# Fundamentals of Computer Graphics

Peter Shirley
School of Computing
University of Utah

*with*

Michael Ashikhmin
Michael Gleicher
Stephen R. Marschner
Erik Reinhard
Kelvin Sung
William B. Thompson
Peter Willemsen

A K Peters
Wellesley, Massachusetts

# Contents

Preface xi

1 Introduction 1

1.1 Graphics Areas . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2 Major Applications . . . . . . . . . . . . . . . . . . . . . . 2
1.3 Graphics APIs . . . . . . . . . . . . . . . . . . . . . . . . 3
1.4 3D Geometric Models . . . . . . . . . . . . . . . . . . . . 4
1.5 Graphics Pipeline . . . . . . . . . . . . . . . . . . . . . . . 4
1.6 Numerical Issues . . . . . . . . . . . . . . . . . . . . . . . 5
1.7 Efficiency . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
1.8 Software Engineering . . . . . . . . . . . . . . . . . . . . . 8

2 Miscellaneous Math 15

2.1 Sets and Mappings . . . . . . . . . . . . . . . . . . . . . . 15
2.2 Solving Quadratic Equations . . . . . . . . . . . . . . . . . 19
2.3 Trigonometry . . . . . . . . . . . . . . . . . . . . . . . . . 20
2.4 Vectors . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.5 2D Implicit Curves . . . . . . . . . . . . . . . . . . . . . . 30
2.6 2D Parametric Curves . . . . . . . . . . . . . . . . . . . . 36
2.7 3D Implicit Surfaces . . . . . . . . . . . . . . . . . . . . . 38
2.8 3D Parametric Curves . . . . . . . . . . . . . . . . . . . . 40

2.9 3D Parametric Surfaces . . . 41
2.10 Linear Interpolation . . . 42
2.11 Triangles . . . 43

3 Raster Algorithms 51
3.1 Raster Displays . . . 51
3.2 Monitor Intensities and Gamma . . . 52
3.3 RGB Color . . . 54
3.4 The Alpha Channel . . . 56
3.5 Line Drawing . . . 57
3.6 Triangle Rasterization . . . 63
3.7 Simple Antialiasing . . . 67
3.8 Image Capture and Storage . . . 68

4 Signal Processing 71
4.1 Digital Audio: Sampling in 1D . . . 72
4.2 Convolution . . . 75
4.3 Convolution Filters . . . 89
4.4 Signal Processing for Images . . . 96
4.5 Sampling Theory . . . 104

5 Linear Algebra 119
5.1 Determinants . . . 119
5.2 Matrices . . . 121

6 Transformation Matrices 135
6.1 Basic 2D Transforms . . . 135
6.2 Basic 3D Transforms . . . 147
6.3 Translation . . . 151
6.4 Inverses of Transformation Matrices . . . 154
6.5 Coordinate Transformations . . . 154

7 Viewing 159
7.1 Drawing the Canonical View Volume . . . 160
7.2 Orthographic Projection . . . 162
7.3 Perspective Projection . . . 166
7.4 Some Properties of the Perspective Transform . . . 172
7.5 Field-of-View . . . 173

8 Hidden Surface Elimination 177
8.1 BSP Tree . . . . . 177
8.2 Z-Buffer . . . . . 186

9 Surface Shading 191
9.1 Diffuse Shading . . . . . 191
9.2 Phong Shading . . . . . 194
9.3 Artistic Shading . . . . . 197

10 Ray Tracing 201
10.1 The Basic Ray-Tracing Algorithm . . . . . 202
10.2 Computing Viewing Rays . . . . . 203
10.3 Ray-Object Intersection . . . . . 205
10.4 A Ray-Tracing Program . . . . . 209
10.5 Shadows . . . . . 211
10.6 Specular Reflection . . . . . 212
10.7 Refraction . . . . . 213
10.8 Instancing . . . . . 216
10.9 Sub-Linear Ray-Object Intersection . . . . . 218
10.10 Constructive Solid Geometry . . . . . 229
10.11 Distribution Ray Tracing . . . . . 229

11 Texture Mapping 239
11.1 3D Texture Mapping . . . . . 240
11.2 2D Texture Mapping . . . . . 246
11.3 Tessellated Models . . . . . 248
11.4 Texture Mapping for Rasterized Triangles . . . . . 250
11.5 Bump Textures . . . . . 252
11.6 Displacement Mapping . . . . . 253
11.7 Environment Maps . . . . . 253
11.8 Shadow Maps . . . . . 255

12 A Full Graphics Pipeline 259
12.1 Clipping . . . . . 259
12.2 Location of Clipping Segment of the Pipeline . . . . . 260
12.3 An Expanded Graphics Pipeline . . . . . 264
12.4 Backface Elimination . . . . . 265
12.5 Triangle Strips and Fans . . . . . 266
12.6 Preserved State . . . . . 266
12.7 A Full Graphics Pipeline . . . . . 267

13 Data Structures for Graphics 269
13.1 Triangle Meshes . . . 269
13.2 Winged-Edge Data Structure . . . 270
13.3 Scene Graphs . . . 272
13.4 Tiling Multidimensional Arrays . . . 274

14 Sampling 279
14.1 Integration . . . 279
14.2 Continuous Probability . . . 284
14.3 Monte Carlo Integration . . . 288
14.4 Choosing Random Points . . . 291

15 Curves 301
15.1 Curves . . . 301
15.2 Curve Properties . . . 307
15.3 Polynomial Pieces . . . 310
15.4 Putting Pieces Together . . . 318
15.5 Cubics . . . 321
15.6 Approximating Curves . . . 327
15.7 Summary . . . 344

16 Computer Animation 347
16.1 Principles of Animation . . . 348
16.2 Keyframing . . . 352
16.3 Deformations . . . 360
16.4 Character Animation . . . 361
16.5 Physics-Based Animation . . . 367
16.6 Procedural Techniques . . . 370
16.7 Groups of Objects . . . 373
16.8 Notes . . . 376

17 Using Graphics Hardware 379
17.1 What is Graphics Hardware . . . 379
17.2 Describing Geometry for the Hardware . . . 380
17.3 Processing Geometry into Pixels . . . 387

18 Building Interactive Graphics Applications 401
18.1 The Ball Shooting Program . . . 402
18.2 Programming Models . . . 404
18.3 The Modelview-Controller Architecture . . . 421

18.4 Example Implementations . . . 433
18.5 Applying Our Results . . . 443
18.6 Notes . . . 446
18.7 Exercises . . . 447

19 Light 451
19.1 Radiometry . . . 451
19.2 Transport Equation . . . 460
19.3 Photometry . . . 462

20 Color 465
20.1 Light and Light Detectors . . . 466
20.2 Tristimulus Color Theory . . . 466
20.3 CIE Tristimulus Values . . . 468
20.4 Chromaticity . . . 469
20.5 Scotopic Luminance . . . 472
20.6 RGB Monitors . . . 472
20.7 Approximate Color Manipulation . . . 473
20.8 Opponent Color Spaces . . . 474

21 Visual Perception 477
21.1 Vision Science . . . 478
21.2 Visual Sensitivity . . . 479
21.3 Spatial Vision . . . 495
21.4 Objects, Locations, and Events . . . 509
21.5 Picture Perception . . . 517

22 Tone Reproduction 521
22.1 Classification . . . 524
22.2 Dynamic Range . . . 525
22.3 Color . . . 527
22.4 Image Formation . . . 529
22.5 Frequency-Based Operators . . . 529
22.6 Gradient-Domain Operators . . . 531
22.7 Spatial Operators . . . 532
22.8 Division . . . 534
22.9 Sigmoids . . . 535
22.10 Other Approaches . . . 540
22.11 Night Tonemapping . . . 543
22.12 Discussion . . . 544

23 Global Illumination 547
23.1 Particle Tracing for Lambertian Scenes . . . . . . . . . . . . . 548
23.2 Path Tracing . . . . . . . . . . . . . . . . . . . . . . . . . . . 551
23.3 Accurate Direct Lighting . . . . . . . . . . . . . . . . . . . . . 553

24 Reflection Models 561
24.1 Real-World Materials . . . . . . . . . . . . . . . . . . . . . . . 561
24.2 Implementing Reflection Models . . . . . . . . . . . . . . . . . 563
24.3 Specular Reflection Models . . . . . . . . . . . . . . . . . . . 565
24.4 Smooth Layered Model . . . . . . . . . . . . . . . . . . . . . . 566
24.5 Rough Layered Model . . . . . . . . . . . . . . . . . . . . . . 569

25 Image-Based Rendering 577
25.1 The Light Field . . . . . . . . . . . . . . . . . . . . . . . . . . 578
25.2 Creating a Novel Image from a Set of Images . . . . . . . . . . 579

26 Visualization 583
26.1 2D Scalar Fields . . . . . . . . . . . . . . . . . . . . . . . . . 583
26.2 3D Scalar Fields . . . . . . . . . . . . . . . . . . . . . . . . . 585

References 595

Index 613

# Preface

This book is a product of several graphics courses I have taught at Indiana University and the University of Utah. All graphics books must choose between teaching the low-level details "under the hood" of graphics programs or teaching how to use modern graphics APIs, such as OpenGL, Direct3D, and Java3D. This book chooses the former approach. I do not have a good justification for this choice other than that I have taught both styles of courses, and the students in the "low-level" courses seemed to understand the material better than the other students and even seemed to use the APIs more effectively. There are many reasons this might be true, and the effect may not transfer to other teachers or schools. However, I believe that teaching the fundamentals is usually the right approach, whether in graphics, another academic discipline, or a sport.

