousseau, Holger Winnemöller, Pascal Barla, Joëlle Thollot, and David Salesin. Diffusion Curves: A Vector Representation for Smooth-Shaded Images. In *ACM Trans. Graph.*, 27(3), pages 92:1–92:8, August 2008.

[OG97] Marc Olano and Trey Greer. Triangle Scan Conversion using 2D Homogeneous Coordinates. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 89–96. ACM Press, 1997.

[O'G10] Maureen O'Gara. Azul Zings Its Java Hardware—Poof, It's Software. *JAVA Developer's Journal*, 15(6), 2010.

[Ols09] Dan Olsen. *Building Interactive Systems: Principles for Human Computer Interaction*. Cengage Learning, Inc., 2009.

[ON94] Michael Oren and Shree K. Nayar. Generalization of Lambert's reflectance model. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 239–246, New York, NY, USA, 1994. ACM.

[O'N06] B. O'Neill. *Elementary Differential Geometry, Revised 2nd Edition*. Elsevier Science, 2006.

[ORM08] Ryan Overbeck, Ravi Ramamoorthi, and William R. Mark. Large Ray Packets for Real-time Whitted Ray Tracing. In *IEEE/EG Symposium on Interactive Ray Tracing (IRT)*, pages 41–48, August 2008.

[OS88] Stanley Osher and James A. Sethian. Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations. *J. Comput. Phys.*, 79(1):12–49, November 1988.

[OS09] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd Edition, 2009.

[OTS06] Aude Oliva, Antonio Torralba, and Philippe G. Schyns. Hybrid images. *ACM Trans. Graph.*, 25(3):527–532, July 2006.

[P+10] Jef Poskanzer et al. PPM file format. `http://netpbm.sourceforge.net/doc/ppm.html`, 2010.

[Pan11] Jacopo Pantaleoni. VoxelPipe: A programmable pipeline for 3D voxelization. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 99–106, New York, NY, USA, 2011. ACM.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[Par07] Rick Parent. *Computer Animation, Second Edition: Algorithms and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd Edition, 2007.

[PBBW95] Randy Pausch, Tommy Burnette, Dan Brockway, and Michael E. Weiblen. Navigation and locomotion in virtual worlds via flight into hand-held miniatures. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pages 399–400, New York, NY, USA, 1995. ACM.

[PBD+10] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. OptiX: A General Purpose Ray Tracing Engine. *ACM Trans. Graph.*, 29(4), pages 66:1–66:13, August 2010.

[PC01] Frank Perbet and Maric-Paule Cani. Animating prairies in real-time. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, I3D '01, pages 103–110, New York, NY, USA, 2001. ACM.

[PD84] Thomas Porter and Tom Duff. Compositing digital images. *SIGGRAPH Comput. Graph.*, 18(3):253–259, January 1984.

[Pea85] Darwyn R. Peachey. Solid texturing of complex surfaces. *SIGGRAPH Comput. Graph.*, 19(3):279–286, July 1985.

[Per85] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, July 1985.

[Per95] Ken Perlin. Real Time Responsive Animation with Personality. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):5–15, 1995.

[Per02a] Ken Perlin. Improved Noise reference implementation. `http://mrl.nyu.edu/~perlin/noise/`, 2002.

[Per02b] Ken Perlin. Improving noise. *ACM Trans. Graph.*, 21(3):681–682, July 2002.

[PG96] Ken Perlin and Athomas Goldberg. Improv: A system for scripting interactive actors in virtual worlds. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 205–216, New York, NY, USA, 1996. ACM.

[PH06] Sang Il Park and Jessica K. Hodgins. Capturing and animating skin deformation in human motion. *ACM Trans. Graph.*, 25(3):881–889, July 2006.

[PH10] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd Edition, 2010.

[Pho75] Bui Tuong Phong. Illumination for Computer Generated Pictures. *Communications of the ACM*, 18:311–317, June 1975.

[Pin88] Juan Pineda. A parallel algorithm for polygon rasterization. *SIGGRAPH Comput. Graph.*, 22(4):17–20, 1988.

[PL10] J. Pantaleoni and D. Luebke, HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry. *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 87–95, Saarbrucken, Germany, 2010. Eurographics Association.

[Plü68] J. Plücker. *Neue geometrie des raumes gegründet auf die betrachtung der geraden linie als raumelement*. Druck und verlag von B.G. Teubner, Leipzig, 1868.

[POC05] Fábio Policarpo, Manuel M. Oliveira, and João L. D. Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, I3D '05, pages 155–162, New York, NY, USA, 2005. ACM.

[Poya] C. Poynton. Color FAQ. `http://www.poynton.com/ColorFAQ.html`, accessed 2012.

[Poyb] C. Poynton. Gamma FAQ. `http://www.poynton.com/GammaFAQ.html`, accessed 2012.

[PP93] Kris Popat and Rosalind W. Picard. Novel Cluster-Based Probability Model for Texture Synthesis, Classification, and Compression. In *Visual Communications and Image Processing*, pages 756–768, 1993.

[PP07] Bob Palais and Richard Palais. Euler's fixed point theorem: The axis of a rotation. *Journal of Fixed Point Theory and Applications*, 2(2):215–220, December 2007.

[Pre95] W. H. Press. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1995.

[PRS02] Jenny Preece, Yvonne Rogers, and Helen Sharp. *Interaction Design*. Wiley, 2002.

[Pur11] E. Purcell. *Electricity and Magnetism*. Cambridge University Press, 2011.

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 335–342, New York, NY, USA, 2000. ACM Press/Addison-Wesley.

[Ras60] N. Rashevsky. *Mathematical biophysics: Physico-mathematical foundations of biology*. Dover Publications, 1960.

[Ras90] Roshdi Rashed. A pioneer in anaclastics: Ibn Sahl on burning mirrors and lenses. *Isis*, 81:464–491, 1990.

[RBF08] Ganesh Ramanarayanan, Kavita Bala, and James A. Ferwerda. Perception of complex aggregates. *ACM Trans. Graph.*, 27(3):1–10, 2008.

[Ree83] W. T. Reeves. Particle Systems—A Technique for Modeling a Class of Fuzzy Objects. *ACM Trans. Graph.*, 2:91–108, April 1983.

[Rey87] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 25–34, New York, NY, USA, 1987. ACM.

[RFWB07] Ganesh Ramanarayanan, James Ferwerda, Bruce Walter, and Kavita Bala. Visual equivalence: Towards a new standard for image fidelity. *ACM Trans. Graph.*, 26(3), pages 76:1–76:11, July 2007.

[RH01] Ravi Ramamoorthi and Pat Hanrahan. An efficient representation for irradiance environment maps. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 497–500, New York, NY, USA, 2001. ACM.

[RH04] Ravi Ramamoorthi and Pat Hanrahan. A signal-processing framework for reflection. *ACM Trans. Graph.*, 23(4):1004–1042, October 2004.

[RKB04] Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. GrabCut: Interactive foreground extraction using iterated graph cuts. *ACM Trans. Graph.*, 23(3):309–314, August 2004.

[RKLC+11] Jonathan Ragan-Kelley, Jaakko Lehtinen, Jiawen Chen, Michael Doggett, and Frédo Durand. Decoupled sampling for graphics pipelines. *ACM Trans. Graph.*, 30(3):17:1–17:17, May 2011.

[Rob] Doug Roble. Personal communication.

[Roc95] I. Rock. *Perception*. Scientific American Library Series. Scientific American Library, 1995.

[Rod16] Olinde Rodrigues. De l'attraction des sphéroïdes. *Correspondence sur l'École Impériale Polytechnique (Thesis for the Faculty of Science of the University of Paris)*, 3(3):361–385, 1816.

[Roy88] Halsey Royden. *Real Analysis (3rd Edition)*. Prentice Hall, Englewood Cliffs, NJ, USA, 1988.

[RPG99] Mahesh Ramasubramanian, Sumanta N. Pattanaik, and Donald P. Greenberg. A perceptually based physical error metric for realistic image synthesis. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 73–82, New York, NY, USA, 1999. ACM Press/Addison-Wesley.

[RR78] D. R. Reddy and S. Rubin. Representation of three-dimensional objects. Technical Report CMU-CS-78-113, Dept. of Computer Science, Carnegie-Mellon University, USA, April 1978.

[RS09] Austin Robison and Peter Shirley. Image space gathering. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 91–98, New York, NY, USA, 2009. ACM.

[RSSF02] Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda. Photographic tone reproduction for digital images. *ACM Trans. Graph.*, 21(3):267–276, July 2002.

[Rus78] Richard M. Russell. The CRAY-1 computer system. *Commun. ACM*, 21(1):63–72, 1978.

[Rus08] Holly Rushmeier. Input for participating media. In *ACM SIGGRAPH 2008 Courses*, SIGGRAPH '08, pages 6:1–6:24, New York, NY, USA, 2008. ACM.

[RW80] Steven M. Rubin and Turner Whitted. A 3-dimensional representation for fast rendering of complex scenes. *SIGGRAPH Comput. Graph.*, 14(3):110–116, July 1980.

[SA04] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 2.0)*. OpenGL Architecture Review Board, 2004.

[SA07] Perumaal Shanmugam and Okan Arikan. Hardware accelerated ambient occlusion techniques on GPUs. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 73–80, New York, NY, USA, 2007. ACM.

[SAC+11] Peter Shirley, Timo Aila, Jonathan Cohen, Eric Enderton, Samuli Laine, David Luebke, and Morgan McGuire. A Local Image Reconstruction Algorithm for Stochastic Rendering. In *Proceedings of the ACM Symposiumon Interactive 3D Graphics and Games*, February 2011.

[SaLY+08] Pitchaya Sitthi-amorn, Jason Lawrence, Lei Yang, Pedro V. Sander, and Diego Nehab. An improved shading cache for modern GPUs. In *Proceedings of the 23rd ACM SIGGRAPH/Eurographics Symposium on Graphics Hardware*, GH '08, pages 95–101, 2008. Eurographics Association.

[SBGS69] R. A. Schumaker, B. Brand, M. Gilliland, and W. Sharp. Study for Applying Computer-Generated Images to Visual Simulation. Technical Report AFHRL-TR-69-14, U.S. Air Force Human Resources Laboratory, September 1969.

[SC99] Daniel J. Simons and Christopher F. Chabris. Gorillas in our midst: Sustained inattentional blindness for dynamic events. *Perception*, 28:1059–1074, 1999.

[SCB88] Maureen C. Stone, William B. Cowan, and John C. Beatty. Color gamut mapping and the printing of digital color images. *ACM Trans. Graph.*, 7(4):249–292, October 1988.

[Sch94] Christophe Schlick. An Inexpensive BRDF Model for Physically-based Rendering. *Computer Graphics Forum*, 13:233–246, 1994.

[Sch97] Gernot Schaufler. Nailboards: A Rendering Primitive for Image Caching in Dynamic Scenes. In *Proceedings of the Eurographics Workshop on Rendering Techniques '97*, pages 151–162, London, UK, 1997. Springer-Verlag.

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27:18:1–18:15, August 2008.

[SD11] Pradeep Sen and Soheil Darabi. On Filtering the Noise from the Random Parameters in Monte Carlo Rendering. *ACM Trans. Graph.*, 31(3): June 2012.

[SDS95] Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin. Wavelets for Computer Graphics: A Primer, Part 1. *IEEE Comput. Graph. Appl.*, 15(3):76–84, May 1995.

[Sed] Robert Sedgewick. Left-leaning Red-Black Trees. Talk presented at Workshop on Analysis of Algorithms, Maresias, Brazil, April 2008.

[Sek04] Dean Sekulic. Efficient Occlusion Culling. In Randima Fernando (Ed.), *GPU Gems*, Chapter 29. Addison-Wesley, 2004.

[SFB+09] Jeremy Sugerman, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, and Pat Hanrahan. GRAMPS: A programming model for graphics pipelines. *ACM Trans. Graph.*, 28(1):1–11, 2009.

[SG69] J. Spanier and E. M. Gelbard. *Monte Carlo principles and neutron transport problems*. Addison-Wesley series in computer science and information processing. Addison-Wesley 1969.

[SG81] Stuart Sechrest and Donald P. Greenberg. A visible polygon reconstruction algorithm. *SIGGRAPH Comput. Graph.*, 15(3):17–27, August 1981.

[SH74] Ivan E. Sutherland and Gary W. Hodgman. Reentrant polygon clipping. *Commun. ACM*, 17(1):32–42, January 1974.

[SH93] Peter Schröder and Pat Hanrahan. On the form factor between two polygons. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 163–164, New York, NY, USA, 1993. ACM.

[SH07] Alla Safonova and Jessica K. Hodgins. Construction and optimal search of interpolated motion graphs. *ACM Trans. Graph. (SIGGRAPH 2007)*, 26(3), August 2007.

[Sha49] Claude Elwood Shannon. Communication in the Presence of Noise. *Proceedings of the IRE*, 37(1):10–21, 1949.

[SHH99] Subhash Suri, Philip M. Hubbard, and John F. Hughes. Analyzing bounding boxes for object intersection. *ACM Trans. Graph.*, 18:257–277, July 1999.

[Shi10] Peter Shirley. Personal communication, 2010.

[Sho85] Ken Shoemake. Animating rotation with quaternion curves. *SIGGRAPH Comput. Graph.*, 19(3):245–254, July 1985.

[Sho92] Ken Shoemake. ARCBALL: A user interface for specifying three-dimensional orientation using a mouse. In *Proceedings of the Conference on Graphics Interface '92*, pages 151–156, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[SKS02] L. Shams, Y. Kamitani, and S. Shimojo. Visual illusion induced by sound. *Cognitive Brain Research*, 14(1):147–152, 2002.

[SL88] P. Samuel and S. Levy. *Projective Geometry*. Undergraduate texts in mathematics: Readings in mathematics. Springer-Verlag, 1988.

[Slo08] Peter-Pike Sloan. Stupid Spherical Harmonics (SH) Tricks, 2008.

[Smi] Roy Smith. Personal communication, rec.boats newsgroup.

[Smi95] Alvy Ray Smith. A Pixel Is Not A Little Square, A Pixel Is Not A Little Square, A Pixel Is Not A Little Square! (And A Voxel Is Not A Little Cube). Technical report, Technical Memo 6, Microsoft Research, 1995.

[SML11] Marco Salvi, Jefferson Montgomery, and Aaron Lefohn. Adaptive transparency. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 119–126, New York, NY, USA, 2011. ACM.

[SNB07] Pedro V. Sander, Diego Nehab, and Joshua Barczak. Fast triangle reordering for vertex locality and reduced overdraw. *ACM Trans. Graph.*, 26(3), July 2007.

[SP04] Robert W. Sumner and Jovan Popović. Deformation transfer for triangle meshes. *ACM Trans. Graph.*, 23(3):399–405, August 2004.

[Spa66] Edwin H. Spanier. *Algebraic Topology*. Springer, New York, NY, 1966.

[SPCJ09] Ben Schneiderman, Catherine Plaisant, Maxine Cohen, and Steven Jacobs. *Designing the User Interface: Strategies for Effective Human-Computer Interaction, 5th Edition*. Addison-Wesley, 2009.

[Spi65] Michael Spivak. *Calculus on manifolds. A modern approach to classical theorems of advanced calculus*. W. A. Benjamin, Inc., New York-Amsterdam, 1965.

[Spi79a] Michael Spivak. *A comprehensive introduction to differential geometry. Vol. I*. Publish or Perish Inc., Wilmington, DE, USA, 2nd Edition, 1979.

[Spi79b] Michael Spivak. *A comprehensive introduction to differential geometry. Vol. III*. Publish or Perish Inc., Wilmington, DE, USA, 2nd Edition, 1979.

[SPR06] A. Sheffer, E. Praun, and K. Rose. *Mesh Parameterization Methods and Their Applications*. Foundations and Trends in Computer Graphics and Vision Series. Now Publishers, 2006.