## How to Use this Book

The book begins with nine chapters that roughly correspond to a one-semester course which takes students through the graphics pipeline and basic ray tracing. It has students implement everything—i.e., it is not a "learn OpenGL" style text. However, the pipeline presented is consistent with the one implemented in graphics hardware, and students who have used the book should find OpenGL or other common APIs familiar in many ways.

The second part of the book is a series of advanced topics that are not highly ordered. This allows a variety of second-semester courses and a few weeks of advanced topics in a first semester course.

For the first semester, I would suggest the following as a possible outline of initial assignments:

1. Math homework at the end of Chapter 2 followed by at least one in-class exam.

2. Line rasterization.

3. Triangle rasterization with barycentric color interpolation.

4. Orthographic wireframe drawing.

5. Perspective wireframe drawing.

6. BSP-tree with flat-shaded triangles and wireframe edges with only trivial z-clipping and with mouse-driven viewpoint selection.

7. Finite-precision z-buffer implementation with only trivial z-clipping.

Following these assignments, the instructor could do assignments on ray tracing or could have the students add shadow-maps, Phong lighting, clipping, and textures to their z-buffers, or they could move the students into programming with a 3D API.

## About the Cover

The cover image is from *Tiger in the Water* by J. W. Baker (brushed and airbrushed acrylic on canvas, 16" by 20", www.jwbart.com).

The subject of a tiger is a reference to a wonderful talk given by Alain Fournier (1943–2000) at the Cornell Workshop in 1998. His talk was an evocative verbal description of the movements of a tiger. He summarized his point:

> Even though modelling and rendering in computer graphics have been improved tremendously in the past 35 years, we are still not at the point where we can model automatically a tiger swimming in the river in all its glorious details. By automatically I mean in a way that does not need careful manual tweaking by an artist/expert.
>
> The bad news is that we have still a long way to go.
>
> The good news is that we have still a long way to go.

## Online Resources

The web site for this book is http://www.cs.utah.edu/~shirley/fcg2/. I will maintain an errata list there as well as links to people using the book in classes. Although I do not provide slides for the course, Rich Riesenfeld has graciously agreed to make his excellent slides available, and a pointer to those slides will be available at the book's web site. Most of the figures in this book are in Abobe Illustrator format. I would be happy to convert specific figures into portable formats on request. Please feel free to contact me at shirley@cs.utah.edu.

## Changes in this Edition

There are many small changes in the material from the first edition of this book, but the two large changes are the addition of a bibliography and the addition of new chapters written by colleagues. These colleagues are people I think are clear thinkers and communicators, and I invited them each to write a chapter with arm-twisting designed to get certain key topics covered. Most of them have used the book in a class and were thus familiar with its approach. The bibliography is not meant to be extensive, but is instead a place for readers to get started. I am sure there are omissions there, and I would like to hear about any crucial references we have missed. The new chapters are:

**Signal Processing** by Stephen Marschner, Cornell University (Chapter 4).

**Curves** by Michael Gleicher, University of Wisconsin (Chapter 15).

**Computer Animation** by Michael Ashikhmin, SUNY at Stony Brook (Chapter 16).

**Using Graphics Hardware** by Peter Willemsen, University of Minnesota Duluth (Chapter 17).

**Building Interactive Graphics Applications** by Kelvin Sing, University of Washington Bothell (Chapter 18)

**Visual Perception** by William B. Thompson, University of Utah (Chapter 21).

**Tone Reproduction** by Erik Reinhard, University of Central Florida (Chapter 22).

## Acknowledgements

The following people have provided helpful comments about the book: Josh Andersen, Zeferino Andrade, Michael Ashikhman, Adam Berger, Adeel Bhutta, Solomon Boulos, Stephen Chenney, Michael Coblenz, Greg Coombe, Frederic Cremer, Brian Curtin, Dave Edwards, Jonathon Evans, Karen Feinauer, Amy Gooch, Eungyoung Han, Chuck Hansen, Andy Hanson, Dave Hart, John Hart, Helen Hu, Vicki Interrante, Henrik Wann Jensen, Shi Jin, Mark Johnson, Ray Jones, Kristin Kerr, Dylan Lacewell, Mathias Lang, Philippe Laval, Marc Levoy, Howard Lo, Ron Metoyer, Keith Morley, Eric Mortensen, Tamara Munzner, Koji Nakamaru, Micah Neilson, Blake Nelson, Michael Nikelsky, James O'Brien, Steve Parker, Sumanta Pattanaik, Matt Pharr, Peter Poulos, Shaun Ramsey, Rich Riesenfeld, Nate Robins, Nan Schaller, Chris Schryvers, Tom Sederberg, Richard Sharp, Sarah Shirley, Peter-Pike Sloan, Tony Tahbaz, Jan-Phillip Tiesel, Bruce Walter, Alex Williams, Amy Williams, Chris Wyman, and Kate Zebrose.

Ching-Kuang Shene and David Solomon allowed me to borrow examples from their works. Henrik Jensen, Eric Levin, Matt Pharr, and Jason Waltman generously provided images. Brandon Mansfield was very helpful in improving the content of the discussion of hierarchical bounding volumes for ray tracing. Carrie Ashust, Jean Buckley, Molly Lind, Pat Moulis, and Bob Shirley, provided valuable logistical support. Miranda Shirley provided valuable distractions.

I am extremely thankful to J. W. Baker helping me to get the cover I envisioned. In addition to being a talented artist, he was a great pleasure to work with personally.

Many works were helpful in preparing this book, and most of them appear in the notes for the appropriate chapters. However, a few pieces that influenced the content and presentation do not, and I list them here. I thank the authors for their help. These include the two classic computer graphics texts I first learned the basics from as a student: *Computer Graphics: Principles & Practice* (Foley, Van Dam, Feiner, & Hughes, 1990), and *Computer Graphics* (Hearn & Baker, 1986). Other texts include both of Alan Watt's classic books (Watt, 1993, 1991), Hill's *Computer Graphics Using OpenGL* (Francis S. Hill, 2000), Angel's *Interactive Computer Graphics: A Top-Down Approach With OpenGL* (Angel, 2002), Hughes Hoppe's University of Washington dissertation (Hoppe, 1994), and Rogers' two classic graphics texts (D. F. Rogers, 1985, 1989).

This book was written using the LaTeX document preparation software on an Apple Powerbook. The figures were made by the author using the *Adobe Illustrator* package. I would like to thank the creators of those wonderful programs.

I'd like to thank the University of Utah for allowing me to work on this book during sabbatical.

I would like to especially thank Alice and Klaus Peters for encouraging me to write the first edition of this book, for their great skill in bringing a book to fruition and for their dedication to making their books the best they can be. In addition to finding many errors in formulas and language in the second edition, they put in many weeks of extremely long hours in the home stretch of the process, and I have no doubt this book would not have been finished without their extraordinary efforts.

Salt Lake City
April 2005

Peter Shirley

# 1

# Introduction

The term *Computer Graphics* describes any use of computers to create or manipulate images. This book takes a slightly more specific view and deals mainly with algorithms for image generation. Doing computer graphics inevitably requires some knowledge of specific hardware, file formats, and usually an API[1] or two. The specifics of that knowledge are a moving target due to the rapid evolution of the field, and therefore such details will be avoided in this text. Readers are encouraged to supplement the text with relevant documentation for their software/hardware environment. Fortunately the culture of computer graphics has enough standard terminology and concepts that the discussion in this book should map nicely to most environments. This chapter defines some basic terminology, and provides some historical background as well as information sources related to computer graphics.

## 1.1 Graphics Areas

It is always dangerous to try to categorize endeavors in any field, but most graphics practitioners would agree on the following major areas and that they are part of the field of computer graphics:

[1] An *application program interface* (API) is a software interface for basic operations such as line drawing. Current popular APIs include OpenGL, Direct3D, and Java3D.

- **Modeling** deals with the mathematical specification of shape and appearance properties in a way that can be stored on the computer. For example, a coffee mug might be described as a set of ordered 3D points along with some interpolation rule to connect the points and a reflection model that describes how light interacts with the mug.
- **Rendering** is a term inherited from art and deals with the creation of shaded images from 3D computer models.
- **Animation** is a technique to create an illusion of motion through sequences of images. Here, modeling and rendering are used, with the handling of time as a key issue not usually dealt with in basic modeling and rendering.