[SPW02] K. Sung, A. Pearce, and C. Wang. Spatial-Temporal Antialiasing. *IEEE Transactions on Visualization and Computer Graphics*, 8(2):144–153, 2002.

[SSD+09] Cyril Soler, Kartic Subr, Frédo Durand, Nicolas Holzschuch, and François Sillion. Fourier depth of field. *ACM Trans. Graph.*, 28(2):1–12, 2009.

[SSM+05] Philipp Slusallek, Peter Shirley, William Mark, Gordon Stoll, and Ingo Wald. Introduction to real-time ray tracing. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[SSP08] Boris Springborn, Peter Schröder, and Ulrich Pinkall. Conformal equivalence of triangle meshes. *ACM Trans. Graph.*, 27(3):77:1–77:11, August 2008.

[SSS74] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A Characterization of Ten Hidden-Surface Algorithms. *ACM Comput. Surv.*, 6(1):1–55, March 1974.

[SSW+06] P. Shirley, P. Slusallek, I. Wald, W. Mark, G. Stoll, D. Manocha, and A. Stephens. State of the Art in Interactive Ray Tracing, August 2006. SIGGRAPH 2006 Course.

[ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-D shapes. *SIGGRAPH Comput. Graph.*, 24(4):197–206, September 1990.

[ST91] Kenji Seki and Haruo Tamazawa. The Use of the Voxel Flinger (Reality) Three-Dimensional Image Processor in Evaluating Oral-Maxillofacial Diseases. In *Proceedings of the Second International Conference on Image Management and Communication*, 1991.

[Sta98] Jos Stam. Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 395–404, New York, NY, USA, 1998. ACM.

[Ste99] Norman Steenrod. *The topology of fibre bundles*. Princeton University Press, Princeton, NJ, USA, 1999.

[Str86] Steve Strassmann. Hairy brushes. *SIGGRAPH Comput. Graph.*, 20(4):225–232, August 1986.

[Str88] Paul Strauss. *BAGS: The Brown Animation Generation System*. PhD thesis, Brown University, USA, 1988.

[Sut63] Ivan E. Sutherland. Sketchpad: A Man-Machine Graphical Communication System. In E. Calvin Johnson (ed.), *Proceedings of the 1963 Spring Joint Computer Conference*, Volume 23 of *AFIPS Conference Proceedings*, pages 329–346, Baltimore, MD, USA, 1963. American Federation of Information Processing Societies, Spartan Books Inc.

[SW83] Paolo Sabella and Michael Wozny. Toward Fast Color-Shaded Images of CAD/CAM Geometry. *IEEE Computer Graphics and Applications*, 3:60–71, 1983.

[SWHS97] Michael P. Salisbury, Michael T. Wong, John F. Hughes, and David H. Salesin. Orientable textures for image-based pen-and-ink illustration. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 401–406, New York, NY, USA, 1997. ACM Press/Addison-Wesley.

[Swi08] S. Swink. *A Game Designer's Guide to Virtual Sensation*. Focal Press, 2008.

[SWL98] W. C. Snyder, Z. Wan, and X. Li. Thermodynamic constraints on reflectance reciprocity and Kirchhoff's law. *Applied Optics*, 37:3464–3470, 1998.

[SYLH10] Kwang Won Sok, Katsu Yamane, Jehee Lee, and Jessica Hodgins. Editing dynamic human motions via momentum and force. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, pages 11–20, 2010. Eurographics Association.

[SZT10] Jian Sun, Jiejie Zhu, and M. F. Tappen. Context-constrained hallucination for image super-resolution. In *2010 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 231–238, June 2010.

[Tau95] Gabriel Taubin. A signal processing approach to fair surface design. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pages 351–358, New York, NY, USA, 1995. ACM.

[TD06] Yaakov Tsaig and David L. Donoho. Compressed sensing. *IEEE Trans. Inform. Theory*, 52:1289–1306, 2006.

[Tel92] Seth Jared Teller. *Visibility computations in densely occluded polyhedral environments*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 1992.

[Tho61] J. E. Thornton. Parallel operation in the Control Data 6600. *Fall Joint Computer Conference*, 26:33–40, 1961.

[TJ07] Christopher D. Twigg and Doug L. James. Many-worlds browsing for control of multibody dynamics. *ACM Trans. Graph.*, 26(3), July 2007.

[TM07] Paul Allen Tipler and Gene Mosca. *Physics for Scientists and Engineers, Volume 2, Sixth Edition*. W. H. Freeman, 2007.

[TS67] K. E. Torrance and E. M. Sparrow. Theory for Off-Specular Reflection from Roughened Surfaces. *J. Opt. Soc. Am.*, 57(9):1105–1112, September 1967.

[Tur52] Alan Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society B*, 237:37–72, 1952.

[Tur91] Greg Turk. Generating textures on arbitrary surfaces using reaction-diffusion. *SIGGRAPH Comput. Graph.*, 25(4):289–298, July 1991.

[Uni90] International Telecommunication Union. ITU-R Recommendation BT.709. Basic parameter values for the HDTV standard for the studio and for international programme exchange. Technical Report BT.709 [formerly CCIR Rec. 709], International Telecommunication Union, 1211 Geneva 20, Switzerland, 1990.

[vB95] Kees van Overveld and Bart Barenbrug. All You Need Is Force: A constraint-based approach for rigid body dynamics in computer animation. In Dimitri Terzopoulos and Daniel Thalmann (Eds.), *Computer Animation and Simulation '95*, pages 80–94. Springer-Verlag, 1995.

[Vea96] Eric Veach. Non-symmetric scattering in light transport algorithms. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, pages 81–90, London, England, UK, 1996. Springer-Verlag.

[Vea97] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, USA, 1997.

[VG94] E. Veach and L. Guibas. Bidirectional estimators for light transport. In *Eurographics Rendering Workshop 1994 Proceedings*, pages 147–162, June 1994.

[VG97] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 65–76, New York, NY, USA, 1997. ACM Press/Addison-Wesley.

[Vla08] A. Vlachos. Post Processing in The Orange Box, 2008. Conference Session. Advanced Direct3D Tutorial Day, Game Developer's Conference, San Francisco, CA, USA, February 18, 2008.

[Voo91] Douglas Voorhies. Space-Filling Curves and a Measure of Coherence. In James Arvo (ed.), *Graphics Gems II*, pages 26–30. Academic Press, 1991.

[VPBM01] Alex Vlachos, Jörg Peters, Chas Boyd, and Jason L. Mitchell. Curved PN triangles. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, I3D '01, pages 159–166, New York, NY, USA, 2001. ACM.

[WA77] Kevin Weiler and Peter Atherton. Hidden surface removal using polygon area sorting. *SIGGRAPH Comput. Graph.*, 11:214–222, July 1977.

[War69] John Edward Warnock. *A hidden surface algorithm for computer generated halftone pictures*. PhD thesis, The University of Utah, USA, 1969.

[War92] Gregory J. Ward. Measuring and modeling anisotropic reflection. *SIGGRAPH Comput. Graph.*, 26(2):265–272, July 1992.

[War94] Gregory J. Ward. The RADIANCE lighting simulation and rendering system. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 459–472, New York, NY, USA, 1994. ACM.

[War96] Joe Warren. Barycentric Coordinates for Convex Polytopes. In *Advances in Computational Mathematics 6*, pages 97–108, 1996.

[Wat70] G. S. Watkins. A real-time visible surface algorithm. Technical Report UTEC-CSc-70-101, Computer Science Dept., Univ. of Utah, USA, June 1970.

[Wat90] Mark Watt. Light-water interaction using backward beam tracing. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, pages 377–385, New York, NY, USA, 1990. ACM.

[WBB08] I. Wald, C. Benthin, and S. Boulos. Getting Rid of Packets: Efficient SIMD Single-Ray Traversal using Multi-branching BVHs. In *IEEE/Eurographics Symposium on Interactive Ray Tracing '08*, 2008.

[WBM00] David J. Ward, Alan F. Blackwell, and David J. C. MacKay. Dasher: A data entry interface using continuous gestures and language models. In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology*, UIST '00, pages 129–137, New York, NY, USA, 2000. ACM.

[WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Trans. Graph.*, 26(1), 2007.

[WCG87] John R. Wallace, Michael F. Cohen, and Donald P. Greenberg. A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity methods. *SIGGRAPH Comput. Graph.*, 21(4):311–320, August 1987.

[Wen06] Carsten Wenzel. Real-time Atmospheric Effects in Games. http://developer.amd.com/media/gpu_assets/Wenzel-Real-time_Atmospheric_Effects_in_Games.pdf, 2006.

[Wer61] M. Wertheimer. Experimental studies on the seeing of motion. In T. Shipley (Ed.), *Classics in Psychology*, pages 1032–1088. Philosophical Library, New York, 1961.

[WGG99] Brian Wyvill, Andrew Guy, and Eric Galin. Extending the CSG Tree: Warping, Blending and Boolean Operations in an Implicit Surface Modeling System. *Comput. Graph. Forum*, 18(2):149–158, 1999.

[Whi10] T. Whitted. Disaggregated Graphics: Rich Clients for Clouds, 2010. Keynote for High Performance Graphics conference, June 2010, Saarbrucken, Germany.

[WHSG97] Bruce Walter, Philip M. Hubbard, Peter Shirley, and Donald P. Greenberg. Global illumination using local linear density estimation. *ACM Trans. Graph.*, 16(3):217–259, July 1997.

[Wik] Wikipedia. Adobe Photoshop. en.wikipedia.org/wiki/Adobe_Photoshop, accessed October 2012.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12(3):270–274, 1978.

[Wil83] Lance Williams. Pyramidal parametrics. *SIGGRAPH Comput. Graph.*, 17(3):1–11, July 1983.

[Wil94] Lance R. Williams. *Perceptual Completion of Occluded Surfaces*. PhD thesis, University of Massachusetts, Amherst, MA, USA, 1994.

[WK91] Andrew Witkin and Michael Kass. Reaction-diffusion textures. *SIGGRAPH Comput. Graph.*, 25(4):299–308, July 1991.

[WL00] Li-Yi Wei and Marc Levoy. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 479–488, New York, NY, USA, 2000. ACM Press/Addison-Wesley.

[WMG+09] Ingo Wald, William R. Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker, and Peter Shirley. State of the Art in Ray Tracing Animated Scenes. *Computer Graphics Forum*, 28(6):1691–1722, 2009.

[WMW86] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer*, 2:227–234, 1986.

[WPG02] B. Walter, S. N. Pattanaik, and D. P. Greenberg. Using Perceptual Texture Masking for Efficient Image Synthesis. *Computer Graphics Forum*, 21(3):393–399, 2002.

[WREE67] Chris Wylie, Gordon Romney, David Evans, and Alan Erdahl. Half-tone perspective drawings by computer. In *Proceedings of the November 14–16, 1967, Fall Joint Computer Conference*, AFIPS '67 (Fall), pages 49–58, New York, NY, USA, 1967. ACM.

[WS82] G. Wyszecki and W. S. Stiles. *Color science: Concepts and methods, quantitative data and formulae*. Wiley Classics Library. Wiley, 1982.

[WS94] Georges Winkenbach and David H. Salesin. Computer-generated pen-and-ink illustration. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 91–100, New York, NY, USA, 1994. ACM.

[WSBW01] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. In Alan Chalmers and Theresa-Marie Rhyne (Eds.), *Computer Graphics Forum*, pages 153–164, 2001.

[WSHD04] Joe Warren, Scott Schaefer, Anil N. Hirani, and Mathieu Desbrun. Barycentric coordinates for convex sets. *Advances in Computational Mathematics*, 27(3), pages 318–338, October 2007.

[WTF06] Rachel Weinstein, Joseph Teran, and Ronald Fedkiw. Dynamic Simulation of Articulated Rigid Bodies with Contact and Collision. *IEEE Trans. Vis. Comput. Graph.*, 12(3):365–374, 2006.

[Wu91] Xiaolin Wu. Fast Anti-Aliased Circle Generation. In James Arvo (Ed.), *Graphics Gems II*, pages 446–450. Academic Press, Boston, MA, USA, 1991.

[WW82] T. Whitted and D. M. Weimer. A Software Testbed for the Development of 3D Raster Graphics Systems. *ACM Trans. Graph.*, 1(1):43–58, 1982.

[WW94] William Welch and Andrew Witkin. Free-form shape design using triangulated surfaces. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 247–256, New York, NY, USA, 1994. ACM.

[WWZ+07] Lvdi Wang, Li-Yi Wei, Kun Zhou, Baining Guo, and Heung-Yeung Shum. High Dynamic Range Image Hallucination. In *Rendering Techniques*, pages 321–326, 2007.

[WXSC04] Jue Wang, Yingqing Xu, Heung-Yeung Shum, and Michael F. Cohen. Video tooning. *ACM Trans. Graph.*, 23:574–583, August 2004.

[Wym05] Chris Wyman. Interactive image-space refraction of nearby geometry. In *Proceedings of the 3rd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*, GRAPHITE '05, pages 205–211, New York, NY, USA, 2005. ACM.

[WZ10] Chun-Chih Wu and Victor Zordan. Goal-directed stepping with momentum control. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, pages 113–118, 2010. Eurographics Association.

[YD08] Jerry Yee and James Davis. Crowd rendering with non-planar 3D impostors. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, I3D '08, page 7:1, New York, NY, USA, 2008. ACM.

[Yel83] J. I. Yellott Jr, Spectral consequences of photoreceptor sampling in the rhesus retina. *Science*, 221:382–385, July 1983.

[YG89] Hussein Yahia and André Gagalowicz. Interactive Animation of Object Orientations. In *PIXIM '89*, pages 265–275, Paris, France, September 25–29, 1989.

[YKH04] Katsu Yamane, James J. Kuffner, and Jessica K. Hodgins. Synthesizing Animations of Human Manipulation Tasks. *ACM Trans. Graph. (SIGGRAPH 2004)*, 23(3), August 2004.

[YMMT08] Barbara Yersin, Jonathan Maïm, Fiorenzo Morini, and Daniel Thalmann. Real-time Crowd Motion Planning: Scalable Avoidance and Group Behavior. *Vis. Comput.*, 24(10):859–870, September 2008.

[You06] Peter Young. Coverage Sampled Antialiasing. Technical report, NVIDIA, October 2006.

[You10] Peter Young. Coverage sampled antialiasing (CSAA). Technical report, NVIDIA, Santa Clara, CA, USA, 2010.

[YRPF09] Thomas Y. Yeh, Glenn Reinman, Sanjay J. Patel, and Petros Faloutsos. Fool me twice: Exploring and exploiting error tolerance in physics-based animation. *ACM Trans. Graph.*, 29(1):5:1–5:11, December 2009.

[ZF99] Robert Zeleznik and Andrew Forsberg. UniCam—Gestural camera controls for 3D environments. In *Proceedings of the 1999 Symposium on Interactive 3D Graphics*, I3D '99, pages 169–173, New York, NY, USA, 1999. ACM.

[ZSS97] Denis Zorin, Peter Schröder, and Wim Sweldens. Interactive multiresolution mesh editing. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 259–268, New York, NY, USA, 1997. ACM Press/Addison-Wesley.

Index

Page numbers followed by “f” indicate a figure; and those followed by “t” indicate a table.

A
AABB trees, 1093
Absorption, 737
Abstract coordinate system, 39, 42
  to specify scene, 42–44
Abstract geometric *vs.* ready for rendering, 467
Abstraction, defined, 10
Abstraction, in expressive rendering, 947, 959–961
  factorization, 947
  kinds of, 947
  schematization, 947
  simplification, 947
Abstraction considerations, 444
Abstraction distance, 1138
A-buffer, 1057
Acceleration data structures, 472
Accretion, 569
ACC surfaces. *See* Approximate Catmull-Clark subdivision surfaces
Accumulation buffer, 1056
ACM. *See* Association for Computing Machinery
ACM Transactions on Graphics, 922
Acne, shadow, 416
Active edge table, 1041
Additive color, 760
Adjacency information, on meshes, 338–339
Adjacent vertex, 637
Adjoint transformation, 253
Affine combination, 154, 160
Affine combination of points, 160
Affine transformations, 182, 234, 259
Affordances (user interfaces), 572
Albedo, 547
Algebra, geometric, 284
Aliasing, 331, 544, 557–559, 837, 1055
  in line rendering, 544
Aliasing revisited, 527–729
Alpha-to-coverage, 366
Alpha value, 481
Alternative mesh structures, 187ff, 635ff, 338
AM. *See* Application model (AM)
Ambient light, 8, 122, 124
Ambient occlusion, 742
Ambient reflection, 136
AMIP. *See* Application-Model-to-IM-Platform Pipeline (AMIP)
Analytic BSDFs, 358
Angles, 686–688
  solid, 686–688
  subtended, 687
Animation element, in XAML, 55
Animation(s), 94, 963
  burden of temporal coherence in, 985–987
  considerations for rendering, 975–987
  creating a sailing ship firing a cannon (simulation), 969–972, 970f
  creating a walking character, 966–969
  double buffering, 975–976
  implicit curves in, 631–632
  implicit shapes in, 631–632
  interlacing, 978–980
  level-set approach to, 631–632
  motion blur, 980–983
  motion perception, 976–978
  navigating corridors (motion planning), 972–973
  notations related to, 973–975
  physically based, 963, 989
  problem of the first frame in, 984–985
  root frame, 972
  stop-motion, 987
  temporal aliasing, 980–983
  temporal coherence, exploiting, 983–984
  triple buffering, 976
  ways to produce, 964
Animator, 966, 989
Anisotropic materials, 883
Antialiasing, 498ff, 982, 985, 1055
  coverage sampling, 1058–1059
  multisample, 1057–1058
  spatial, 1055–1060

Antialiasing (*continued*)
supersampled, 1056–1057
supersampling techniques for, 985–986
API, 15, 25–26, 32
Appearance modeling, 742
Appel's ray-casting algorithm, 449, 797, 838
Apple Lisa, 568
Application model (AM), 36, 466–468
Application-Model-to-IM-Platform Pipeline (AMIP), 468–474
Application programming interfaces. *See* API
Approximate Catmull-Clark subdivision surfaces, 613
Approximation (graphics)
common forms of, 825–831
of the series solution of rendering equation, 847–848
Arcball interface, 280, 584
evaluation of, 584
practical implications of, 584
Architectural considerations, 444–445
Arctan, 152
Area-and-angle preserving, as a property of texture mapping, 555
Area lights, 377–379, 550, 698, 740, 784, 847, 866-868, 888, 915–918, 925–926. *See also* Area luminaire
hemispherical, 378–379
rectangular, 377–378
reflecting illumination from, 896
Area luminaire, 785, 788, 798, 891, 895. *See also* Area lights
Area-subdivision algorithms, 1041
Area-weighted radiance, 910. *See also* Biradiance
Aspect ratios
and field of view, 316–317
of triangles, 197
Associated transformations, 294
Attenuated geometric light source, 133

B
Back buffer, 971, 975
Back face, of polygon, 337
Backface culling, 337, 1023, 1028, 1047–1049
Backscattering, 730
Baking (models), 247
Band, of energies, 672
Band-limiting, 514, 522–523, 524, 534, 541
reconstruction and, 524–527
sampling and, 514–515
Band reconstruction, 534
Barycentric coordinates, 172, 183, 202, 203, 216, 218
analogs of, 182
Barycentric coordinates of x, 219
Barycentric interpolation, code for, 203–207
Basic graphics systems, 20–23
and graphics data, 21–23
Basis functions, 208, 596, 597, 600, 608, 609, 612, 625, 848–850. *See also* Tent-shaped functions
Basis matrix, 597–598, 603
Beckmann distribution function, 732
Beta phenomenon, 977
Bézier curve, 598, 607
Bézier patches, 607, 608–610, 609f
described, 608
Bicubic tensor product patch, 609
Bidirectional path tracing, 853, 870–871
schematic representation of, 853f
Bidirectional reflectance distribution function (BRDF), 646–647, 703, 783, 814, 834, 852, 946
anisotropic, 883, 885
Blinn-Phong, 883
cosine weighted, 820
glass and, 705–706
Lambertian, 814, 883
mirrors and, 705–706
Phong, 883
reciprocity and, 705–706
Bidirectional scattering distribution function (BSDF), 354–362, 704, 712, 820, 852
analytic, 358
isotropic, 883
local representation of, 882–887
measured, 358
Bidirectional surface scattering reflectance distribution function (BSSRDF), 704, 712, 738
Bidirectional transmittance distribution function (BTDF), 704
Bijective, 152. *See also* Injective
Billboard clouds, 348
Billboards, 648
and impostors, 347–348
Binary space partition (BSP) trees, 1023–1024, 1030, 1084–1089
building, 1089–1092
C++ implementation of, 1086
conservative ball intersection, 1088
conservative box intersection, 1088
first-ray intersection, 1088
*k*d tree, 1089
oct tree, 1090
pseudocode for visibility testing in, 1032
quad trees, 1090, 1091f
ray-primitive intersection, 1030–1032
2D binary space partition tree (BSP), 1086f
Binary tree (data structure), 1077
1D example, 1079
Binned rendering, 1137–1138
abstraction distance in, 1138
advantages of, 1138
deferred shading in, 1137–1138
drawbacks of, 1138

excess latency, 1138
full-scene anti-aliasing in, 1137
local memory in, 1137
poor multipass operation, 1138
properties of, 1137–1138
unbounded memory requirements, 1138
Biradiance, 910. *See also* Area-weighted radiance
Birefringence, 682
Bisection method, 830–831
Bitmaps, 38
Black body, 672
Blending, 362–364
and translucency, 361–364
Blindness, motion-induced, 114
Blinn-Phong BRDF, general form of, 721
Blinn-Phong model (reflection model), 138, 395ff, 414, 721–723
Blobby modeling, 343
Blob tree, 624
Bloom and lens flare, 369
Bloom focus, 336
Blue noise distribution, 921
Blue screening, 485
Blurring, 317, 543, 545, 983
Body-centered Euler angles, 272
Body-centered rotation, 272. *See also* Object-centered rotation
Boilerplate, 83
Boltzmann's constant, 674
Bottom-up construction, and composition, 140–144
Boundaries, and light transport, 798
Boundary component, 641
Boundary edge, 194, 637
determining, 638
Boundarylike vertex, 194
Boundary of a simplex, 638
Boundary vertex, 194, 638
Bounded color models, 771
Bounding box, 38, 197, 198f, 285, 420, 429, 631, 983
Bounding-box optimization, 420–421
beyond, 429
Bounding geometry, 1068
Bounding Volume Hierarchy (BVH), 916, 1049, 1092–1093, 1092f
Bounding volumes, 1068
BRDF. *See* Bidirectional reflectance distribution function (BRDF)
Brewster's angle, 682
Brightness (light), 108, 750, 756
just noticeable difference (JND), 754
perception of, 750–756
Brush (geometric primitive), 38
Brushstroke coherence, 986
BSDF. *See* Bidirectional scattering distribution function (BSDF)
B-spline basis matrix, 603
B-splines
cubic, 602–603
nonuniform, 604
nonuniform rational, 604
rational, 604
uniform spacing of, 604
BSSRDF. *See* Bidirectional surface scattering reflectance distribution function (BSSRDF)
BTDF. *See* Bidirectional transmittance distribution function (BTDF)
Buckets, 1093, 1093f. *See also* Grid cells
Buffers, 327–330
color, 328
depth, 329
framebuffer, 329
stencil, 329
Buffer swap, 443
Building blocks of ray optics, 330
Building transformations, from view specification, 303–310
Bump mapping, 547, 550–551

C

Cached and precomputed information on meshes, 340–341
Caching, 983, 1129ff
Callback procedure, 23
Camera(s), 336–337
depth of field, 301
design, 406
focal distance of, 301
orthographic, 315–317
perspective camera specification, 301–303
position of, 301
specifications and transformations, 299–317
transformation and rasterizing renderer ripeline, 310–312
Camera coordinates. *See* Camera-space coordinates
Camera setup, 460–461
Camera-space coordinates, 22, 299, 928
Camera visibility. *See* Primary visibility
Candelas (measurement unit of luminous intensity), 751
Capsule (3D volume), 1066
Cartoons, hand-animated, 966
Cathode-ray tubes (CRTs), 20, 770
Catmull-Clark subdivision surfaces, 610–613
Catmull-Rom spline, 540, 540f, 598–601
applications of, 602
generalization of, 601–602
nonuniform, 601
uniform, 601
Caustics, and light transport, 798
Cdf. *See* Cumulative distribution function (cdf)
Channels, 483
color, 483
object ID, 485
Chateau (user interface), 589–590

Chromatic aberration, 336–337, 680
Chunking rasterizer. *See* Tiling rasterizer
CIE chromaticity diagram, 765
  applications of, 766
  defining complementary colors, 766
  excitation purity and, 766
  indicating gamuts, 766
CIE *Luv* color coordinates, 767
CIE *L*u*v** uniform color space, 767
Circularly polarized wave, 677
Client area, 37, 39, 46
Clipping, 59, 63, 1045–1046
  of data, 24
  near-plane, 422, 1044, 1046
  Sutherland-Hodgman 2D clipping, 1045–1046
  whole-frustum clipping, 1044, 1047
Clipping planes, 122
Closed interval, 150
Closed meshes, 190
Closed oriented meshes, 195
Closed surface, 638
Clustering, 665–666
Clusters, 1043
CMTM. *See* Composite modeling transformation matrix
CMYK color, 774–775
Coded apertures, 493
Codomain, 151. *See also* Domain
  for texture maps, 553–554
Coefficient of extinction, 682, 728, 738
Coefficient of restitution, 1012
Coherence, 950
  spatial, 950
  temporal, 950, 962
Colatitude, 688
Collision proxy geometry, 337
Color bleeding, 839
Colorblindness, 746
Color buffer, 328
Color constancy, 110, 748
Color description, 756–758, 771–774
Colorimetry, 747
Color interpolation, 777–779
Color matching, 748
Color models
  bounded, 771
  RGB, 772–774
  YIQ, 775
Color naming, 748
Color palettes, 777
Color perception
  peripheral, 781–782
  physiology of the eye and, 748–750
  strengths and weaknesses, 761
Color percepts, 747
Color(s)
  choice, 777
  CIE description of, 762–766
  CMY, 774–775
  CMYK, 774–775
  coding, 779–780
  complementary, 766
  conventional color wisdom, 758–761
  description, 756–758, 771–774
  implications of, 746
  intensity-independent, 765
  interpolating, 777–779
  matching, 748
  naming, 748
  nonspectral, 766
  palettes, 777
  perceived distance between, 767
  perception of, 750–756
  percepts, 747
  perceptual spaces, 767–768
  primary, 758–759
  RGB sliders and, 761
  sensations, 747
  standard description of, 761–766
  use in computer graphics, 779–780
Color selection interfaces
  hue-lightness-saturation (HLS) interface, 776–777
  hue-saturation-value (HSV) interface, 776–777
Color sensations, 747
Color specification in WPF, 133
Color wisdom, conventional, 758–761
  blue and green make cyan, 760
  color is RGB, 761
  objects have colors, 759–760
  primary colors, 758–759
  purple isn't a real color, 759
Comb function, and transform of, 520
Commission Internationale de l'Éclairage (CIE), 755, 762–766
  chromaticity diagram, 765
  description of color, 762–766
Complementary colors, 766
Complex applications, processing demands of, 14
Complex conjugate, 512
Component hierarchy, top-down design of, 139–140
Components, reuse of, 144–147
Composion of transformations, 235
Composite component, constructing, 142
Composited image, 485
Composite modeling transformation matrix, 314
Composite transformation matrix, 246, 314, 463
Compositing of images
  operations, 488–489
  physical units and, 489–490
  simplifying, 487
Compression, use of splines for, 605
Compressive sensing, 530

Computability, as a property of texture mapping, 555
Computational photography, 493
Computations, stability of, 278
Computer-based animation industry, 932
Computer graphics, 1–33
  2D transformation library, 287–298
  3D transformation library, 287–298
  applications, 24–25
  basic graphics systems, 20–23
  brief history of, 7–9
  current and future application areas of, 4–6
  deep understanding vs. common practice, 12
  definition of, 2
  examples of, 9–10
  goals, resources, and appropriate abstractions, 10–12
  graphical user interfaces (GUI) and, 567–574
  graphics pipeline, 14–15
  interaction in graphics systems, 23–24
  introduction to, 1–4
  kinds of packages, 25–26
  learning, 31–33
  numbers and orders of magnitude in, 12–15
  physical/mathematical/numerical models of, 11
  relationship with art, design, and perception, 19–20
  using color in, 779–780
  world of, 4
Conceptual design (user interface), 570
Conductive materials, 714
Cones (color receptors), 107, 749
  generalized, 757
Conformal mapping, 555
Conservative rasterization, 1096
Conservative rasterizer, 430
Conservative visibility, 1023
Conservative visibility algorithm, 1023
Conservative visibility testing, 1023
  backface culling, 1023
  frustum culling, 1023
  methods of, 1023
  spatial data structures and, 1023
Conservative voxelization, 1096
Constancy
  application of, 111
  color, 110
  and its influences, 110–111
  shape, 110
  size, 110
Constant shading, 127
Constructive solid geometry (CSG), 450
Content, preparing viewport for, 120–122
Continuation, 111–112
  applications, 112
Continuous probability. *See* Continuum probability
Continuum, defined, 808
Continuum probability, 808–810, 815–818
Contour curve, 1048
Contour drawing, 551–552
Contour generator, 953. *See also* Contour(s)
Contour lines, 616
Contour points, 952
Contour(s), 551, 952, 953
  of a smooth surface, 644
  suggestive, 957–958
Contribution/detail culling, 470
Control data, 599
Control points, 599
Control templates, 49
Convex boundary polygon, 1045, 1045f
Convex cone, 747
Convex hull property, of cubic B-splines, 603
Convex polygons, 175
Convolution, 500–503
  defined, 500
  like computations, 504–505
  properties of, 503–504
Convolution multiplication theorem, 521
Cook-Torrance model, 731–732
Coons, Steven A., 608
Coordinate frame(s), 240–241
  defined, 240
  rigid, 240
Coordinates, 153
  operations on, 153–155
Coordinate system(s), 90–91
  abstract coordinate system, 42–44
  floating-point coordinates, 38–39
  integer coordinate system, 38–39
  physical coordinate system, 38
  spectrum of, 44–45
  transformations and, 229–230
  WPF canvas, 45–46
Coordinate vector, 155
Cornell box, 903, 911, 916–917
Corner-cutting, on polyline, 81, 83
Correspondence (meshes), 661
  building, 661
Cosine weighted BRDF, 820
Cosine-weighted sampling
  on hemisphere, 815
Cotangent rule, 658
Covectors, 163, 184, 520
  transforming, 250–253
Coverage
  binary, 1027–1028
  partial. *See* Partial coverage
Coverage of a pixel, 213
Coverage sampling antialiasing (CSAA), 1058–1059
Coverage testing, 422
Crease edges, 953
Critical angle, 682

Cross product, 157–158
CRT. *See* Cathode-ray tubes (CRTs)
CSG. *See* Constructive solid geometry (CSG)
C++ Standard Template Library (STL), 1074
CTM. *See* Composite transformation matrix
Cube map, 554
Cube mapping, 340
Cubic B-spline filter, 540, 540f
Cubic B-splines, 602–603
  convex hull property of, 603
  formula for, 602–603
  nonuniform, 602
  uniform, 602–603
Culling
  contribution/detail, 470
  occlusion, 470
  portal, 470
  sector-based, 470
  view-frustum, 470
Cumulative distribution function (cdf), 685
Curvature, 955
  line of, 956, 956f
  radial, 957
Curvature shadows, 946
Curved-surface
  representation and rendering, 128
Curves
  implicit, 616–619
Cyan-Magenta-Yellow (CMY) color, 774–775
Cybersickness, 571
Cylinder kernel, 910–911