There are many other areas that involve computer graphics, and whether they are core graphics areas is a matter of opinion. These will all be at least touched on in the text. Such related areas include the following:

- **User interaction** deals with the interface between input devices such as mice and tablets, the application, and feedback to the user in imagery and other sensory feedback. Historically, this area is associated with graphics largely because graphics researchers had some of the earliest access to the input/output devices that are now ubiquitous.
- **Virtual reality** attempts to *immerse* the user into a 3D virtual world. This typically requires at least stereo graphics and response to head motion. For true virtual reality, sound and force feedback should be provided as well. Because this area requires advanced 3D graphics and advanced display technology, it is often closely associated with graphics.
- **Visualization** attempts to give users insight via visual display. Often there are graphic issues to be addressed in a visualization problem.
- **Image processing** deals with the manipulation of 2D images and is used in both the fields of graphics and vision.
- **3D scanning** uses range-finding technology to create measured 3D models. Such models are useful for creating rich visual imagery, and the processing of such models often requires graphics algorithms.

## 1.2 Major Applications

Almost any endeavor can make some use of computer graphics, but the major consumers of computer graphics technology include the following industries:

- **Video games** increasingly use sophisticated 3D models and rendering algorithms.

- **Cartoons** are often rendered directly from 3D models. Many traditional 2D cartoons use backgrounds rendered from 3D models which allows a continuously moving viewpoint without huge amounts of artist time.

- **Film special effects** use almost all types of computer graphics technology. Almost every modern film uses digital compositing to superimpose backgrounds with separately filmed foregrounds. Many films use computer-generated foregrounds with 3D models.

- **CAD/CAM** stands for *computer-aided design* and *computer-aided manufacturing*. These fields use computer technology to design parts and products on the computer and then, using these virtual designs, to guide the manufacturing procedure. For example, many mechanical parts are now designed in a 3D computer modeling package, and are then automatically produced on a computer-controlled milling device.

- **Simulation** can be thought of as accurate video gaming. For example, a flight simulator uses sophisticated 3D graphics to simulate the experience of flying an airplane. Such simulations can be extremely useful for initial training in safety-critical domains such as driving, and for scenario training for experienced users such as specific fire-fighting situations that are too costly or dangerous to create physically.

- **Medical imaging** creates meaningful images of scanned patient data. For example, a magnetic resonance imaging (MRI) dataset is composed of a 3D rectangular array of density values. Computer graphics is used to create shaded images that help doctors digest the most salient information from such data.

- **Information visualization** creates images of data that do not necessarily have a "natural" visual depiction. For example, the temporal trend of the price of ten different stocks does not have an obvious visual depiction, but clever graphing techniques can help humans find patterns in such data.

## 1.3 Graphics APIs

A key part of using graphics libraries is dealing with an *application program interface* (API). An API is a software interface that provides a model for how an application program can access system functionality, such as drawing an image into a window. Typically, the two key issues in designing graphics programs are dealing with graphics calls such as "draw triangle" and handling user interaction such as a button press.

Most APIs have a user-interface toolkit of some kind that uses *callbacks*. Callbacks refer to the process of using function pointers or virtual functions to pass a reference to a function. For example, to associate an action with a button press, an underlying function is dynamically associated with the button press. In this way, the user-interface toolkit can process the event of the button press, and any action can be associated with it by the programmer.

There are currently two dominant paradigms for APIs. The first is the integrated approach of Java where the graphics and user-interface toolkits are integrated and portable *packages* that are fully standardized and supported as part of the language. The second is represented by Direct3D and OpenGL, where the drawing commands are part of a software library tied to a language such as C++, and the user-interface software is an independent entity that might vary from system to system. In this latter approach, it is problematic to write portable code, although for simple programs it may be possible to use a portable library layer on top of the system specific event-handling.

Whatever your choice of API, the basic graphics calls will be largely the same, and the concepts of this book will apply.

## 1.4 3D Geometric Models

A key part of graphics programs is using 3D geometric models. These models describe 3D objects using mathematical primitives such as spheres, cubes, cones, and polygons. The most ubiquitous type of model is composed of 3D triangles with shared vertices, which is often called a *triangle mesh*. These meshes are sometimes generated by artists using an interactive modeling program and sometimes by range scanning devices. In either case, these models usually contain many triangles, most of them small, so your programs should be optimized for such datasets.

## 1.5 Graphics Pipeline

Almost all modern computers now have an efficient 3D *graphics pipeline*. This is a special software/hardware subsystem that efficiently draws 3D primitives in perspective. Usually these systems are optimized for processing 3D triangles with shared vertices. The basic operations in the pipeline map the 3D vertex locations to 2D screen positions, and shade the triangles so that they both look realistic and appear in proper back-to-front order.

Although drawing the triangles in valid back-to-front order was once the most important research issue in computer graphics, it is now almost always solved using the *z-buffer*, which uses a special memory-buffer to solve the problem in a brute-force manner.

It turns out that the geometric manipulation used in the graphics pipeline can be accomplished almost entirely in a 4D coordinate space composed of three traditional geometric coordinates and a fourth *homogeneous* coordinate that helps us handle perspective viewing. These 4D coordinates are manipulated using 4 by 4 matrices and 4-vectors. The graphics pipeline, therefore, contains much machinery for efficiently processing and composing such matrices and vectors. This 4D coordinate system is one of the most subtle and beautiful constructs used in computer science, and it is certainly the biggest intellectual hurdle to jump when learning computer graphics. A big chunk of the first part of every graphics book deals with these coordinates.

The speed of most modern graphics pipelines is roughly proportional to the number of triangles being drawn. Because interactivity is typically more important to applications than visual quality, it is worthwhile to minimize the number of triangles used to represent a model. In addition, if the model is viewed in the distance, fewer triangles are needed than when the model is viewed from a closer distance. This suggests that it is useful to represent a model with a varying *level-of-detail* (LOD).

## 1.6 Numerical Issues

Many graphics programs are really just 3D numerical codes. Numerical issues are often crucial in such programs. In the "old days," it was very difficult to handle such issues in a robust and portable manner because machines had different internal representations for numbers, and even worse, handled exceptions in many incompatible fashions. Fortunately, almost all modern computers conform to the *IEEE floating point* standard (IEEE Standards Association, 1985). This allows the programmer to make many convenient assumptions about how certain numeric conditions will be handled.

Although IEEE floating point has many features that are valuable when coding numeric algorithms, there are only a few that are crucial to know for most situations encountered in graphics. First, and most important, is to understand that there are three "special" values for real numbers in IEEE floating point:

**infinity** ($\infty$) This is a valid number that is larger than all other valid numbers.

**minus infinity** ($-\infty$) This is a valid number that is smaller than all other valid numbers.

**not a number (NaN)** This is an invalid number that arises from an operation with undefined consequences, such as zero divided by zero.

The designers of IEEE floating point made some decisions that are extremely convenient for programmers. Many of these relate to the three special values above in handling exceptions such as division by zero. In these cases an exception is logged, but in many cases the programmer can ignore that. Specifically, for any positive real number $a$, the following rules involving division by infinite values hold:

$$\begin{aligned} +a/(+\infty) &= +0 \\ -a/(+\infty) &= -0 \\ +a/(-\infty) &= -0 \\ -a/(-\infty) &= +0 \end{aligned}$$

Note that IEEE floating point distinguishes between $-0$ and $+0$. In most graphics programs this distinction does not matter, but it is worth keeping in mind for more classical numeric algorithms.

Other operations involving infinite values behave the way one would expect. Again for positive $a$, the behavior is:

$$\begin{aligned} \infty + \infty &= +\infty \\ \infty - \infty &= \text{NaN} \\ \infty \times \infty &= \infty \\ \infty/\infty &= \text{NaN} \\ \infty/a &= \infty \\ \infty/0 &= \infty \\ 0/0 &= \text{NaN} \end{aligned}$$

The rules in a Boolean expression involving infinite values are as expected:

1. All finite valid numbers are less than $+\infty$.

2. All finite valid numbers are greater than $-\infty$.

3. $-\infty$ is less than $+\infty$.

The rules involving expressions that have NaN values are simple:

1. Any arithmetic expression that includes NaN results in NaN.

2. Any Boolean expression involving NaN is false.

Perhaps the most useful aspect of IEEE floating point is how divide-by-zero is handled; for any positive real number $a$, the following rules involving division by zero values hold:

$$
\begin{aligned}
+a/+0 &= +\infty \\
-a/-0 &= -\infty
\end{aligned}
$$

Note that some care must be taken if negative zero ($-0$) might arise in a code, but there are many numeric codes that become much simpler if the programmer takes advantage of IEEE floating point. For example, consider the expression:

$$a = \frac{1}{\frac{1}{b}+\frac{1}{c}}.$$

Such expressions arise with resistors and lenses. If divide-by-zero resulted in a program crash (as was true in many systems before IEEE floating point), then two *if* statements would be required to check for small or zero values of $b$ or $c$. Instead, with IEEE floating point, if $b$ or $c$ are zero, we will get a zero value for $a$ as desired. Another common technique to avoid special checks is to take advantage of the Boolean properties of NaN. Consider the following code segment:

```
$a = f(x)$
if ($a > 0$) then
   do something
```

Here, the function $f$ may return "ugly" values such as $\infty$ or NaN. Because the *if* statement is false for $a =$ NaN or $a = -\infty$ and true for $a = +\infty$, no special checks are needed. This makes programs smaller, more robust, and more efficient.

## 1.7 Efficiency

There are no magic rules for making code more efficient. Efficiency is achieved through careful tradeoffs, and these tradeoffs are different for different architectures. However, for the foreseeable future, a good heuristic is that programmers should pay more attention to memory access patterns than to operation counts. This is the opposite of the best heuristic of a decade ago. This switch has occurred because the speed of memory has not kept pace with the speed of processors. Since that trend continues, the importance of limited and coherent memory access for optimization should only increase.

A reasonable approach to making code fast is to proceed in the following order, taking only those steps which are needed:

1. Write the code in the most straightforward way possible. Compute data as needed on the fly without storing it.

2. Compile in optimized mode.

3. Use whatever profiling tools exist to find critical bottlenecks.

4. Examine data structures to look for ways to improve locality. If possible, make data unit sizes match the cache/page size on the target architecture.

5. If profiling reveals bottlenecks in numeric computations, examine the assembly code generated by the compiler for missed efficiencies. Rewrite source code to solve any problems you find.

The most important of these steps is the first one. Most "optimizations" make the code harder to read without speeding things up. In addition, time spent upfront optimizing code is usually better spent correcting bugs or adding features. Also, beware of suggestions from old texts; some classic tricks such as using integers instead of reals may no longer yield speed because some modern CPUs can usually perform floating point operations just as fast as they perform integer operations. In all situations, profiling is needed to be sure of the merit of any optimization for a specific machine and compiler.

## 1.8 Software Engineering

A key part of any graphics program is to have good classes or routines for geometric entities such as vectors and matrices, as well as graphics entities such as RGB colors and images. These routines should be made as clean and efficient as possible. Most graphics programmers use C++, so some discussion of that language is in order. A critical issue is whether locations and displacements should be separate classes because they have different operations, e.g., a location multiplied by one-half makes no geometric sense while one-half of a displacement does (Goldman, 1985). This is a personal decision, but I believe strongly in the KISS ("keep it simple, stupid") principle, and in that light the argument for two classes is not compelling enough to justify the added complexity (for a counter argument see (DeRose, 1989)). This implies that some basic classes that should be written include:

- vector2: A 2D vector class that stores an $x$ and $y$ component. It should store these components in a length-2 array so that an indexing operator can be well supported. You should also include operations for vector addition, vector subtraction, dot product, cross product, scalar multiplication, and scalar division.
- vector3: A 3D vector class analogous to vector2.
- hvector: A homogeneous vector with four components (see Chapter 7).
- rgb: An RGB color that stores three components. You should also include operations for RGB addition, RGB subtraction, RGB multiplication, scalar multiplication, and scalar division.
- transform: A four-by-four matrix for transformations. You should include a matrix multiply and member functions to apply to locations, directions, and surface normal vectors. As shown in Chapter 6, these are all different.
- image: A 2D array of RGB pixels with an output operation.

In addition, you might or might not want to add classes for intervals, orthonormal bases, and coordinate frames. You might also consider unit-length vectors, although I have found them more pain than they are worth. There are several basic decisions to be made which are outlined in the following sections.

### 1.8.1 Float versus Double

Modern architecture suggests that keeping memory use down and maintaining coherent memory access are the keys to efficiency. This suggests using single-precision data. However, avoiding numerical problems suggests using double-precision arithmetic. The tradeoffs depend on the program, but it is nice to have a default in your class definitions. I suggest using doubles for geometric computation and floats for color computation. Where memory usage is high, as it is for triangle meshes, I suggest storing float data, but converting to double when data is accessed through member functions.

### 1.8.2 Inlining

Inlining is a key to efficiency for utility classes such as RGB. Almost all RGB and vector functions should be inlined. Be sure to profile your code to make sure that things are actually being inlined. Non-utility code and other large functions should not be inlined unless the profiler shows them to be hogging runtime. Even then be sure making them inline does not slow the code further. Note, that on most systems, the inline function definitions must be in the header files. For member functions, these can be linked to the declarations, for example:

```
class vector3 {
      .
      .
      .
    double lengthSquared { return x()*x()+y()*y()
        +z()*z(); }
};
```

### 1.8.3 Member Functions versus Non-Member Operators

For operators such as the addition of two vectors, we can make them either a member of a vector class or an operator that exists outside of the class. I suggest that such operators always exist outside of a class. This is because it is the only solution for something like the multiplication operator for a double and a vector (as opposed to vector times double). Since we have to make it a non-member in such cases, we may as well be consistent and always make it a non-member. We should make such operators as compact as possible, for example:

```
inline vector3 operator+(vector3 a, vector3 b) {
    return vector3( a.x() + b.x(), a.y()
        + b.y(), a.z() + b.z() );
}
```

Note that for non-inlined operators and for some compilers, using a `const` reference for argument passing avoids some data copying:

```
inline vector3 operator+(const vector3& a, const vector3& b) {
    return vector3( a.x() + b.x(), a.y()
        + b.y(), a.z() + b.z() );
}
```

### 1.8.4 Include Guards

All classes should have include guards surrounding the class declarations. The names of these guards should follow some simple naming convention. For example:

```
#ifndef VECTOR3H
#define VECTOR3H

class vector3 {
        .
        .
        .

};

#endif
```

This prevents problems when a header file is included more than once which is almost unavoidable in practice. Note that when VECTOR3H is already defined, the header file is still opened and one line is read. For large libraries, this file opening can dominate compilation time (Lakos, 1996). In such cases, an ugly, but effective, solution is to add a check when the include is made:

```
#ifndef VECTOR3H
#include <vector3.h>
#endif
```

### 1.8.5 Debugging Compiles

You should generously sprinkle asserts throughout your code. An assert is a macro that stops the program if the Boolean statement it contains is false. For example:

```
#include <assert.h>

assert( fabs( v.length() - 1 ) < 0.00001 );
```

makes sure that $v$ is close to unit length. Asserts are excellent to add during debugging as well as during development. If you ever add one, leave it in. You might well add a bug later that triggers it again. Note that in an optimized run, you need to define the preprocessor variable NDEBUG to turn off the asserts. This is typically accomplished with the compiler flag -DNDEBUG. When compiling in debugging mode, variables can be set to illegal values such as NaN so that uninitialized variables crash the program when used.

### 1.8.6 Experimental Debugging

If you ask around, you may find that as programmers become more experienced, they use traditional debuggers less and less. One reason for this is that using such debuggers is more awkward for complex programs than for simple programs. Another reason is that the most difficult errors are conceptual ones where the wrong thing is being implemented, and it is easy to waste large amounts of time stepping through variable values without detecting such cases.

In graphics programs there is an alternative to traditional debugging that is often very useful. The downside to it is that it is very similar to what computer programmers are taught not to do early in their careers, so you may feel "naughty" if you do it: we create an image and observe what is wrong with it. Then, we develop a hypothesis about what is causing the problem and test it. For example, in a ray tracing program we might have many somewhat random looking dark pixels. This is the classic "shadow acne" problem that most people run into when they write a ray tracer. Traditional debugging is not helpful here; instead, we must realize that the shadow rays are hitting the surface being shaded. We might notice that the color of the dark spots is the ambient color, so the direct lighting is what is missing. Direct lighting can be turned off in shadow, so you might hypothesize that these points are incorrectly being tagged as in shadow when they are not. To test this hypothesis, we could turn off the shadowing check and recompile. This would indicate that these are false shadow tests, and we could continue our detective work. The key reason this method can sometimes be good practice is that we never had to spot a false value or really determine our conceptual error. Instead, we just narrowed in on our conceptual error experimentally. Typically only a few trials are needed to track things down, and this type of debugging is enjoyable.

In the cases where the program crashes, a traditional debugger is useful for pinpointing the site of the crash. You should then start backtracking in the program, using asserts and recompiles, to find where the program went wrong. These asserts should be left in the program for potential future bugs you will add. This again means the traditional step-though process is avoided, because that would not be adding the valuable asserts to your program.