D

Dangling edge, 637
Darken operation, 488
Data structures
  characterizing, 1077–1079
  generic use of, 1077
  ordered, 1077
  selecting, 1077
  spatial, 1065–1102
DDA. *See* Digital Difference Analyzer (DDA)
Debugging, 411–412
  rendering and, 915–919
Declarative animation
  dynamics animation via, 55–58
Declarative specification, 40
  vs. procedural code, 40
Deferred lighting, 441–442
Deferred-rendering method, 440
Deferred shading, 446, 1135–1137
  difficulties with, 1136
  excess storage and bandwidth in, 1136
  goal of, 1135
  multi-sample anti-aliasing (MSAA), incompatibility with, 1136
  shader-specified visibility and, 1136
Defocus, 1060–1061
Deformation (meshes), 660
Deformation transfer, 660–664
Degenerate transformation, 224
Degree
  of an edge, 637
  of a vertex, 637
Delta function, 519
Density estimation, 912
Depth buffer, 329, 392, 1023, 1028, 1034–1040
  common applications in visibility determination, 1035–1036
  common encodings, 1037–1040
  depth prepass, 1036
  encodings, 1037–1040
  screen-space visibility determination and, 1037
Depth buffer encodings, 1037–1040
  choices for, 1037
  hyperbolic in camera-space z, 1037
  linear in camera-space z, 1037
Depth complexity, defined, 446
Depth complexity of a ray, 1028
Depth map. *See* Depth buffer
Depth of field, 107, 301
Depth prepass, 1036
Depth-sort algorithm, 1042–1043
Depth value, 481
Derivative-based definitions of radiometric terms, 655–656, 700–702
The Design of Everyday Things (Norman), 593
Detail objects, 1051
Development tools, 41
Device code, 432
Device coordinates, 39
Diagonal matrices, 230
Differential coordinates, 655–657
Diffraction, 676, 677
  defined, 676
Diffuse, defined, 8
Diffuse reflection, 136
  physical models for, 726–727
Diffuse scattering, 713, 716
Diffusion (morphogens), 561
Diffusion curves, 961
Digital cameras, 5
  characteristics of, 13–14
Digital Difference Analyzer (DDA), 431
Digital signal processing, 545
Digital video cameras, 5
Direct3D, 452
Directed acyclic graph (DAG), 144, 248
Directed edges, 636
Directed-edge structure, 195
Direct illumination, 372–373
  interface to, 372–373
Directional curvature in direction **u**, 956
Directional hemispherical reflectance, 708

Directional light, 125
Directionally diffuse, 102
Direct light, 370. *See also* Indirect light
Direct lighting, 834. *See also* Direct illumination
Direct shadows, 946
Dirty bit flags, 983
Dirty rectangles, 983
Discrete attributes, 651
Discrete differential geometry, 644, 667
Discrete probability, 803–804
  relationship to programs, 803–804
Discrete probability space, 803
Displacement, 157
Displacement maps, 344, 547, 557ff, 562f, 647
Display-form-factor independence, 45
Display list, 473
Display transformation, 46–47
Distant objects, 346–348
Distribution (random variable), 806
Distribution ray tracing, 317, 838
Division of modeling principle, 210
DockPanel. *See* Panel
Dollying (camera control technique), 585
Domain, defined, 151. *See also* Codomains
Domain restriction, 827
Dominant wavelength, 747
Dot product, 158–159
Double-buffered rendering, 971, 975–976
Draw calls, 434
  executing, 442–444
Drawing, Dürer's, 62f, 64f, 68–72, 1035f
Drawing primitives, 461–462
Dual contouring, 653
Dual paraboloid, 554
Dual space, 163
Dual vectors. *See* Covectors
Dürer, Albrecht, 61–65
Dürer rendering algorithm
  implementation, 65–68
Dürer's drawing, 68–72
Dürer woodcut, 61–65, 1035f
Dynamic Canvas algorithm, 986, 986f
Dynamic range, 8
Dynamics, 463, 989, 996–1008
Dynamics animation, via declarative animation, 55–58

E

Early-depth-test
  defined, 446
  example, 445–447
Early *z*-cull, 1136
Edge aligns, 427
Edge collapse, 197
Edge-collapse costs, 649–652
Edge detection, 533, 544, 545
Edge(s), 189
  boundary, 637, 954
  crease, 953
  dangling, 637
  directed, 636
  interior, 637
  sharp, 651
  smooth, 953
Edges (computer vision), 952
Edge-swap operation, 197–198
Edge vectors, 175
Electric field
  linearly polarized, 678
Electromagnetic spectrum, 330–331, 675f
Elements, 41
  animation elements, 55–56
Elliptically polarized light, 679
Embedding topology, 637
Emission, 369, 737
Emissive lighting, 138
Emitters, 334–335
Empirical/phenomenological models, of scattering, 713, 717–725
Energy, photons transport, 333
Energy conservation, 714
Energy function (meshes), 650
Environment map, 549, 550
Environment mapping, 340, 549–550, 939–940
Equations
  approximate solutions of, 825–826
  approximating, 826
  domain restriction, 827
  methods for solving, 825–831
  Newton's method for solving, 831
  statistical estimators, using, 827–830
  using bisection for solving, 830–831
Estimation
  summing a series by, 828–830
Estimator random variable, 818
  bias, 822–823
  consistent sequence of, 818, 822–823
  unbiased, 818
  variance of, 818
Estimators. *See* Estimator random variable
Euclidean distance, 767
Euler angles, 267–269
  body-centered, 272
Euler characteristic of a mesh surface, 641
Euler integration, 278
Even function, 508
Event (interaction), 92
  handling, 85, 92–93
Event (probability), 802, 809
Excitation purity, 747
  chromaticity diagram and, 766
  defined, 747
Expectation. *See* Expected value
Expected value, 804–806, 810
  properties of, 806–808
  of a random variable, 810
  related terms, 806–808

Explicit equation, 341
Explicit Euler integration, 1016
Explicit Euler method, 1019. *See also* Forward Euler method
Explicit trapezoidal methods, 1020
Exponents (display process), 769–771
  encoding of, 769–771
Exposure time, 980. *See also* Shutter time
Expressive rendering, 945–962
  abstraction in, 947
  challenges of, 949–950
  examples of, 948
  geometric curve extraction in, 952–959
  gradient-based, 952
  marks, 950–951
  perception and salient features, 951–952
  perceptual relevance, 947
  research in, 947–948
  spatial coherence in, 950
  strokes, 950–951
  temporal coherence in, 950
Extended marching cubes algorithm, 653
Extensibility via shaders, 453
Extensible Application Markup Language (XAML), 35, 41, 928
  animation elements, 55–56
  structure of, 41–42
Eye, 106–110
  gross physiology of, 106–107
  luminous efficiency function for, 751
  physiology of, 748–750
  receptors, 107–110
  resolution, 13
Eye coordinates, 314
Eye path, 796
Eye ray
  and camera design notes, 406
  generating, 404–406
  testing eye-ray computation, 406–407
Eye ray visibility. *See* Primary visibility

F

Factorization (abstraction), 947
FF. *See* Fixed-function (FF)
Field of view, and aspect ratio, 316–317
Field radiance, 834, 846
Fields (half-resolution frames), 978
FillEllipse, 39
Fill rate, 14, 636
Filter(s)
  applying, 500
  Catmull-Rom filter, 542t
  cubic B-spline filter, 542t
  filtering f with, 502
  Gaussian filter, 542t, 543, 545
  Mitchell-Netravali filter, 542t
  separable, 544
  sinc filter with spacing one, 542t
  and their Fourier transforms, 542t
  unit box filter, 542t
Filtering, 500, 502, 557–559
Final gather step in photon mapping, 913
Finite element method, radiosity, 839
Finite element models, 349
Finite series
  summing by sampling and estimation, 828–829
Finite-state automaton (FSA), 574, 857
  probabilistic, 858f
Finite support, 535
Finite-support approximations, 540–541
First-person-shooter controls, 588
Fitts, Paul, 572
Fitts' Law, 572, 587
Fixed-function (FF), 452
  era, 452–453
  to programmable rendering pipeline, 452–454
Fixed-function 3D-graphics pipeline, 119
Fixed point, 325–326
Flat shading, 20. *See also* Constant shading
Floating point, 325, 326–327
Floating-point coordinates, 38–39
Flow curve, 1015
Fluorescence, 671
Flux responsivity, 792
Focal distance, 301
Focal points, 951
  and caustics, 798
Focus dot (camera manipulation), 586
Fog, 351
Fold set, 953. *See also* Contour(s)
Foreground image, 485
Form factor (radiosity), 840
  computation of, 842
Form factor (display), 58–59
Forward Euler integration, 1018
Forward Euler method, 1019. *See also* Explicit Euler method
Forward-rendering design, 440
Fourier-like synthesis, 559–560
Fourier transform, 497
  applications of, 522
  of box, 517
  definitions, 511
  examples of, 516–517
  of function on interval, 511–514
  inverse, 520–521
  properties of, 521
  scaling property of, 521
Fourth-order Runge-Kutta method, 1020
Fovea, 107
Fractional linear transformation, 256
Fragment (pixel), 18, 1055, 1056f
Fragment generation, 17
Fragments, 18
Fragment shaders, 466, 930. *See also* Pixel shaders
Fragment stage, 433
Framebuffers, 329, 971
  front buffer, 971

Frame coherence, 983. *See also* Temporal coherence
Frames (individual images), 963
Frequency-based synthesis, and analysis, 509–511
Frequency domain, 513
Fresnel, Augustin-Jean, 681
Fresnel equations, 681, 727–729
  unpolarized form of, 683
Fresnel reflectance, 683, 727–729
Fresnel's law, 681–683, 682f
  and polarization, 681–683
Frobenius norm, 663
Front buffer, 971, 975
Front face, of polygon, 337
Frontface polygon, 1048
Frustum clipping, 1028, 1045–1046
Frustum culling, 1023, 1028, 1044
Functional design (user interface), 570
Function classes, 505–507
Function *L*
  writing in different ways, 706–707
Functions, 151–152
  basis, 208
  interpolated, 203
  piecewise constant, 187
  tabulated, 201

G

G3D (open source graphics system), 241, 295, 321, 356, 933
Game application platforms, 478
Game engines, 25. *See also* Game application platforms
Gamma, defined, 771
Gamma correction, 769–771
  defined, 771
  encoding, 398, 769–771
Gamuts (color), 331, 766
  chromaticity diagram and, 766
  matching problem, 766
Gaussian filter, 542t, 543, 545
GDI, 38
Generalized cone, 757
General position assumption, 291–292, 292f
General purpose computing on GPUs (GPGPU), 1142
Generics in programming languages, 1068
Gentle slope interface, 569
Genus of a surface, 196
Geometric algebra, 284
Geometric curve extraction, 952–959
Geometric light, 124, 133
Geometric model, 2, 41ff, 117ff,
Geometric modeling, 595
Geometric objects, 93–94
Geometric optics, 726
Geometric shapes, 470–472
Geometry
  collision proxy, 337
  instancing, 349–350
  large-scale object, 337
  projective, 257
Geometry matrix, 597
Geometry processing, 458–460
Geometry shaders, 931
Geomorph, 649, 650f
GIF. *See* Graphics Interchange Format (GIF)
Gimbal lock, 269, 994
Glass
  BRDF and, 705–706
Global illumination, 340
Glossy highlights, 134, 353, 359–361
Glossy scattering, 414, 716
GLUT (OpenGL Utility Toolkit), 456
Gonioreflectometer, 702
Gouraud, Henri, 128
Gouraud shading, 128, 723, 743, 933. *See also* Phong shading
  fragment shader for, 937
  vertex shader for, 935
GPGPU (general purpose computing on GPUs), 1142
GPU. *See* Graphics Processing Units (GPUs)
GPU architectures, 1108–1111
  binned rendering and, 1137–1138
  deferred shading and, 1135–1137
  Larrabee (CPU/GPU hybrid), 1138–1142
  organizational alternatives of, 1135–1142
Grabcut (technology-enabled interface), 590–591
Gradient-domain painting, 961
Graftals (scene-graph elements), 986, 986f
Graphical user interfaces (GUI), 4, 23–24
  affordances, 572
  arcball interface, 584
  choosing the best, 587–588
  conceptual design, 570
  examples of, 588–591
  functional design, 570
  lexical design, 570
  multitouch, 574–580
  natural user interfaces (NUIs), 571
  sequencing design, 570
  suggestive interface, 589
  trackball interface, 580–584
Graphics applications, 21
  architectures of, 466–478
  kinds of, 24–25
Graphics data, 21–23
Graphics Interchange Format (GIF), 484
Graphics packages
  kinds of, 25–26, 451ff
Graphics pipeline, 14–15, 36, 119, 310, 452, 458ff, 927, 1109ff, 432–434
  defined, 434
  forms of, 927–929
  parts of, 17–18
  stages of, 16–19
Graphics platforms, 21, 22, 25, 26

Graphics Processing Units (GPUs), 18, 20
Graphics processor architecture, 8, 1107ff
Graphics program
  with shaders, 932–937
Graphics workstations, 932
Grays (color), 756
Grayscale, 482
Great circle, 273
Grid, as a class of spatial data structures, 1093–1101
  construction, 1093–1095
  ray intersection, 1095–1099, 1096f, 1098f
  selecting grid resolution, 1099–1101
Grid cells, 1093, 1093f. *See also* Buckets
Grid resolution, selecting, 1099–1101
GUI. *See* Graphical user interfaces (GUI)

H
Haar wavelets, 531
  transform, 531
Half-edges, 338
Half-open intervals, 150
Half-plane bounded by l, 174
Half-planes
  and triangles, 174–175
Half-vector, 721
Hand-animated cartoons, 966
Hash grid, 904, 1095
HDR images. *See* High dynamic range (HDR) images
Heat, defined, 672
Heat equation, 529
Heightfields, 344
Helmholtz reciprocity, 703–704, 714. *See also* Reciprocity
Hemicube, 842
Hemisphere
  cosine-weighted sampling on, 815
  producing a cosine-distributed random sample on, 815
  producing a uniformly distributed random sample on, 814
Hemisphere area light, 378–379
Hermite basis functions, 596
Hermite curve, 595–598
Hermite functions, 596
Heun's method, 1019–1020
Hidden surface removal, 1023. *See also* Visible surface determination
Hierarchical depth buffer, 1024, 1050. *See also* Hierarchical *z*-buffer
Hierarchical modeling, 35, 55, 313ff, 463–464
Hierarchical occlusion culling, 1049–1050
Hierarchical rasterization, 430
Hierarchical *z*-buffer, 1050. *See also* Hierarchical depth buffer
High-aspect-ratio triangles, 197
High dynamic range (HDR) images, 481
High-level design, 388–393
High-level vision, 105
Hit point (Unicam), 585
Homogeneous clip space, 429, 1047
Homogenization, 236, 254, 259
Homogenizing transformation, 265
Host code, 432
HoverCam (camera manipulator), 591
Hue (color description), 756
Hue-lightness-saturation (HLS) interface, 776–777
Hue-saturation-value (HSV) interface, 776–777
Human-computer interaction (HCI), 568
  arcball interface, 584
  interaction event handling, 573–574
  mouse-based camera manipulation (Unicam), 584–587
  mouse-based object manipulation in 3D, 580–584
  multitouch interaction for 2D manipulation, 574–580
  prescriptions in, 571–573
  suggestive interface, 589
  two-contact interaction, 578
Human visual perception, 101–115
Human visual system, 29–30
Hybrid pipeline era, 453
Hyperbolic depth encoding, 1038–1040
  complementary or reversed, 1039
Hyperbolic interpolation, 423