## Notes

The discussion of software engineering is influenced by the *Effective C++* series (Meyers, 1995, 1997), the *Extreme Programming* movement (Beck & Andres, 2004), and (Kernighan & Pike, 1999). The discussion of experimental debugging is based on discussions with Steve Parker. There are a number of annual conferences related to computer graphics, and these can be found by doing web searches by their title:

- ACM SIGGRAPH Conference
- Graphics Interface Conference
- Game Developers' Conference (GDC)
- Eurographics Conference
- Pacific Graphics Conference
- Eurographics Symposium on Rendering
- Solid Modeling Conference
- IEEE Visualization Conference

# 2

# Miscellaneous Math

Much of graphics is just translating math directly into code. The cleaner the math, the cleaner the resulting code. Thus, much of this book concentrates on using just the right math for the job. This chapter reviews various tools from high school and college mathematics, and is designed to be used more as a reference than as a tutorial. It may appear to be a hodge-podge of topics, and indeed it is; each topic is chosen because it is a bit unusual in "standard" math curricula, because it is of central importance in graphics, or because it is not typically treated from a geometric standpoint. In addition to establishing a review with the notation used in the book, the chapter also emphasizes a few points that are sometimes skipped in the standard undergraduate curricula, such as barycentric coordinates on triangles. This chapter is not intended to be a rigorous treatment of the material; instead intuition and geometric interpretation are emphasized. A discussion of linear algebra is deferred until Chapter 5 just before transformation matrices are discussed. Readers are encouraged to skim this chapter to familiarize themselves with the topics covered and to refer back to it as needed. The exercises at the end of the chapter may be useful in determining which topics need a refresher.

## 2.1 Sets and Mappings

*Mappings*, also called *functions*, are basic to mathematics and programming. Like a function in a program, a mapping in math takes an argument of one *type* and maps it to (returns) an object of a particular type. In a program we say "type;" in math we would identify the set. When we have an object that is a member of a set, we use the $\in$ symbol. For example:

$$a \in \mathbf{S},$$

can be read "$a$ is a member of set $\mathbf{S}$." Given any two sets $\mathbf{A}$ and $\mathbf{B}$, we can create a third set by taking the *Cartesian product* of the two sets, denoted $\mathbf{A} \times \mathbf{B}$. This set $\mathbf{A} \times \mathbf{B}$ is composed of all possible ordered pairs $(a, b)$ where $a \in \mathbf{A}$ and $b \in \mathbf{B}$. As a shorthand, we use the notation $\mathbf{A}^2$ to denote $\mathbf{A} \times \mathbf{A}$. We can extend the Cartesian product to create a set of all possible ordered triples from three sets, and so on for arbitrarily long ordered tuples from arbitrarily many sets.

Common sets of interest include:

- $\mathbb{R}$: the real numbers.
- $\mathbb{R}^+$: the non-negative real numbers (includes zero).
- $\mathbb{R}^2$: the ordered pairs in the real 2D plane.
- $\mathbb{R}^n$: the points in n-dimensional Cartesian space.
- $\mathbb{Z}$: the integers.
- $S^2$: the set of 3D points (points in $\mathbb{R}^3$) on the unit sphere.

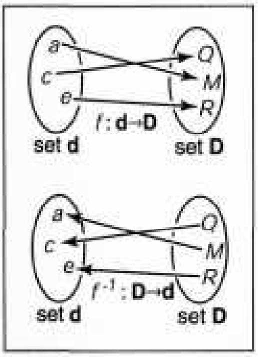


**Figure 2.1.** A bijection $f$ and the inverse function $f^{-1}$. Note that $f^{-1}$ is also a bijection.

Note that although $S^2$ is composed of points embedded in three-dimensional space, they are on a surface that can be parameterized with two variables, so it can be thought of as a 2D set. Notation for mappings uses the arrow and a colon, for example:

$$f : \mathbb{R} \mapsto \mathbb{Z},$$

which you can read: "There is a function called $f$ that takes a real number as input and maps it to an integer." Here, the set that comes before the arrow is called the *domain* of the function, and the set on the right-hand side is called the *target*. The subset of the target that contains all image points under the function (i.e., points in $\mathbb{Z}$ so that there exists a point in $\mathbb{R}$) is called the *range* of the function. Computer programmers might be more comfortable with the equivalent language: "There is a function called $f$ which has one real argument and returns an integer". In other words, the set notation above is equivalent to the common programming notation:

$$\text{integer } f(\text{real}) \quad \leftarrow \text{equivalent} \rightarrow \quad f : \mathbb{R} \mapsto \mathbb{Z}.$$

So the colon-arrow notation can be thought of as a programming syntax. It's that simple.

### 2.1.1 Inverse Mappings

If we have a function $f : \mathbf{A} \mapsto \mathbf{B}$, there may exist an *inverse function* $f^{-1} : \mathbf{B} \mapsto \mathbf{A}$, which is defined by the rule $f^{-1}(b) = a$ where $b = f(a)$. This definition only works if each $b \in B$ is an image point under $f$ and if there is only one point $a$ such that $f(a) = b$. Such mappings or functions are called *bijections*. A bijection maps every $a \in \mathbf{A}$ to a unique $b \in \mathbf{B}$, and for every $b \in \mathbf{B}$, there is exactly one $a \in \mathbf{A}$ such that $f(a) = b$ (Figure 2.1). A bijection between a group of riders and horses indicates that everybody rides a single horse, and every horse is ridden. The two functions would be *rider(horse)* and *horse(rider)*. These are inverse functions of each other. Functions that are not bijections have no inverse (Figure 2.2).

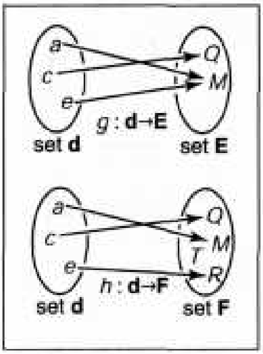

a
c
e
Q
M
$g : \mathbf{d} \to \mathbf{E}$
set **d**
set **E**
a
c
e
Q
M
T
R
$h : \mathbf{d} \to \mathbf{F}$
set **d**
set **F**

**Figure 2.2.** The function $g$ does not have an inverse because two elements of $\boldsymbol{d}$ map to the same element of $\boldsymbol{E}$. The function $h$ has no inverse because element $T$ of $\boldsymbol{F}$ has no element of $\boldsymbol{d}$ mapped to it.

An example of a bijection is $f : \mathbb{R} \mapsto \mathbb{R}$,with $f(x) = x^3$. The inverse function is $f^{-1}(x) = \sqrt[3]{x}$. This example shows that the standard notation can be somewhat awkward because $x$ is used as a dummy variable in both $f$ and $f^{-1}$. It is sometimes more intuitive to use different dummy variables, with $y = f(x)$ and $x = f^{-1}(y)$. This yields the more intuitive $y = x^3$ and $x = \sqrt[3]{y}$. An example of a function that does not have an inverse is $sqr : \mathbb{R} \mapsto \mathbb{R}$, where $sqr(x) = x^2$. This is true for two reasons: first $x^2 = (-x)^2$, and second no members of the domain map to the negative portions of the target. Note that we can define an inverse if we restrict the domain and range to $\mathbb{R}^+$. Then $\sqrt{x}$ is a valid inverse.

### 2.1.2 Intervals

Often we would like to specify that a function deals with real numbers that are restricted in value. One such constraint is to specify an *interval*. An example of an interval is the real numbers between zero and one, not including zero or one. We denote this $(0, 1)$. Because it does not include its endpoints, this is referred to as an *open interval*. The corresponding *closed interval* is denoted with square brackets: $[0, 1]$. This notation can be mixed, i.e., $[0, 1)$ includes zero but not one. When writing an interval $[a, b]$, we assume that $a \leq b$. The three common ways to represent an interval are shown in Figure 2.3. The Cartesian products of intervals are used often. For example, to indicate that a point $\mathbf{x}$ is in the unit cube in 3D, we say $\mathbf{x} \in [0, 1]^3$.

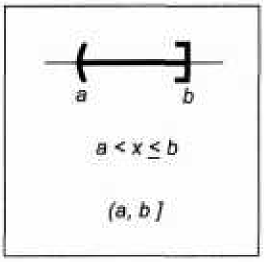


**Figure 2.3.** Three equivalent ways to denote the interval from $a$ to $b$ that includes $b$ but not $a$.

Intervals are particularly useful in conjunction with set operations: *intersection*, *union*, and *difference*. For example, the intersection of two intervals is the set of points they have in common. The symbol $\cap$ is used for intersection. For example, $[3, 5) \cap [4, 6] = [4, 5)$. For unions, the symbol $\cup$ is used to denote points in either interval. For example, $[3, 5) \cup [4, 6] = [3, 6]$. Unlike the first two operators, the difference operator produces different results depending on argument order.

The minus sign is used for the difference operator, which returns the points in the left interval that are not also in the right. For example, $[3,5) - [4,6] = [3,4)$ and $[4,6] - [3,5) = [5,6]$. These operations are particularly easy to visualize using interval diagrams (Figure 2.4).