I
Identity matrix, 225
iid, *see* Independent dentically distributed random variables
Illuminant C (CIE chromaticity diagram), 765
Illumination, 9, 340, 362, 370ff, 722, 751, 785
Image
  choosing format of, 484–485
  composited, 485
  compositing, 485–490
  defined, 482
  enlarging, 534–537
  file formats, 483
  foreground, 485
  gradient, 544
  information stored in, 482–483
  losslessly compressed, 483
  meaning of pixel during compositing, 486
  Moiré patterns, 544
  other operations and efficiency, 541–544
  processing, 492–493
  representation and manipulation, 481–494
  RGB, 482
  scaling down, 537–538
  types of, 490–491
Image-based texture mapping, 559
Image display, 29

Image gradient, 544, 961
Images
  and signal processing, 493–532
Image space, 22, 245
Image-space photon mapping, 876
Immediate mode (IM), 452
  vs. retained mode (RM), 39–40
Implementation platform, 393–403
  and scene representation, 400–402
  and selection criteria, 393–395
  and test scene, 402–403
  utility classes, 395–400
Implicit curves, 616–619
Implicit functions, representing, 621–624
  mathematical models and, 623–624
Implicit lines, 164
Implicitly defined shapes, 164, 615–633
  advantages of, 615
  in animation, 631–632
  disadvantages of, 615
Implicit surfaces, 341–343, 619–620
  ray tracing, 631
  ray-tracing, 342–343
Importance function, 792, 819
Importance-sampled single-sample estimate theorem, 818–819
Importance sampling, 802, 818–820, 822, 854
  integration and, 818–820
  multiple, 820, 868–870
Impostors, 348
  and billboards, 347–348
Impulses, 356, 713, 784, 1010–1012. *See also* Snell-transmissive scattering
  deriving impulse equations, 1010–1011
  magnitude of, 740, 793
Impulse scattering, 715–716, 740, 784. *See also* Impulses; Snell-transmissive scattering
Incremental scanline rasterization, 431
Independent identically distributed (iid) random variables, 808
Independent random variables, 807
  properties for, 807
Indexed face sets, 77
Indexed triangle meshes, 338
Indexing arrays, 156
Indexing vectors, 156
Index of refraction, 107, 332. *See also* Refractive index
Indication (expressive rendering), 948
Indirect light, 370. *See also* Direct light
Indirect lighting, 834
Infinite series
  summing by sampling and estimation, 829–830
Infinite support, 535
Information visualization, 4, 37
Inheritance, as key extraction method, 1073–1074
Initialization in OpenGL, 456–458
Injective function, 151. *See also* Surjective function
Inner product, 158
Inscattering, 738
Inside/outside testing, 175–177
Instance transform, 139–140
Instancing, described, 450
Instantiated templates, 39, 50
Integral, of spectral radiance, 692
Integral equation, 786
Integration
  importance sampling and, 818–820
Intel Core 2 Extreme QX9770 CPU, 1138
Intensity (light), 700, 769–771
  encoding of, 769–771
  high-light perception of, 770
  low-light perception of, 770
Intensity-independent colors, 765
Interaction, keyboard, 95
Interface, 434–444
Interior edge, 637
Interiors of nonsimple polygons, 177
Interior vertices, 194, 638
Interlaced television broadcast and storage formats, 978
Interlacing, 978–980
  pulldown, 979–980
  telecine, 978–980
International Color Consortium (ICC), 772
Interpolated function
  properties of, 203
Interpolated shading (Gouraud), 128–129
Interpolating curve, 600
Interpolation
  bilinear, 622
  hyperbolic, 423
  perspective-correct, 256, 312, 422–424
  precision for incremental, 427–428
  rational linear, 423
  between rotations, 276–278, 277f
  spherical linear, 275–276
  vs. transformations, 259
Interpolation schemes, 621–622
  bilinear interpolation, 622
Intersections, 167–171
  of lines, 165–167
  ray-plane, 168–170
  ray-sphere, 170–171
Interval
  Fourier transform on, 518
Invariant under affine transformations, 182
Inverse Fourier transform, 520–521
Inverse function, 151
Inverse tangent functions, 152–153
Invertibility, as a property of texture mapping, 555
Inward edge normal, 175
Irradiance, 697–699. *See also* Radiosity
  defined, 697

Irradiance due to a single source, 698
Irradiance map, 557
Isocontour, 341
Isocurves, 616
Isosurfaces, 619. *See also* Level surfaces

J
Jaggies (image), 31
Joint transform, 140, 146–147
Just noticeable difference (JND), 754

K
k-dimensional structures, 1080–1081
*k*d tree, 1089
Kernel (photon map), 910
  cylinder, 910–911
Kernel (photon mapping), 874, 1142
Key (data structure), 1068, 1077
Keyboard interaction, 95
Key frame, 966, 989. *See also* Key pose
Key pose, 966, 989. *See also* Key frame
Keys and bounds, extracting, 1073–1077
  inheritance, use of, 1073–1074
  traits, 1074–1077
Kinect (interface device), 568
Kinetic energy, 1021
Knee joint
  adding, 143–144
Knots, 601
Kubelka-Munk coloring model, 760

L
L^2 difference. *See* L^2 distance
L^2 distance, 104
L^2 (space of functions), 506ff
ℓ^2 (space of sequences), 506ff
"Lab" color, 767
Lafortune model (light scattering model), 723–724
Lag. *See* Latency
Lambertian, 28, 358–359
Lambertian bidirectional reflectance distribution function (BRDF), 720, 883
Lambertian emitter, 695
Lambertian luminaire, 785
Lambertian reflectance, 28, 358, 413, 720, 925
Lambertian reflectors, 719–721
Lambertian scattering, 413–414, 716, 725
Lambertian shading, 353
Lambertian wall paint, 708
Lambert's Law, 358
Laplacian coordinates, 655–657
  applications, 657–660
  properties of, 657
Large-scale object geometry, 337
Larrabee (CPU/GPU hybrid), 1138–1142, 1139f
  cache coherence, 1140
  capability of, 1140
  correct provisioning, 1141
  efficient parallelization, 1141
  flexibility in, 1140
  vs. GeForce 9800 GTX, 1140
  generality in, 1140
  Intel's IA-32 instruction set architecture (ISA), use of, 1140
  latency hiding, 1140
  multiple processing cores, 1139
  sequence optimization, 1141
  specialized, fixed-function hardware, 1139
  SPMD and, 1140
  texture evaluation, 1139
  wide vectors, 1139
Latency, 17, 1123–1126
Lateral inhibition, 108
Law of conservation linear momentum, 1011
Layout, defined, 85
LCD. *See* Liquid-crystal displays (LCDs)
LED-based interior lighting, 752
Legacy models, 324
Lemniscate of Bernoulli, defined, 616f
Lens flare and bloom, 366, 369
Level of detail (LOD), 347
Level set, 164, 341, 616
Level set methods, 631
Levels of detail (geometric representations), 645–649
  determining, 645–646
  parametric curves and, 649
  surfaces and, 649
Level surfaces, 619. *See also* Isosurfaces
Lexical design (user interface), 570
Light(s), 26, 330–333, 784
  ambient, 122, 124
  area, 888
  bending of, at an interface, 679–680
  defined, 669
  direct, 835, 865
  directional, 125
  elliptically polarized, 679
  excitation purity and, 747
  geometric, 124
  hemisphere area light, 378–379
  indirect, 835, 865
  infrared, 672
  interaction with objects, 118–119
  interaction with participating media, 737–738
  metameric, 768
  point, 886–887, 888–889
  representation of, 887–889
  light capture, 29
  measuring, 692–699
  modeling as a continuous flow, 683–692
  monospectral, 747
  omni-light, 379–380
  other measurements of, 700
  path, 796
  physical properties of, 669–670

quantized, 670
rectangular area light, 377–378
scattering, 388–390
spectral distribution of, 746–748
transport, 335–336, 783–787
ultraviolet, 671
unpolarized, 683
wavelength of, 670
wavelike, 670
wave nature of, 674–677
Light energy
and photon arrival rates, 12–13
Light geometry, 133
Lighting, 312
direct, 834, 913
indirect, 834
and materials, 458
Phong, 930
programmable, 930
vs. shading in fixed-function rendering, 127–128
Lighting specification, 120–128
Light maps, 341
Lightness, 755, 756
CIE definition of, 755
Light path, 796
Light transport, 783–787
alternative formulations of, 846–847
boundaries and, 798
caustics and, 798
classification of paths, 796–799
Metropolis, 871–872, 915
perceptually significant phenomena and, 797–799
polarization and, 798
shadow and, 797
symbols used in, 784t
transport equation, 786–787
Light-transport paths
classification of, 796–799
Linear combination, 157
Linear depth, 1040
Linear depth encoding, 1040
Linear interpolation, 201
Linearly polarized electric field, 678
Linear radiance, 398
Linear transformation, 221, 259, 307
degenerate (or singular) transformation, 224
examples of, 222–224
multiplication by a matrix as, 224
nonuniform scaling, 223
properties of, 224–233
rotation, 222–223
shearing, 223
Linear waves, 675
Linear *z,* 1040
Line of curvature, 956, 956f
Lines, intersections of, 165–167
Linked list (data structure), 1077
1D example, 1078–1079
Link of a vertex, 208, 641
Liquid-crystal displays (LCDs), 20
List, as a class of spatial data structures, 1081–1083, 1082f
C++ implementation of conservative ball-primitive intersection in, 1083
C++ implementation of ray-primitive intersection in, 1083
unsorted 1D list, 1081
List-priority algorithms, 1040–1043
BSP sort, 1043
clusters, 1043
depth-sort algorithm, 1042–1043
painter's algorithm, 1041–1042
Live Paint, 1042
Local flatness (surface), 643, 882
Local Layering, 1042
LOD. *See* Level of detail (LOD)
Look vectors, 304
LookDirection, 122
Losslessly compressed image, 483
Lossy compression, 471, 483
Low-level vision, 105
Lucasfilm, 932
Luma, 771, 775
Lumens, 707
Luminaire models, 369
and direct and indirect light, 370
and nonphysical tools, 371–372
practical and artistic considerations of, 370–377
and radiance function, 370
Luminaires, 369, 784
area lights, 888
computer graphics, 369
representation of, 888–889
Lambertian, 785
point lights, 888–889
representation of, 888–889
Luminance, 707, 747, 755
of light source, 751
signal representative of, 770–771
Luminous efficiency, 707
Luminous efficiency function, 751
Luminous intensity, 751

M

Mach banding, 20, 211
Macintosh, 568
Magnitude, of impulses, 740, 793
Manifold meshes, 190, 191, 193–195
2D mesh as, 193
boundaries, 194
orientation of triangles in, 193–194
Manifold-with-boundary meshes, 195
operations on, 195
Mappings
application examples of, 557t
cylindrical, 555

Mappings (*continued*)
examples of, 555–556
reflection, 556
spherical, 555
Marching cubes, 625, 628–629
algorithm, 628
extended, 653
generalization of, 652–653
variants, 652–654
Marching squares, 627
Hermite version of, 653
Markov chains, 857–861
estimating matrix entries with, 858–859, 860–861
Metropolis light transport, 871–872
path tracing and, 856–857
Markov property, 857
Marks (expressive rendering), 950–951
creation of, 951
imitation of artistic technique for creating, 951
physical simulation, 951
scanning/photography approach for creating, 951
Mask, 485–486
Masking, 730
Master templates, 39
Material, in scattering, 712
Material models, 353–358
software interface to, 740–741
Materials
lighting and, 458
Mathematical model, 2, 11. *See also* Geometric model; Numerical model; Physical models
and sampled implicit function representations, 623–624
Mathematics, and computer graphics, 30–31
Matrix associated to a transformation, 224
Matrix/matrices, 156
diagonal, 230
identity, 225
invertible, 225
orthogonal, 230
properties of, 230–231
rank of, 231, 261
rotation, 270–272
singular value decomposition (SVD) of, 230
special orthogonal, 230
Matrix multiplication, 161–162
Matrix transformations, 222
interpolating, 280
Matter, 336
Matting problem, 367
MaxBounce (photon mapping), 873
McCloud, Scott, 947
Mean. *See* Expected value
Measured BSDFs, 358
Measured/captured models, of scattering, 713, 725–726
Measurement,
and sampling, 507
value of, 323–324
Measurement equation, 791–792
Measure of a solid angle, 687
Megakernel tracing, 1033
Memoization (component of dynamic programming technique), 983
Memory practice, 435–437
Memory principles, 434–435, 1117ff
Mesh(es), 338–341
adjacency information on, 338–339
alternative mesh structures, 338
applications, 652–667
beautification, 197
cached and precomputed information on, 340–341
closed, 190, 642
connected unoriented, 639
differential coordinates for, 657
embedding topology for, 637
functions on, 201–220
geometry, 643–644
icosahedral, 648
indexed triangle, 338
Laplacian coordinates for, 657
manifold, 191
manifold-with-boundary meshes, 195
meaning of, 644–645
nonmanifold, 195–196
nontriangle, 637
operations, 197
orientation of triangles in, 193–194
oriented, 191, 639–640
other simplification approaches, 652
per-vertex properties and, 339–340
polygonal, 953
progressive, 649–652
quad, 611, 611f
repair, 654–655
simplices, 208
simplification, 188, 197
subdivision of, 211
terminology, 641
terminology for, 208
topology of, 189, 637–643
triangle, 187, 187f, 188f
unoriented, 191
winged edge polyhedral representation and, 338
Mesh beautification, 197
Mesh flattening, 667
Mesh geometry, 643–644
Mesh Laplacians, 656
Mesh operations, 197
edge collapse, 197
edge-swap, 197–198
mesh beautification, 197
mesh simplification, 197

Mesh repair, 654–655
Mesh specification, 120–128
Mesh structures, 211
  memory requirements for, 196–197
Mesh topology, 637–643
Metaball modeling, 343
Metadata, 483
Metameric lights, 768
Metamers, 768
Metropolis light transport, 871–872, 915
Microfacets, 729
Microgeometry, 901
Micropolygon rasterization, 431–432
Micropolygons, 340, 431
Microsoft Office Picture Manager, 569
Minecraft, 964
MIP maps, 217, 491–492, 1120
Mirrors
  BRDF of, 705–706
  and point lights, 886–887
Mirror scattering, 715, 717–719
Mitchell-Netravali filter, 540f
Mixed probabilities, 820–821
Model, defined, 2
Modeling, defined, 2
Modeling space, 21
Modeling stage, 460
Modeling transformation, 51
Modelview, 314
Modelview matrix, 463
Modelview projection matrix, 314
Modified Euler method, 1020
Modular modeling
  motivation for, 138–139
Modulus, of complex numbers, 513
Moiré patterns (image), 544
Monet painting, 948
Monospectral distributions, 747
Monte Carlo approaches, 851–854
  bidirectional path tracing, 853
  classic ray tracing, 851–852
  distribution ray tracing, 838
  path tracing, 853
  photon mapping, 853–854, 872–876
Monte Carlo integration, 783, 796, 854
Monte Carlo rendering, 786, 922
Moore's Law, 8, 932
Morphogens, 561
Motion
  methods for creating, 966–975
  perception, 976–978
  root, 969
Motion blur, 980–983, 1061–1062
  temporal aliasing and, 980–983
Motion-blur rendering, 922
Motion-induced blindness, 114
Motion perception, 976–978
  Beta phenomenon, 977
  strobing, 977
Motion planning, 972–973
Mouse-based object manipulation in 3D, 580–584
  arcball interface, 584
  trackball interface, 580–584
MSAA. *See* Multi-sample anti-aliasing (MSAA)
Multipass rendering, 441
Multiple importance sampling, 820, 868–870
Multiresolution geometry, 471
Multisample antialiasing (MSAA), 433, 1057–1058, 1136
  advantages of, 1058
  drawbacks of, 1057–1058
Multitouch interaction for 2D manipulation, 574–580
Munsell color-order system, 762
Mutation strategy, 871
Mutually perpendicular vectors, 229, 240

N

Natural user interfaces (NUIs), 571
Nearest neighbor (density estimation), 912
Nearest-neighbor field, 564
Nearest-neighbor strategy (animation), 967
Near-plane clipping, 1044, 1046
Negative nodes (BSP tree), 1031
Neighborhood (subdivision surfaces), 610
Neighbor-list table, 191
Nit (photometric term), 751
Nonconvex spaces, 211–213
Nonmanifold meshes, 195–196
Nonphotorealistic camera, 3
Nonphotorealistic rendering (NPR), 945, 986
Nonphysical tools, 371–372
Nonspectral colors, 766
Nonspectral radiant exitance, 700. *See also* Radiosity
Nonuniform B-spline, 604
Nonuniform Catmull-Rom spline, 601–602
Nonuniform rational B-spline, 604
  advantages of, 604
  CAD systems and, 604
Nonuniform scale. *See* Nonuniform scaling transformation
Nonuniform scaling, as linear transformation in the plane, 223
Nonuniform scaling transformation, 223
Nonuniform spatial distribution, 1100
Nonzero winding number rule, 177
Norm, of a vector, 157
Normal. *See* Normal vectors
Normalization process, 72
Normalized Blinn-Phong, 359–361
Normalized device coordinates, 22, 72. *See also* Camera-space coordinates
Normalized fixed point, 325
Normalizing vector, 157
Normal maps, 647
Normal transform. *See* Covectors, transforming

Normal vectors, 16, 27–28, 164, 193–194
Notation, mathematical, 150
Numbers
  and orders of magnitude in graphics, 12–15
Numerical integration, 29, 801–802
  deterministic method, 801–802, 804
  probabilistic or Monte Carlo method, 802
Numerical methods for solving ODEs, 1017–1020
Numerical model, 11. *See also* Mathematical model; Physical models
NURB. *See* Nonuniform rational B-spline
NVIDIA GeForce 9800 GTX GPU, 1138
Nyquist frequency, 515
Nyquist rate, 544

O
Object-centered rotation, 272. *See also* Body-centered rotation
Object coordinates, 140, 245. *See also* Object space
Object ID channel, 485
Object-level scattering, 711–712
Object-oriented API, 41
Objects
  detail, 1051
  interaction with light, 118–119
  and materials, 27–28
Object space, 245. *See also* Modeling space; Object coordinates
Occlusion, 308, 1023ff
  in 2.5D systems, 1042
Occlusion culling, 1023, 1049
  hierarchical, 1049–1050
Occlusion function, 1025
Occlusion query, 1049
Oct tree, 629, 1090
Odd winding number rule, 177
OLED. *See* Organic light-emitting diodes (OLEDs)
Omnidirectional point light
Omni-light, 379–380
One-dimensional (1D) meshes, 189
  boundary of, 190
  data structure for, 191–192
  manifoldmesh, 190
  1-ring (vertices), 641
OpenGL, 452
  compatibility (fixed-function) profile, 454–455
  core API, 466
  programmable pipeline, 464–466
  program structure, 455–456
  utility toolkit, 456
OpenGL ES, 479
OpenGLUtilityToolkit. *See* GLUT(OpenGLUtilityToolkit)
Optic disk, 107
Optimization
  early, 446–447
  and performance, 444–447
Ordinary differential equation (ODE), 998
  general ODE solver, 1016–1017
  numerical methods for, 1017–1020
Oren-Nayar model, 732–734
Organic light-emitting diodes (OLED), 20
Orientation-preserving reflection, 264, 284
Oriented 2D meshes, 194–195
  boundaries and, 194–195
Oriented meshes, 191
  closed, 195
Oriented simplex, 639
Orthogonal matrix, 230
Orthographic cameras, 315–317
Orthographic projections. *See* Parallel projections
Outer product (matrices), 260
Output merging stage, 433–434
Output-sensitive time cost, 1079
Outscattering, 738
Outside/inside testing, 175–177
Outward edge normal, 175
Over operator, 365

P
Packet tracing, 1061
Painter's algorithm, 1028, 1041–1042
Panning (camera controlling technique), 585
Pantone™ color-matching system, 761
Paraboloid, dual, 554
Parallel projections, 315–316
Parameterization
  building tangent vectors from, 552–553
  of lines, 155
  texture, 555
  of triangles, 171
Parameterized model, 76
Parametric equation, 341
Parametric form of the line between P and Q, 155
Parametric-implicit line intersection, 167
Parametric lines, transforming, 254
Parametric-parametric line intersection, 166
Partial coverage, 364–367, 365, 1028, 1054–1062
  defocus, 1060–1061
  as a material property, 1062
  motion blur, 1061–1062
  spatial antialiasing, 1055–1060
Participating media, 737–738
Particle collision detection, 1008–1009
Particle collisions, 1008–1012
  collision detection, 1008–1009
  impulses, 1010–1012
  normal forces through transient constraints, 1009
  penalty forces, 1009–1010
Particle systems, 350–351
Path mesh, 668

Path tracer
basic, 889–904
building, 864–868
code, 893–901
core procedure in, 893
Kajiya-style, 866, 867
preliminaries, 889–893
symbols used in, 890
Path-tracer code, 893–901
Path tracing, 853, 853f, 855
algorithmic drawbacks to, 855–856
bidirectional, 853, 870–871
constructing a photon map via, 873
and Markov chains, 856–857
path tracer, building, 864–868
Pdf. *See* Probability density function (pdf)
Pen (geometric attribute), 38
Penalty force
application of, 1009–1010
computation of, 1009–1010
defined, 1009
Pencil of rays, 1060
Penumbra, 505, 798
Perception, human visual. *See* Human visual perception
Perceptual color spaces, 767–768
Peripheral color perception, 781–782
Perlin noise, 560–561, 561f
Perspective camera specification, 301–303
Perspective-correct interpolation, 422–424
Per-vertex properties
of meshes, 339–340
Phong exponent, 736
Phong lighting, 930
Phong lighting equation, 938
Phong model (reflection model), 721–723
Phong reflectance (lighting) model, 134
Phong shader, 937–939
Phong shading, 723. *See also* Gouraud shading
fragment shader for, 938–939, 940
vertex shader for, 938
Phosphorescence, 671
Photography
computational, 493
Photometry, 670, 700
Photon(s), 369, 670
defined, 872
photon-mapping, 872
propagation, 907–908
Photon arrival rates
and light energy, 12–13
Photon emission, 376–377
Photon map, 872, 908, 913
constructing via photon tracing, 873
Photon mapping, 853–854, 872–876, 904–914
bias in, 875
consistency in, 875
density estimation and, 912
final gather step in, 913
image-space, 876
limitations, 875
main parameters of, 873
phases of, 872
schematic representation of, 853f
Photon propagation, 907–908
Photon tracing, 872
Photopic vision, 753–754
Photorealism, 945
Photorealistic rendering, 31
Physical coordinate system, 38
Physically based animation, 963, 989
Physically based models, of scattering, 713, 727–734
Physical models, 11. *See also* Numerical model; Physical models
Physical optics, 726
Physical units
and compositing, 489–490
Physics scene graphs, 352–353
Pick correlation, 39, 60, 139, 464
Pick path, 464
Piecewise constant function, 187
Piecewise linear extension, 210
animation, use in, 211
defined, 210
limitations of, 210–211
mesh structure and, 211
Piecewise linear reconstruction, 505
Piecewise-smooth curves, 540
Pie menus, 573, 573f
Pitching, 267
Pixar, 932
Pixel coordinates, 22
Pixel program, 433
Pixels, 5
defined, 29
Pixel shader. *See* Pixel program
Pixel shaders, 930. *See also* Fragment shaders
Pixel stages
and vertex stage, 433
Pixel values, 482
Pixmaps. *See* Bitmaps
Planar wave, 676
Planck, Max, 674
Planck's constant, 12, 670, 674
Plenoptic function, 370, 693
PNG. *See* Portable Network Graphics (PNG)
Point lights, 124–125, 133, 886–889
computing direct lighting from, 894
mirrors and, 886–887
reflecting illumination from, 894
Points, 234–235, 288
Point sets, 345–346
Poisson disk process, 921
Polarization, 670, 677–679
circular, 678f
Fresnel's law and, 681–683
linear, 678f

Polarizers, 679
Polling (interaction loop), 574
Polygonal contour extraction, 955
Polygon coordinates, 249. *See also* Modeling coordinates
Polygon meshes, 635
Polygons, 175–182
  back face, 337
  drawing as black box, 23
  front face, 337
  interiors of nonsimple, 177
  micropolygon, 340
  normal to polygon in space, 178–179
  polygon rate, 14
  simple, 175
Polygon soup, 654
Polyhedral manifolds, 191
Polyhedral meshes
  conversion to, 625–629
  conversion to implicits, 629
Polyline, 81
Polymorphic types, 1068
Polymorphism, 1073
"Poor man's Fourier series," 560
Poor sample test efficiency, 420
Popping, 985
Portable Network Graphics (PNG), 484
Portal culling, 470
Portals, 1051, 1052–1054
Positive nodes (BSP tree), 1031
Positive winding number rule, 177
Potential energy, 1021
Potentially visible set (PVS), of primitives, 1050
Power vectors, 875
p-polarized light source, 681
p-polarized waves, 681
Practical lights, 372
Prebaking (models), 247
Premultiplied alpha, 366–367, 487
  problem with, 489
Presentation vector graphics, 1042
Primary colors, 758–759
Primary ray, 1027
Primary visibility, 1023, 1024, 1027
Primitive components
  geometries of, 140–141
  instantiating, 141
Primitives, 6, 38, 461–462, 962, 969, 973, 1028, 1030–1031, 1041–1042, 1044, 1059–1060, 1084–1085, 1087–1092
Primitives per second, 6
Principal curvatures, 956
Principal directions, 956
Probability
  of an event, 803
  continuum, 808–810
  mixed, 820–821
Probability density, 684, 686
Probability density functions (pdf), 684, 808, 810–812
Probability masses, 685, 820
Probability mass function (pmf), 803, 805
Probability of an event, 809
Procedural code, 35, 55, 58
  vs. declarative specification, 40
  dynamics via, 58
Procedural texturing, 549
Programmable graphics card, 8. *See also* GPU.
Programmable lighting, 930. *See also* Programmable shading
Programmable pipeline, 433, 453–454
  abstract view of, 464–466
  OpenGL, 464–466
Programmable rendering pipeline
  fixed-function to, 452–454
Programmable shading, 930, 932. *See also* Programmable lighting
Programmable units, 433
Programmatic interfaces, 1068–1077
Programmer instruction batching, 1033
Programmer's model, 17, 454–464
Progressive meshes, 649–652
  goal of, 649
Progressive refinement (radiosity), 843
Progressive television formats, 978
Projected solid angle, 690
Projection stage, 460
Projection textures, 555, 629
Projective frame, 265
Projective geometry, 77, 257
Projective transformations, 255, 257, 259–260, 263, 291–293, 308
  general position, 291–292, 292f
  properties of, 265–266
Projective transformation theorems, 265–266
Propagation, 332–333
Proxies (data structure), 1068
Pseudoinverse
  defined, 232
  least squares problems and, 232
  SVD and, 231–233
Pseudoinverse Theorem, 232
Pulldown (interlacing), 979–980

Q

Quad fragments, 1137
Quad meshes, 611, 611f, 635
Quadratic error function, 653
Quad trees, 1090, 1091f
Quake video game, 1052
Quantitative invisibility, 1028
Quaternions, 273, 283, 993
QuickDraw, 38

R

Radial curvature, 957
Radiance, 333, 397, 693, 694–695
  area-weighted, 910

emitted, 785, 832
field, 786, 834, 846
impulse-reflected, 794
linear, 398
surface, 786–787, 834, 846
Radiance computations, 683, 695–697
Radiance function, 370
Radiance propagation, 907–908
Radiant exitance, 699
spectral, 699
Radiant flux, 699. *See also* Radiant power
Radiant power, 699. *See also* Radiant flux
Radiometry, 669, 694
Radiosity, 333, 700, 797, 838–844. *See also* Nonspectral radiant exitance
characteristics, 838
color bleeding and, 839
as finite element method, 839
meshing in, 843
Radiosity equation, 840
Randomized algorithms
random variables and, 802–815
Random parametric filtering (RPF), 922
Random point, 812
Random variable(s)
in continuum probability, 809
defined, 803
estimator, 818
expected value of, 810
identically distributed, 808
independent, 807
independent identically distributed (iid), 808
and randomized algorithms, 802–815
random point and, 812
uniform, 807
Random variable with mixed probability, 820
Raster devices, 8
Raster graphics, 209
history of, 931–932
Rasterization, 18, 418–432, 1027, 1061
conservative, 1096
defined, 387, 391
hierarchical, 430
and high-level design, 388–393
incremental scanline, 431
micropolygon, 431–432
and ray casting, 387–449
rendering with API of, 432–434
swapping loops, 418–419
triangles first, 391–393
Rasterizer algorithm, 418
Rasterizing shadows, 428–429
Rasterizing stage, 433
Rasters, 391, 978, 979
Rational B-spline, 604
Rational numbers, 325
Ray bumping, 886, 1027
Ray casting, 1023, 1028, 1029–1034
defined, 387, 391
implementation platform and, 393–403
pixels first, 391–393
and rasterization, 387–449
renderer, 403–404
and sampling framework, 407–408
time cost of, 1029
Ray intersection, 1095–1099
Ray intersection query, 1026
Ray optics
building blocks of, 330
Ray packet, 445
Ray packet tracing, 1027, 1033
Ray-plane intersection, 168–170
Ray-sphere intersection, 170–171
Ray tests, parallel evaluation of, 1032–1034
Ray tracer
steps involved in, 929
Ray-tracing
defined, 391
recursive, 851–852
implicit surfaces, 342–343, 631
Ray-triangle intersection, 408–411, 1073
Reaction (morphogens), 561
Ready for rendering
vs. abstract geometric, 467
Realistic lighting
producing, 124–127
Realistic rendering, building blocks for, 26–31
Real numbers, 324–325
Real-time 3D graphics platforms, 351–480
introduction, 351–352
Reciprocity, 714. *See also* Helmholtz reciprocity
and BRDF, 705–706
Reconstruction, 505
and band limiting, 524–527
piecewise linear, 505
Rectangular area light, 377–378
Recursive approach, 861–864
Reencoding, 470–471
Reference frame, 963
Reference renderer, 388
Reflectance, 133–138, 702–704, 711ff
ambient reflection, 136
diffuse reflection, 136
emissive lighting, 138
phong reflectance (lighting) model, 134
specular reflection, 137–138
WPF reflectance model, 133–138
Reflectance equation, 703, 786
Reflection mapping, 556
Reflection model, 723. *See also* Scattering model
Reflective scattering, 697, 715
Reflective surface, 27
Refractive index, 679
Refractive scattering, 716
Rejection sampling, 823