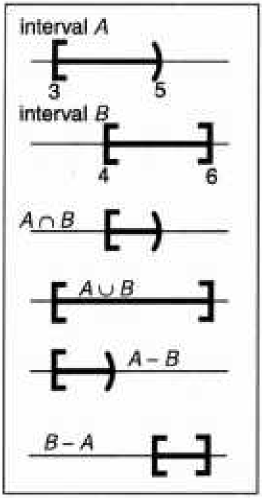


**Figure 2.4.** Interval operations on [3,5) and [4,6].

### 2.1.3 Logarithms

Although not as prevalent as they were before calculators, *logarithms* are often useful in problems where equations with exponential terms arise. By definition, every logarithm has a *base* $a$. The "log base $a$" of $x$ is written $\log_a x$, and is defined as "the exponent to which $a$ must be raised to get $x$," i.e.,

$$y = \log_a x \quad \Leftrightarrow \quad a^y = x.$$

Note that the logarithm base $a$ and the function that raises $a$ to a power are inverses of each other. This basic definition has several consequences:

$$\begin{aligned} a^{\log_a(x)} &= x. \\ \log_a(a^x) &= x. \\ \log_a(xy) &= \log_a x + \log_a y. \\ \log_a(x/y) &= \log_a x - \log_a y. \\ \log_a x &= \log_a b \ \log_b x. \end{aligned}$$

When we apply calculus to logarithms, the special number $e = 2.718\ldots$ turns out to be helpful. The logarithm with base $e$ is called the *natural logarithm*. The natural logarithm arises so often we adopt the common shorthand ln to denote it:

$$\ln x \equiv \log_e x.$$

Note that the "≡" symbol can be read "is equivalent by definition." Like $\pi$, the special number $e$ arises in a remarkable number of contexts. Many fields use a particular base in addition to $e$ for manipulations and omit the base in their notation, i.e., $\log x$. For example, astronomers often use base 10 and theoretical computer scientists often use base 2. Because computer graphics borrows technology from many fields we will avoid this shorthand.

The derivatives of logarithms and exponents illuminate why the natural logarithm is "natural":

$$\begin{aligned} \frac{d}{dx}\log_a x &= \frac{1}{x \ln a}. \\ \frac{d}{dx}a^x &= a^x \ln a. \end{aligned}$$

The constant multipliers above are unity only for $a = e$.

## 2.2 Solving Quadratic Equations

A *quadratic equation* has the form

$$Ax^2 + Bx + C = 0,$$

where $x$ is a real unknown, and $A$, $B$, and $C$ are known constants. If you think of a 2D $xy$ plot with $y = Ax^2 + Bx + C$, the solution is just whatever $x$ values are "zero crossings" in $y$. Because $y = Ax^2 + Bx + C$ is a parabola, there will be zero, one, or two real solutions depending on whether the the parabola misses, grazes, or hits the $x$-axis (Figure 2.5).

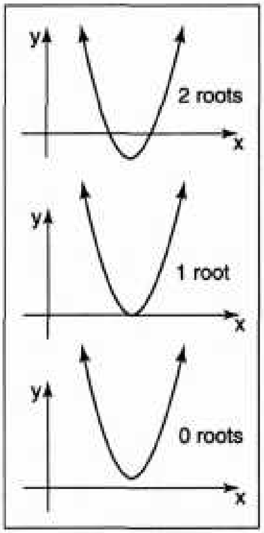


**Figure 2.5.** The geometric interpretation of the roots of a quadratic equation is the intersection points of a parabola with the *x*-axis.

To solve the quadratic equation analytically, we first divide by $A$:

$$x^2 + \frac{B}{A}x + \frac{C}{A} = 0.$$

Then we "complete the square" to group terms:

$$\left(x + \frac{B}{2A}\right)^2 - \frac{B^2}{4A^2} + \frac{C}{A} = 0.$$

Moving the constant portion to the right-hand side and taking the square root gives

$$x + \frac{B}{2A} = \pm\sqrt{\frac{B^2}{4A^2} - \frac{C}{A}}.$$

Subtracting $B/(2A)$ from both sides and grouping terms with the denominator $2A$ gives the familiar form:

$$x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}. \tag{2.1}$$

Here the "±" symbol means there are two solutions, one with a plus sign and one with a minus sign. Thus $3 \pm 1$ equals "two or four". Note that the term which determines the number of real solutions is

$$D \equiv B^2 - 4AC,$$

which is called the *discriminant* of the quadratic equation. If $D > 0$, there are two real solutions (also called *roots*). If $D = 0$, there is one real solution (a "double" root). If $D < 0$, there are no real solutions.

For example, the roots of $2x^2 + 6x + 4 = 0$ are $x = -1$ and $x = -2$, and the equation $x^2 + x + 1$ has no real solutions. The discriminants of these equations are $D = 4$ and $D = -3$, respectively, so we expect the number of solutions given. In programs, it is usually a good idea to evaluate $D$ first, and return "no roots" without taking the square root if $D$ is negative.

## 2.3 Trigonometry

In graphics we use basic trigonometry in many contexts. Usually, it is nothing too fancy, and it often helps to remember the basic definitions.

### 2.3.1 Angles

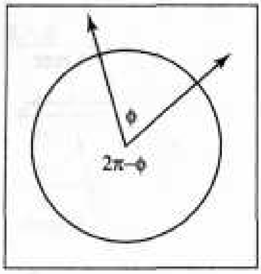


**Figure 2.6.** Two half-lines cut the unit circle into two arcs. The length of either arc is a valid angle "between" the two half-lines. Either we can use the convention that the smaller length is the angle, or that the two half-lines are specified in a certain order and the arc that determines angle $\phi$ is the one swept out counterclockwise from the first to the second half-line.

Although we take angles somewhat for granted, we should return to their definition so we can extend the idea of the angle onto the sphere. An angle is formed between two half-lines (an infinite ray stemming from an origin) or directions, and some convention must be used to decide between the two possibilities for the angle created between them as shown in Figure 2.6. An *angle* is defined by the length of the arc segment it cuts out on the unit circle. A common convention is that the smaller arc length is used, and the sign of the angle is determined by the order in which the two half-lines are specified. Using that convention, all angles are in the range $[-\pi, \pi]$.

Each of these angles is *the length of the arc of the unit circle that is "cut" by the two directions*. Because the perimeter of the unit circle is $2\pi$, the two possible angles sum to $2\pi$. The unit of these arc lengths is *radians*. Another common unit is degrees, where the perimeter of the circle is 360 degrees. Thus, an angle that is $\pi$ radians is 180 degrees, usually denoted $180°$. The conversion between degrees and radians is

$$\text{degrees} = \frac{180}{\pi}\text{ radians};$$

$$\text{radians} = \frac{\pi}{180}\text{ degrees}.$$

### 2.3.2 Trigonometric Functions

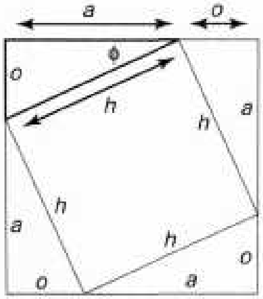


**Figure 2.7.** A geometric demonstration of the Pythagorean theorem.

Given a right triangle with sides of length $a$, $o$, and $h$, where $h$ is the length of the longest side (which is always opposite the right angle), or *hypotenuse*, an important relation is described by the *Pythagorean theorem*:

$$a^2 + o^2 = h^2.$$

You can see that this is true from Figure 2.7, where the big square has area $(a+o)^2$, the four triangles have the combined area $2ao$, and the center square has area $h^2$.

Because the triangles and inner square subdivide the larger square evenly, we have $2ao + h^2 = (a + o)^2$, which is easily manipulated to the form above.

We define *sine* and *cosine* of $\phi$, as well as the other ratio-based trigonometric expressions:

$$\begin{aligned}\sin\phi &\equiv o/h,\\ \csc\phi &\equiv h/o,\\ \cos\phi &\equiv a/h,\\ \sec\phi &\equiv h/a,\\ \tan\phi &\equiv o/a,\\ \cot\phi &\equiv a/o.\end{aligned}$$

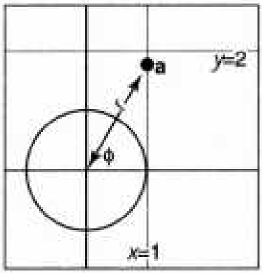


**Figure 2.8.** Polar coordinates for the point $(x_a, y_a) = (1, \sqrt{3})$ is $(r_a, \phi_a) = (2, \pi/3)$.

These definitions allow us to set up *polar coordinates*, where a point is coded as a distance from the origin and a signed angle relative to the positive $x$-axis (Figure 2.8). Note the convention that angles are in the range $\phi \in (-\pi, \pi]$, and that the positive angles are counterclockwise from the positive $x$-axis. This convention that counterclockwise maps to positive numbers is arbitrary, but it is used in many contexts in graphics so it is worth committing to memory.