Rendering, 6
  animation and, 975–987
  binned, 1137–1138
  debugging and, 915–919
  double-buffered, 971, 975–976
  expressive, 945–962
  intersection queries in graphics that arise from, 1066–1067
  Monte Carlo, 922
  motion blur, 980–983
  motion-blur, 922
  nonphotorealistic, 945, 986
  pen-and-ink, 950
  photorealistic, 31
  stroke-based, 955
  temporal aliasing, 980–983
Rendering equation, 373–376, 703, 783, 786–787, 831–836
  approximating, 826
  approximations of the series solution of, 847–848
  discretization approach, 838–844
  domain restriction, 827
  for general scattering, 789–792
  Markov chain approach for solving, 857–861
  methods for solving, 825–831
  Monte Carlo approaches for solving, 851–854
  recursive approach for solving, 861–864
  series solution of, 844–846
  simplifying, 840
  spherical harmonics approach, 848–851
Replication
  in spectrum, 523–524
Representations
  comparing, 278–279
  evaluating, 322–323
  of implicit functions, 624–625
  of light, 887–888
  and standard approximations, 321–384
  surface, 882–887
  triangle fan, 338
Resolution, 8
  eye's resolution, 13
Resolution dependence, 38
Resolved framebuffer, 1056, 1056f
Restricted transformations, 295–297
  advantages of, 295
  disadvantages of, 295
Retained mode (RM), 452
  vs. immediate mode (IM), 39–40
Retina, 107
Retroreflective scattering, 716
Reuse of components, 144–147
Reyes micropolygon rendering algorithm, 982–983
RGB color cube, 772–773
RGB color model, 772–774
RGB format, 481
RGB image, 482
Ridges, 956–957
  apparent, 958–959
Right-handed coordinate system, 158
Rigid coordinate frame, 240
Ringing, 510f, 529, 538
Rodrigues' formula, 293
Rods (color receptors), 107, 749
  saturated, 755
Rolling, defined, 267
Root frame, 972
Root frame animation, 972
Root motion, 969
Rotation, as linear transformation in the plane, 222–223
Rotation about z by the an angle, 266
Rotation around z, 239. *See also* Rotation in the xy-plane
Rotation by an angle in the xy-plane of 3-space, 266
Rotation in the xy-plane, 239. *See also* Rotation around z
Rotation matrix, 270–272
  finding an axis and angle from, 270–272
Rotation(s), 264, 266
  3-sphere and, 273–278
  axis-angle description of, 269–270
  interpolating between, 276–278
  vs. rotation specifications, 279–280
Rule of five, 698, 925
Russian roulette, 874

S

$\mathcal{S}$ (normalizing vectors), definition, 157
Sahl, Ibn, 679
Sample-and-hold strategy (animation), 967
Sample and hold reconstruction, 505
Samples (implicit functions), 621
Samples (pixel), 1055, 1056f
Sample shaders, 930
Sampling, 29, 507–508, 557–559, 724–725
  approximation of, 519
  and band limiting in interval, 514–515
  cosine-weighted, 815
  importance, 818–820, 854
  integration and, 31
  multiple importance, 868–870
  rejection, 823
  stratified, 920
  summing a series by, 828–830
Sampling framework, 407–408
Sampling strategy, 854
Sampling theorem, 515
Scalability, 469
Scalar attributes, 651
Scalar multiplication, 157, 158
Scale invariance, 911
Scale transformations, 263–264
Scanline interpolation, 208–210
Scanline rendering, 209

Scanners, 5, 5f
Scattering, 711, 792–793
  approximating, 848–851
  diffuse, 713, 716
  diffuselike, 792
  due to transmission, 900
  equation, 790
  glossy, 716
  impulse, 715–716, 784
  kinds of, 714–717
  Lambertian, 716
  of light, 388–390
  mirror, 715, 717–719
  models, 713
  nonspecular, 852
  object-level, 711–712
  physical constraints on, 713–714
  reflective, 715
  refractive, 716
  rendering equation for, 789–792
  retroreflective, 716
  Snell-transmissive, 783–784
  subsurface, 720, 738–739
  surface, 712–714
  transmissive, 715
  volumetric, 737, 793
Scattering equation, 790
Scattering functions, 354–358
Scattering models, 723. *See also* Reflection model
  Blinn-Phong model, 721–723
  Cook-Torrance model, 731–732
  empirical/phenomenological models, 713, 717–725
  Lafortune model, 723–724
  measured/captured models, 713
  Oren-Nayar model, 732–734
  Phong model, 721–723
  physically based models, 713, 727–734
  Torrance-Sparrow model, 729–731
  types of, 713
  wave theory models, 734–736
Scatters, light, 333
Scene, 21, 31, 37
  abstract coordinate system to specify, 42–44
  planning, 120–124
  reduction of complexity, 469
Scene generator, 37
Scene graphs, 39, 118, 351–353
  coordinate changes in, 248–250
  hierarchical modeling using, 138–147
  physics, 352–353
Scene modeling, 945
Scene representation, 400–402
Schematization (abstraction), 947
Schlick approximation, 728–729
Scotopic vision, 753–754
Screen door effect, 986
Screen space, 245
Screen tearing, 976
Second-order Runge-Kutta methods, 1019
Sector (polyhedron), 1050
Sector-based conservative visibility, 1050–1054
  mirrors, 1052–1054
  portals, 1052–1054
  stabbing line, 1051
  stabbing tree, 1051–1052
Sector-based culling, 470
Segments (of a spline), 599
Self-shadowing, 1027. *See also* Shadow acne
Semantic element, 352
Semi-Explicit Euler method. *See* Semi-Implicit Euler method
Semi-Implicit Euler method, 1019
Sensor response, 791
Separable filter, 544
Sequencing design (user interface), 570
Sets, 150
Shaders, 8, 453. *See also* Programs
  creating, 437–442
  defined, 928
  extensibility via, 453
  fragment, 466, 930
  geometry, 931
  historical development, 929–932
  Phong, 937–939
  pixel, 930
  sample, 930
  in scattering model, 723
  simple graphics program with, 932–937
  subdivision surface, 931
  tessellation, 931
  vertex, 930, 931
Shader-specified visibility, 1136
Shader wrapper, 933
  G3D, 933
Shades (color), 756
Shading, 412–413, 723, 1055
  deferred, 1135–1137
  interpolated, 128–129
  vs. lighting in fixed-function rendering, 127–128
  two-tone, 959
Shading language, 927
Shading normals, 339
Shadow, and light transport, 797
Shadow acne, 325, 416, 1027. *See also* Self-shadowing
Shadow map, 428, 848
Shadow mapping, 557t
Shadows, 112–113, 414–417
  acne, 325, 416, 1027
  applications of, 113
  curvature, 946
  direct, 946
Shannon sampling theorem, 515
Shape constancy, 110
Sharp edges, 651

Sharpening, 543, 545
Shearing transformations, 264
  as linear transformation in the plane, 223
Shift-invariant, 529
Shutter time, 980. *See also* Exposure time
SIGGRAPH (Special Interest Group on GRAPHics and Interactive Techniques), 4, 922
Signal processing, 500
  and images, 493–532
Signal, 500
Signed area
  of a plane polygon, 177–178
Signed distance transform of the a mesh, 629
Signed normalized fixed-point 8-bit representation, 551
Silhouette, 952
Silicon Graphics, Inc., 931–932
SIMD. *See* Single Instruction Multiple Data (SIMD)
Simple polygons, 175
Simplex, 208
  boundary of, 208
  categories of, 208
  oriented, 639
Simplicial complices, 198
Simplification, 471
  abstraction, 947
  of triangle meshes, 188, 649
Single Instruction Multiple Data (SIMD), 430, 1033
Singular transformation. *See* Degenerate transformation
Singular value decomposition (SVD), 230
  computing, 231
  matrix properties and, 230–231
  and pseudoinverses, 231–233
Singular values of matrix M, 230
Size constancy, 110
Skyboxes, 348–349
Sky sphere. *See* Skyboxes
Slerp, 275. *See also* Spherical linear interpolation
Slicing, 935
Smith, Alvy Ray, 498
Smooth edges, 953
Smooth manifolds, 190
Snell's law, 679, 683, 728
Snell-transmissive scattering, 783–784. *See also* Impulses
Soft particles, 351
Software-platform independence, 44
Software stack, 468
Solid angles, 370, 686–688
  computations with, 688–690
  measure of, 687
  projected, 690
  subtended, 687
Source (texture image), 563
Source polygon, 1045
Spatial acceleration data structures. *See* Spatial data structures
Spatial antialiasing, 1055–1060
  A-buffer, 1057
  analytic coverage, 1059–1060
  coverage sampling antialiasing (CSAA), 1058–1059
  multisample antialiasing (MSAA), 1057–1058
  supersampled antialiasing (SSAA), 1056–1057
Spatial coherence, 950
Spatial data structures, 353, 1023, 1065–1102
  characterizing, 1077–1079
  extracting keys and bounds, 1073–1077
  generic use of, 1077
  grid, 1093–1101
  hash grid, 1095
  Huffman's algorithm and, 1089
  intersection methods of, 1069–1073
  k-dimensional structures, 1080–1081
  list, 1081–1083
  ordered, 1077
  polymorphic types, 1068
  programmatic interfaces, 1068–1077
  ray intersection, 1095–1099
  selecting, 1077
  trees, 1083–1093
Spatial frequencies, 103
SPD. *See* Spectral power distribution (SPD)
Special orthogonal matrix, 230
Specification
  camera, 301–303
  color, 133
  transformations and camera, 299–317
Spectral irradiance, units of, 698
Spectralon, 720
Spectral power distribution (SPD), 747
  incandescent lights, 748
  monospectral distributions, 747
Spectral radiance, 692
  integral of, 692
Spectral radiant exitance, 699
Spectrum (of a signal), 513
  replication in, 523–524
Specular, in graphics, 8
Specular exponent, 137
Specular power. *See* Specular exponent
Specular reflections, 137, 713
  physical models for, 726–727
Specular (mirror) reflections, 353
Specular surface, 27
Sphere mapping, 340
Sphere-to-cylinder projection theorem, 688
Sphere trees, 649, 1093
Spherical harmonics, 531, 843, 848–851
Spherical linear interpolation, 275. *See also* Slerp
Spline patches, 344
  and subdivision surfaces, 343–344

Splines, 343, 595ff, 599, 607ff, 623
Splitting plane, 1030, 1030f
s-polarized wave, 681
Spot color, 766
Spotlight, 133, 702
Square integrable, 506
Square summable, 506
sRGB standard, 774
Stabbing line, 1051
Stabbing tree, 1051–1052
Stamping, 985
Standard basis vectors, 227
Standard deviation, 807
Standard implicit form for a line, 165
Standard parallel view volume, 307
Standard perspective view volume, 307
Star, of a simplex, 208, 208f
Star, of a vertex, 208, 641
Star of an edge, 641
State machine, 454
State variable, 454
State vectors, 1015
Static frame, 462–463
Statistical estimators, 827–830
Stefan-Boltzmann law, 672
Stencil buffer, 329
Steradians, 688
Steven Anson Coons Award, 608
Stratified sampling, 920
  blue-noise property of, 921
Strobing (motion perception), 977
Strokes (expressive rendering), 950–951
  creation of, 951
  imitation of artistic technique for creating, 951
  oil-paint, 951
  pen-and-ink, 951
  physical simulation, 951
  scanning/photography approach for creating, 951
Styles, 85
  artistic, 947
Subcomponents, 138, 141–144
Subdivision, meshes, 211
Subdivision, of triangle meshes, 188
Subdivision curves, 604
Subdivision surfaces, 344, 607
  Catmull-Clark, 610–613
  modeling with, 613–614
Subdivision surface shaders, 931
Subsurface reflector, 720
Subsurface scattering, 353, 720, 738–739
  computing, 739
  modeling, 739
  physical modeling, 739
  practical effects of modeling, 739
Subtractive color, 760
Suggestive contour generator, 958
Suggestive contours, 957–958
  characteristics of, 958
Suggestive interface, 589
Summary measures, of light, 670
Sum-squared difference, 104
Superposition, 361
Supersampled antialiasing (SSAA), 1056–1057
  advantages of, 1056
  drawbacks of, 1056–1057
  implementing, 1056
Surface mesh
  embedding of, 642
Surface normal, 16, 27–28
Surface radiance, 786–787, 834, 846
  computation from field radiance, 786
Surface radiance function, 787
Surface representations, 882–887
Surface, 390, 607
  with boundary, 637–638
  closed, 638
  implicit, 619–620
  orientable, 639
  oriented, 639
  representations, 882–887
  triangulated, 637–638
Surface scattering, 712–714
  physical constraints on, 713–714
  scattering models, 713
Surface Texture, 132, 547ff
  texturing via stretching, 132
  texturing via tiling, 132
  in WPF, 130–132
Surface with boundary, 638
Surjective, 151. *See also* Bijective
Sutherland-Hodgman 2D clipping, 1045–1046
Swapping loops, 418–419

T

Tabulated functions, 201
Tagged Image File Format (TIFF), 482
Tangent field, 1015–1016
Tangent-space basis, 340
Tangent vectors
  building from a parameterization, 552–553
Target (texture image), 563
Taylor polynomial, 1017
Teddy (user interface), 590
Telecine (interlacing), 978–980
Template (pixels), 563
Templated classes in programming languages, 1068
Temporal aliasing, 980–983
  motion blur and, 980–983
Temporal coherence, 950, 962, 983
  advantage of, 983
  burden of, 985–987
  exploiting, 983–984
Tent-shaped functions, 208. *See also* Basis functions
Tent-shaped graphs. *See* Basis functions; Tent-shaped functions

Tessellation shaders, 652, 931
Test beds, 72, 81
  2D Graphics, 81–98
  application of, 95–98
  details of, 82–87
  structure of, 83–88
  using 2D, 82–83
Test scene, 402–403
Texels, 365, 742
Texture aliasing, 216
Texture coordinates, 339, 548
  assigning, 555–557
  assignment of, 215–216
Texture mapping, 131, 214–215, 547ff
  application examples of, 557t
  defined, 548
  details of, 216
  image-based, 559
  problems, 216–217
  properties of, 555
Texture maps, 15–16, 215, 547ff
  codomains for, 553–554
Texture parameterization, 555
Textures
  modeling, 630
  projection, 555
Texture-space diffusion, 341
Texture synthesis, 559–562
  Fourier-like synthesis, 559–560
  Perlin noise, 560–561, 561f
  reaction-diffusion textures, 561–562
Texturing
  bump mapping, 550–551
  contour drawing, 551–552
  environment mapping, 549–550
  variations of, 549–552
  via stretching, 132
  via tiling, 132
3ds Max transformation widget, 588–589, 588f
3D transformations
  building, 237
3D view manipulation widget, 588–589, 588f
3-space
  essential mathematics and geometry of, 149–182
3:2 pulldown algorithm (interlacing), 979
TIFF. *See* Tagged Image File Format (TIFF)
Tile fragments, 1137
Tiling rasterizer, 430–431
Tilting principle, 180–181
Time domain, 513
Time-state space, 1013–1015
TIN. *See* Triangulated Irregular Network (TIN)
Tints (color), 756
T-junction, 642
Tone mapping, 919
Tones (color), 756
Tool trays, 569
Toon-shading, 940–942
  fragment shader for improved, 942
  pixel shader for, 941
  two versions of, 940–942
  vertex shader for, 940–941
Torrance-Sparrow model, 729–731
Total internal reflection, 682
Trait data structure, 1074
Traits, as key extraction method, 1074–1077
  advantages of, 1076
  disadvantages of, 1076
Transformation
  linear, 307
  modeling hierarchy and camera, 313–315
  perspective-to-parallel, 313
  projective, 308
  rasterizing renderer ripeline and camera, 310–312
  unhinging, 307
  windowing, 300
Transformation associated to the matrix M, 224
Transformation pipeline, 460
Transformations, 221–286, 288–290
  adjoint, 253
  affine, 234, 259
  AffineTransformation2, 288
  associated, 294
  change-of-coordinate, 231
  classes of, 288–289
  composed, 235
  and coordinate systems, 229–230
  covector, 253
  efficiency of, 289–290
  finding the matrix for, 226–228
  fractional linear, 256
  homogenizing, 265
  implementation, 290–293
  vs. interpolation, 259
  linear, 221, 259
  LinearTransformation2, 288
  matrix, 222
  MatrixTransformation2, 288
  modeling, 630
  parametric lines, 254
  projective, 255, 257, 259–260, 263, 291–293
  ProjectiveTransformation2, 289
  restricted, 295–297
  scale, 263–264
  shearing, 264
  specification of, 290
  in three dimensions, 263–285
  in two dimensions, 221–262
  of vectors, 250–253
  windowing, 236–237, 236f
  world-centered rotation, 272
Translation, 222, 233–234, 263
Translation equivariant reconstruction, of signal, 546