Trigonometric functions are periodic and can take any angle as an argument. For example $\sin(A) = \sin(A + 2\pi)$. This means the functions are not invertible when considered with the domain $\mathbb{R}$. This problem is avoided by restricting the range of standard inverse functions, and this is done in a standard way in almost all modern math libraries (e.g., (Plauger, 1991)). The domains and ranges are:

$$\begin{aligned}\text{asin} &: [-1, 1] \mapsto [-\pi/2, \pi/2],\\ \text{acos} &: [-1, 1] \mapsto [0, \pi],\\ \text{atan} &: \mathbb{R} \mapsto [-\pi/2, \pi/2],\\ \text{atan2} &: \mathbb{R}^2 \mapsto [-\pi, \pi].\end{aligned} \tag{2.2}$$

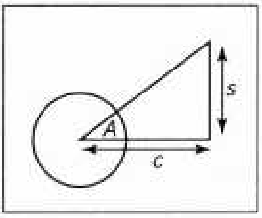


**Figure 2.9.** The function atan2(*s*,*c*) returns the angle *A* and is often very useful in graphics.

The last function, atan2$(s, c)$ is often very useful. It takes an $s$ value proportional to $\sin A$ and a $c$ value that scales $\cos A$ by the same factor, and returns $A$. The factor is assumed to be positive. One way to think of this is that it returns the angle of a 2D Cartesian point $(s, c)$ in polar coordinates (Figure 2.9).

### 2.3.3 Useful Identities

This section lists without derivation a variety of useful trigonometric identities.

Shifting identities:

$$\begin{aligned}\sin(-A) &= -\sin A\\ \cos(-A) &= \phantom{-}\cos A\\ \tan(-A) &= -\tan A\\ \sin(\pi/2 - A) &= \phantom{-}\cos A\\ \cos(\pi/2 - A) &= \phantom{-}\sin A\\ \tan(\pi/2 - A) &= \phantom{-}\cot A\end{aligned}$$

Pythagorean identities:

$$\begin{aligned}\sin^2 A + \cos^2 A &= 1\\ \sec^2 A - \tan^2 A &= 1\\ \csc^2 A - \cot^2 A &= 1\end{aligned}$$

Addition and subtraction identities:

$$\begin{aligned}\sin(A+B) &= \sin A \cos B + \sin B \cos A\\ \sin(A-B) &= \sin A \cos B - \sin B \cos A\\ \sin(2A) &= 2 \sin A \cos A\\ \cos(A+B) &= \cos A \cos B - \sin A \sin B\\ \cos(A-B) &= \cos A \cos B + \sin A \sin B\\ \cos(2A) &= \cos^2 A - \sin^2 A\\ \tan(A+B) &= \frac{\tan A + \tan B}{1 - \tan A \tan B}\\ \tan(A-B) &= \frac{\tan A - \tan B}{1 + \tan A \tan B}\\ \tan(2A) &= \frac{2 \tan A}{1 - \tan^2 A}\end{aligned}$$

Half-angle identities:

$$\begin{aligned}\sin^2(A/2) &= (1 - \cos A)/2\\ \cos^2(A/2) &= (1 + \cos A)/2\end{aligned}$$

Product identities:

$$\begin{aligned}\sin A \sin B &= -(\cos(A+B) - \cos(A-B))/2\\ \sin A \cos B &= \phantom{-}(\sin(A+B) + \sin(A-B))/2\\ \cos A \cos B &= \phantom{-}(\cos(A+B) + \cos(A-B))/2\end{aligned}$$

The following identities are for arbitrary triangles with side lengths $a$, $b$, and $c$, each with an angle opposite it given by $A$, $B$, $C$ respectively (Figure 2.10).

$$\frac{\sin A}{a} = \frac{\sin B}{b} = \frac{\sin C}{c} \qquad \text{(Law of sines)}$$

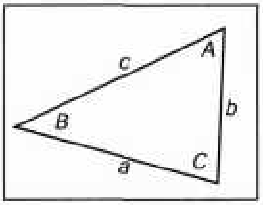


**Figure 2.10.** Geometry for triangle laws.

$$c^2 = a^2 + b^2 - 2ab \cos C \qquad \text{(Law of cosines)}$$

$$\frac{a+b}{a-b} = \frac{\tan\left(\frac{A+B}{2}\right)}{\tan\left(\frac{A-B}{2}\right)} \qquad \text{(Law of tangents)}$$

The area of a triangle can also be computed in terms of these side lengths:

$$\text{triangle area} = \frac{1}{2}\sqrt{(a+b+c)(-a+b+c)(a-b+c)(a+b-c)}.$$

## 2.4 Vectors

A *vector* describes a length and a direction. It can be usefully represented by an arrow. Two vectors are equal if they have the same length and direction even if we think of them as being located in different places (Figure 2.11). As much as possible, you should think of a vector as an arrow and not as coordinates or numbers. At some point we will have to represent vectors as numbers in our programs, but even in code they should be manipulated as objects and only the low-level vector operators should know about their numeric representation (DeRose, 1989). Vectors will be represented as bold characters, e.g., $\mathbf{a}$. A vector's length is denoted $\|\mathbf{a}\|$. A *unit vector* is any vector whose length is one. The *zero vector* is the vector of zero length. The direction of the zero vector is undefined.

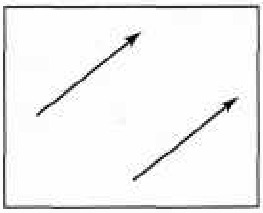


**Figure 2.11.** These two vectors are the same because they have the same length and direction.

Vectors can be used to represent many different things. For example, they can be used to store an *offset*, also called a *displacement*. If we know "the treasure is buried two paces east and three paces north of the secret meeting place," then we know the offset, but we don't know where to start. Vectors can also be used to store a *location*, another word for *position* or *point*. Locations can be represented as a displacement from another location. Usually there is some understood *origin* location from which all other locations are stored as offsets. Note that locations are not vectors. As we shall discuss, you can add two vectors. However, it usually does not make sense to add two locations unless it is an intermediate operation when computing weighted averages of a location (Goldman, 1985). Adding two offsets does make sense, so that is one reason why offsets are vectors. But this emphasizes that a location is not a offset; it is an offset from a specific origin location. The offset by itself is not the location.

### 2.4.1 Vector Operations

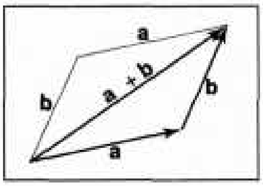


**Figure 2.12.** Two vectors are added by arranging them head to tail. This can be done in either order.

Vectors have most of the usual arithmetic operations that we associate with real numbers. Two vectors are equal if and only if they have the same length and direction. Two vectors are added according to the *parallelogram rule*. This rule states that the sum of two vectors is found by placing the tail of either vector against the head of the other (Figure 2.12). The sum vector is the vector that "completes the triangle" started by the two vectors. The parallelogram is formed by taking the sum in either order. This emphasizes that vector addition is commutative:

$$\mathbf{a} + \mathbf{b} = \mathbf{b} + \mathbf{a}.$$

Note that the parallelogram rule just formalizes our intuition about displacements. Think of walking along one vector, tail to head, and then walking along the other.

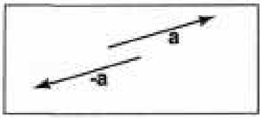


**Figure 2.13.** The vector **-a** has the same length but opposite direction of the vector **a**.

The net displacement is just the parallelogram diagonal. You can also create a *unary minus* for a vector: $-\mathbf{a}$ (Figure 2.13). This is just a vector with the same length but opposite direction. This allows us to also define subtraction:

$$\mathbf{b} - \mathbf{a} \equiv -\mathbf{a} + \mathbf{b}.$$

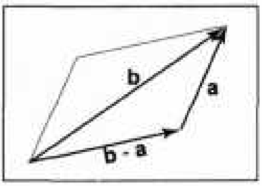


**Figure 2.14.** Vector subtraction is just vector addition with a reversal of the second argument.

You can visualize vector subtraction with a parallelogram (Figure 2.14). We can write

$$\mathbf{a} + (\mathbf{b} - \mathbf{a}) = \mathbf{b}.$$

Vectors can also be multiplied. In fact, there are many ways we can take products involving vectors. First, we can *scale* the vector by multiplying it by a real number $k$. This just multiplies the vector's length without changing its direction. For example, $3.5\mathbf{a}$ is a vector in the same direction as $\mathbf{a}$ but it is 3.5 times as long as $\mathbf{a}$. There are several ways to take the product of two vectors. We later discuss three of them: the dot product, the cross product, and the determinant.

### 2.4.2 Cartesian Coordinates of a Vector

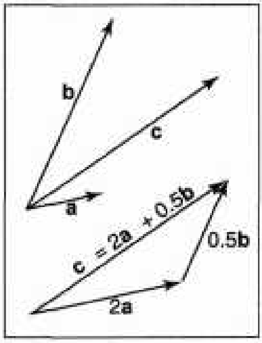


**Figure 2.15.** Any 2D vector ***c*** can be represented by a weighted sum of any two non-parallel 2D vectors **a** and **b**.