Translucency, and blending, 361–364
Transmission/rendering, 353, 367–368
  reduction, 470–472
Transmissive scattering, 715
Transparent surface, 364
Transport, light, 335–336
Transport equation, 786–787
Transport paths, separation of, 844
Transposition, 156
Trees, as a class of spatial data structures, 1083–1093
  binary space partition (BSP) trees, 1084–1089
  Bounding Volume Hierarchy (BVH), 1092–1093, 1092f
  building BSP trees, 1089–1092
  *k*d tree, 1089
  oct tree, 1090
  quad trees, 1090, 1091f
Triangle fan, 338
Triangle list, 338
Triangle meshes, 187, 187f, 188f, 635
  icosahedral, 187
  1D mesh, 189
  shape approximation, 187–188
  simplification of, 188, 649
  subdivision of, 188
  uniformity of, 188
Triangle processing, 17
Triangle reordering for hardware efficiency, 664–667
Triangles, 171–175
  half-planes and, 174–175
  parameterization of, 171
  signed area, 177
  in space, 173–174
Triangle soup, 338
Triangle strip, 338
Triangulated Irregular Network (TIN), 345
Triangulated surfaces, 637–638
Triple buffering, 976
Trotter, Hale, 149
2D barycentric weights, 424–427
2D coverage sampling, 422
2D graphics
  dynamics in 2D graphics using WPF, 55–58
  evolution of, 37–41
  overview of, 36–37
  test beds, 81–98
  and Windows Presentation Foundation (WPF), 35–60
2D manipulation
  multitouch interaction for, 574–580
2D raster graphics platform, 38
2D scene
  WPF to specify, 41–55
2D scissoring, 1044
2D transformations, 238–239
  building, 238–239
2-ring (vertices), 641
2-space
  essential mathematics and geometry of, 149–182
Two-and-a-half dimensional, 43
Two-tone shading, 959

U
Übershader, 441
UI controls, 39, 41
UI generator, 37
Ulam, Stanislaw, 945
Umbilic points, 956
Umbra, 505, 798
Uncanny valley, 19
Undragging, 581
Unhinging transformation, 307
Unicam, 584–587
Uniform color space, 767
Uniform density
  on the sphere, 813
  defined, 809
Uniform spline, 601
Uniformity, of triangle meshes, 188
Uniform random variable, 807
Uniform scaling transformation, 223
Units, 333
Unit vector, 157, 229
Unoccluded two-point transport intensity, 847
Unoriented meshes, 191
Unpolarized light, 683
Unsigned normalized, 325
Up direction, 122, 302
User interface (UI), 6–7, 21
User interface examples, 588–591
  Chateau, 589–590
  first-person-shooter controls, 588
  Grabcut, 590–591
  Photoshop's free-transform mode, 589
  Teddy, 590
  3ds Max transformation widget, 588–589, 588f
Utility classes, 395–400
uv-coordinates, 216

V
Valence, 637
Valleys, 956–957
Value (data structure), 1077
Value of measurement, 323–324
Vanishing point, 77
Variables
  change of, 690–692
Variance, 807, 818
Variance reduction, 921
Vectorization (programmer instruction batching), 1033
Vectors, 155–161, 234–235, 288
  coordinate, 155
  edge, 175
  indexing, 156

Vectors (*continued*)
kinds of, 162–163
length of, 157–161
normal, 164
normalizing, 157
operations, 157–161
transforming, 250–253
Vertex/Vertices, 50, 65, 189
boundary, 194, 638
degree of, 637
interior, 194, 638
link of, 208
locally flat, 643
manifold, 194f
star of, 208
Vertex geometry processing, 17
Vertex geometry transformation, 17
Vertex normal, 129
Vertex shaders, 465, 930, 931
Vertex stage
and pixel stages, 433
Vertical synchronization, 976
Video standards, 775–776
View center (camera manipulation), 586
ViewCube (3D view manipulation widget), 588–589, 588f
View-frustum culling, 470, 1023
Viewing stage, 460
Viewport, 37, 302, 455
Viewport3D, 119, 121
View region, 63
View specification
building transformations from, 303–310
View volume, 77, 120, 302
standard parallel, 307
standard perspective, 305, 307
Vignetting, 336
Virtual arcball, 280–283, 584
Virtual parallelism, 1113ff
Virtual sphere, 580
Virtual trackball, 280–283, 580
Virtual transitions, 671
Visibility, 65
conservative, 1023
coverage (binary), 1027–1028
goals for, 1023
list-priority algorithms, 1040–1043
primary, 1023, 1024, 1027
sector-based conservative, 1050–1054
Visibility determination
applications of depth buffer in, 1035–1036
backface culling, 1047–1049
current practice and motivation, 1028–1029
depth buffer, 1034–1040
frustum clipping, 1028, 1045–1046
frustum culling, 1023, 1028, 1044
hardware rasterization renderers and, 1028
hierarchical occlusion culling, 1049–1050
list-priority algorithms, 1040–1043
partial coverage, 1028, 1054–1062
ray casting, 1029–1034
ray-tracing renderers and, 1028
Visibility function, 786, 799, 1025–1027
evaluating, 1026
Visibility problem. *See* Visibility testing
Visibility testing, 422
Visible contour, 953. *See also* Contour(s)
Visible points, 390–391
Visible spectrum, 330–332
Visible surface determination, 1023. *See also* Hidden surface removal
Vision
photopic, 753–754
scotopic, 753–754
Visual cortex, 103, 106, 108, 110
Visual perception, human. *See* Human visual perception
Visual system, 103–105
applications of, 105, 109–110
components of, 103
Volumetric models, 349–351
Volumetric scattering, 737
Voxelization, conservative, 1096
Voxels, 349–350
VRML, 479
vup, 302

W
Walk cycle, 966
Warnock's Algorithm, 1041
Warped *z*-buffer, 1038
Watertight model (meshes), 643
Wavelength, 332, 675
Wave theory models, 734–736
Wave velocity, 675
w-buffer, 1040. *See also* Depth buffer
WebGL, 479
Weiler-Atherton Algorithm, 1041
Wheel of reincarnation, 18–19
Whites (color), 769
CIE definitions, 769
illuminant *C,* 769
illuminant *E,* 769
Whole-frustum clipping, 1044, 1047
Widgets. *See* UI controls
Wien's displacement law, 710
Wii (interface device), 568
WIMP (windows, icons, menus, pointers) GUI (WIMP GUI), 6, 8, 567
Winding number, 176–177
Window chrome, 36
Windowing transformation, 236–237, 236f, 300
Windows Presentation Foundation (WPF)
2D Graphics using, 35–60
application/developer interface layers, 40–41
canvas coordinate system, 45–46
data dependencies, 91–92
dynamics in 2D graphics using, 55–58

reflectance model, 133–138
to specify 2D scene, 41–55
surface texture in, 130–132
Winged-edge data structure, 196
Wire-frame model, 65
The Wizard of Oz (film), 950
Woodcut, Dürer, 61–65, 1035
World-centered rotation, 272
World coordinate system, 119
World space, 21, 245
WPF. *See* Windows Presentation Foundation (WPF)
WPF 3D, 117
design of, 118
high-level overview of, 119–120

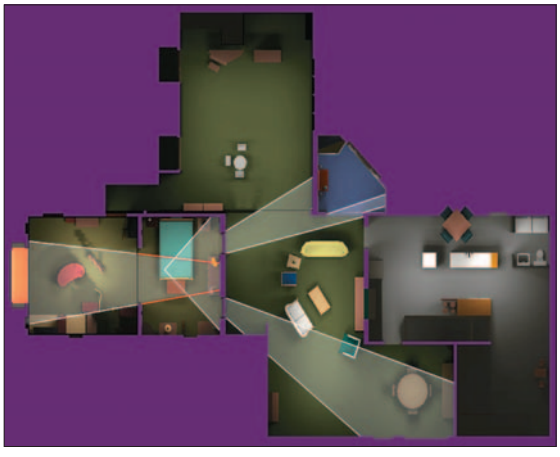
X
X3D (language), 479
XAML. *See* Extensible Application Markup Language (XAML)
Xerox PARC, 8
XToon shading, 942–943
2D texture map for, 942f
atmospheric perspective, 942
vertex and fragment shaders for, 942–943

Y
Yaw, 267
YIQ color model, 775

Z
z-buffer, 306, 310, 392. *See also* Depth buffer
z-buffer value, 1038, 1039f
z-data, 482
Zero set, 164, 616. *See also* Isosurfaces
Zero-to-one coordinates. *See* Normalized device coordinates
z-fighting, 1037
z-values
perspective and, 313

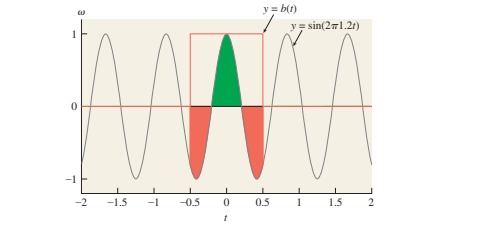
This page intentionally left blank

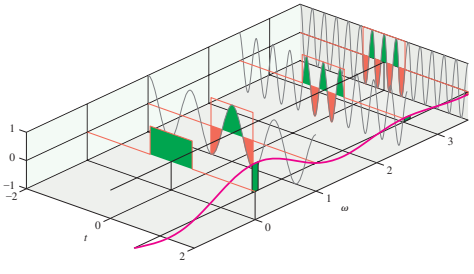

Courtesy of Stephen Marschner, © 2001 ACM, Inc. Reprinted by permission.

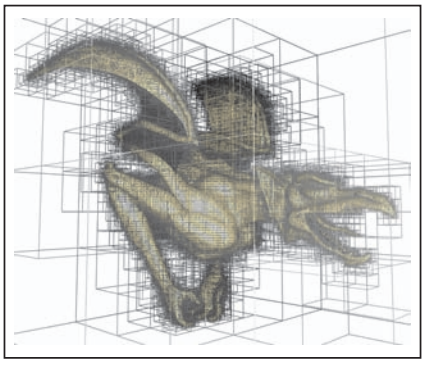



Courtesy of David Luebke © 1995 ACM, Inc. Reprinted by permission.

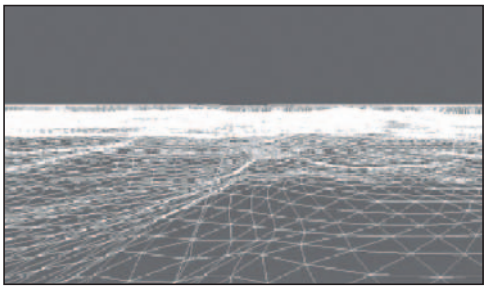
Courtesy of Pradeep Sen and Soheil Darabi, © 2012 ACM, Inc. Reprinted by permission.



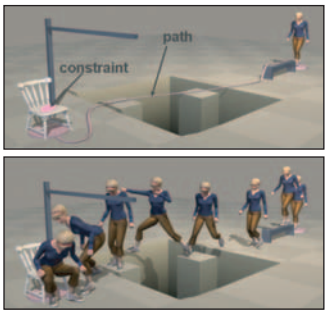




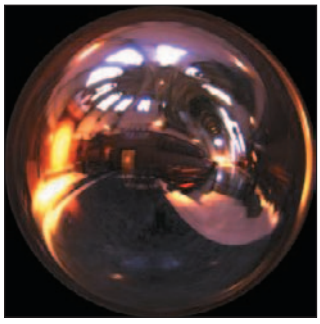
Courtesy of Preshu Ajmera.

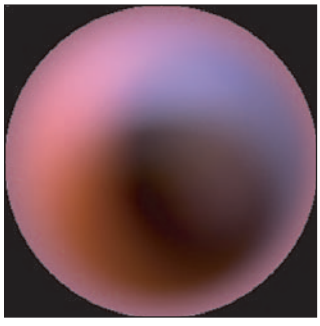



Courtesy of Tiago Sousa, © Crytek.

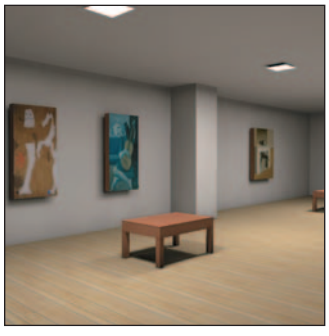


Courtesy of Jessica Hodgins and Alla Safonova © 2007 ACM, Inc. Reprinted by permission.

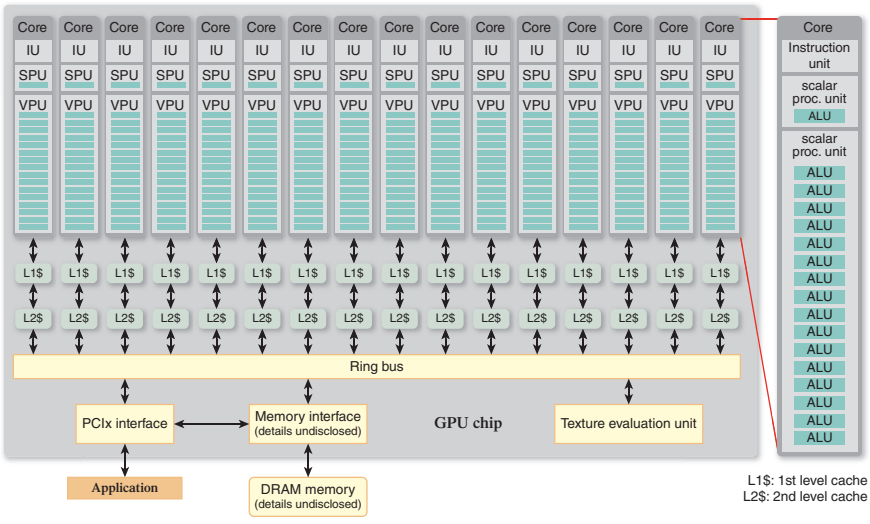


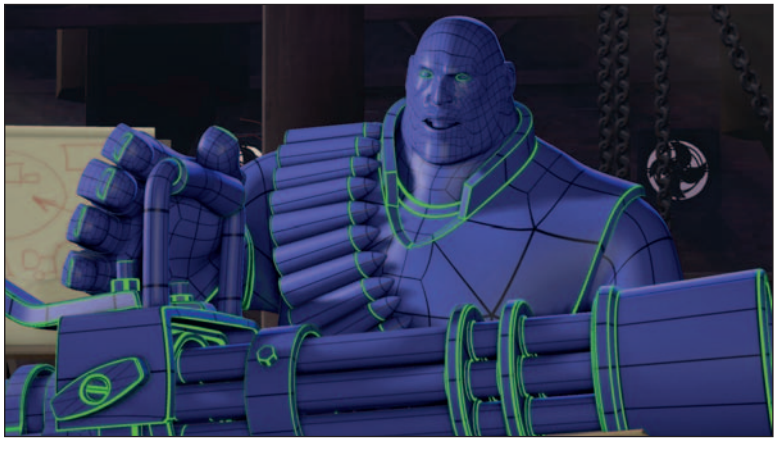


Top: Copyright 2012 University of Southern California, Institute for Creative Technologies. Bottom: Courtesy of Ravi Ramamoorthi and Pat Hanrahan, © 2001 ACM, Inc. Reprinted by permission.



Courtesy of Greg Coombe, "Radiosity on graphics hardware" by Coombe, Harris and Lastra, Proceedings of Graphics Interface 2004.





Top: © Valve, all rights reserved. Bottom: Courtesy of Denis Kovacs; © 2010 ACM, Inc. Reprinted by permission.