A 2D vector can be written as a combination of any two non-zero vectors which are not parallel. This property of the two vectors is called *linear independence*. Two such vectors which are linearly independent form a 2D *basis*, and the vectors are thus referred to as *basis vectors*. For example, a vector $\mathbf{c}$ may be expressed as a combination of two basis vectors $\mathbf{a}$ and $\mathbf{b}$ (Figure 2.15):

$$\mathbf{c} = a_c\mathbf{a} + b_c\mathbf{b}. \tag{2.3}$$

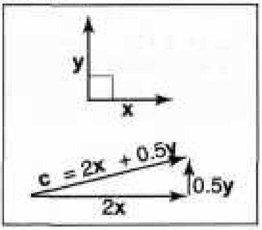


**Figure 2.16.** A 2D Cartesian basis for vectors.

Note that the weights $a_c$ and $b_c$ are unique. This is especially useful if the two vectors are *orthogonal*, i.e., they are at right angles to each other. It is even more useful if they are also unit vectors in which case they are *orthonormal*. If we assume two such "special" vectors $\mathbf{x}$ and $\mathbf{y}$ are known to us, then we can use them to represent all other vectors in a *Cartesian* coordinate system, where each vector is represented as two real numbers. For example, a vector $\mathbf{a}$ might be represented as

$$\mathbf{a} = x_a\mathbf{x} + y_a\mathbf{y},$$

where $x_a$ and $y_a$ are the real Cartesian coordinates of the 2D vector $\mathbf{a}$ (Figure 2.16). Note that this is not really any different conceptually from Equation 2.3, where the basis vectors were not orthonormal. But there is an advantage to a Cartesian coordinate system; by the Pythagorean theorem, the length of $\mathbf{a}$ is

$$\|\mathbf{a}\| = \sqrt{x_a^2 + y_a^2}.$$

By convention we write the coordinates of $\mathbf{a}$ either as an ordered pair $(x_a, y_a)$ or a column matrix:

$$\mathbf{a} = \begin{bmatrix} x_a \\ y_a \end{bmatrix}.$$

Which we use will depend on typographic convenience. We will also occasionally write the vector as a row matrix, which we will indicate as $\mathbf{a}^T$:

$$\mathbf{a}^T = \begin{bmatrix} x_a & y_a \end{bmatrix}.$$

We can also represent 3D, 4D, etc., vectors in Cartesian coordinates. For the 3D case we use a basis vector $\mathbf{z}$ which is orthogonal to both $\mathbf{x}$ and $\mathbf{y}$.

### 2.4.3 Dot Product

The simplest way to multiply two vectors is the *dot* product. The dot product of $\mathbf{a}$ and $\mathbf{b}$ is denoted $\mathbf{a} \cdot \mathbf{b}$ and is often called the *scalar product* because it returns a scalar. The dot product returns a value related to its arguments' length and the angle $\phi$ between them (Figure 2.17):

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \; \|\mathbf{b}\| \; \cos\phi, \tag{2.4}$$

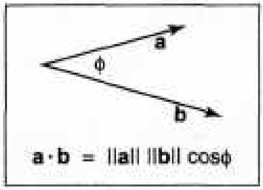


**Figure 2.17.** The dot product is related to length and angle and is one of the most important formulas in graphics.

The most common use of the dot product in graphics programs is to compute the cosine of the angle between two vectors.

The dot product can also be used to find the *projection* of one vector onto another. This is the length $\mathbf{a}{\rightarrow}\mathbf{b}$ of a vector $\mathbf{a}$ that is projected at right angles onto a vector $\mathbf{b}$ (Figure 2.18):

$$\mathbf{a}{\rightarrow}\mathbf{b} = \|\mathbf{a}\| \; \cos\phi = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{b}\|}. \tag{2.5}$$

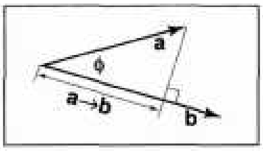


**Figure 2.18.** The projection of **a** onto **b** is a length found by Equation 2.5.

The dot product obeys the familiar associative and distributive properties we have in real arithmetic:

$$\begin{aligned} \mathbf{a} \cdot \mathbf{b} &= \mathbf{b} \cdot \mathbf{a}, \\ \mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) &= \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c}, \\ (k\mathbf{a}) \cdot \mathbf{b} = \mathbf{a} \cdot (k\mathbf{b}) &= k\mathbf{a} \cdot \mathbf{b}. \end{aligned} \tag{2.6}$$

If 2D vectors $\mathbf{a}$ and $\mathbf{b}$ are expressed in Cartesian coordinates, we can take advantage of $\mathbf{x} \cdot \mathbf{x} = \mathbf{y} \cdot \mathbf{y} = 1$ and $\mathbf{x} \cdot \mathbf{y} = 0$ to derive that their dot product is

$$\begin{aligned} \mathbf{a} \cdot \mathbf{b} &= (x_a\mathbf{x} + y_a\mathbf{y}) \cdot (x_b\mathbf{x} + y_b\mathbf{y}) \\ &= x_a x_b(\mathbf{x} \cdot \mathbf{x}) + x_a y_b(\mathbf{x} \cdot \mathbf{y}) + x_b y_a(\mathbf{y} \cdot \mathbf{x}) + y_a y_b(\mathbf{y} \cdot \mathbf{y}) \\ &= x_a x_b + y_a y_b. \end{aligned}$$

Similarly in 3D we can find

$$\mathbf{a} \cdot \mathbf{b} = x_a x_b + y_a y_b + z_a z_b.$$

### 2.4.4 Cross Product

The cross product $\mathbf{a} \times \mathbf{b}$ is usually used only for three-dimensional vectors; generalized cross products are discussed in references given in the chapter notes. The cross product returns a 3D vector that is perpendicular to the two arguments of the cross product. The length of the resulting vector is related to $\sin\phi$:

$$\|\mathbf{a} \times \mathbf{b}\| = \|\mathbf{a}\| \, \|\mathbf{b}\| \sin\phi.$$

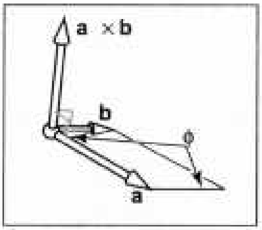


**Figure 2.19.** The cross product **a** × **b** is a 3D vector perpendicular to both 3D vectors **a** and **b**, and its length is equal to the area of the parallelogram shown.

The magnitude $\|\mathbf{a} \times \mathbf{b}\|$ is equal to the area of the parallelogram formed by vectors $\mathbf{a}$ and $\mathbf{b}$. In addition, $\mathbf{a} \times \mathbf{b}$ is perpendicular to both $\mathbf{a}$ and $\mathbf{b}$ (Figure 2.19). Note that there are only two possible directions for such a vector. By definition the vectors in the direction of the *x*-, *y*- and *z*-axes are given by

$$\begin{aligned} \mathbf{x} &= (1, 0, 0), \\ \mathbf{y} &= (0, 1, 0), \\ \mathbf{z} &= (0, 0, 1), \end{aligned}$$

and we set as a convention that $\mathbf{x} \times \mathbf{y}$ must be in the plus or minus $\mathbf{z}$ direction. The choice is somewhat arbitrary, but it is standard to assume that

$$\mathbf{z} = \mathbf{x} \times \mathbf{y}.$$

All possible permutations of the three Cartesian unit vectors are:

$$\begin{aligned} \mathbf{x} \times \mathbf{y} &= +\mathbf{z}, \\ \mathbf{y} \times \mathbf{x} &= -\mathbf{z}, \\ \mathbf{y} \times \mathbf{z} &= +\mathbf{x}, \\ \mathbf{z} \times \mathbf{y} &= -\mathbf{x}, \\ \mathbf{z} \times \mathbf{x} &= +\mathbf{y}, \\ \mathbf{x} \times \mathbf{z} &= -\mathbf{y}. \end{aligned}$$

Because of the $\sin\phi$ property, we also know that a vector cross itself is the zero-vector, so $\mathbf{x} \times \mathbf{x} = \mathbf{0}$ and so on. Note that the cross product is *not* commutative, i.e., $\mathbf{x} \times \mathbf{y} \neq \mathbf{y} \times \mathbf{x}$. The careful observer will note that the above discussion does not allow us to draw an unambiguous picture of how the Cartesian axes relate. More specifically, if we put $\mathbf{x}$ and $\mathbf{y}$ on a sidewalk, with $\mathbf{x}$ pointing East and $\mathbf{y}$ pointing North, then does $\mathbf{z}$ point up to the sky or into the ground? The usual convention is to have $\mathbf{z}$ point to the sky. This is known as a *right-handed* coordinate system. This name comes from the memory scheme of "grabbing" $\mathbf{x}$ with your *right* palm and fingers and rotating it toward $\mathbf{y}$. The vector $\mathbf{z}$ should align with your thumb. This is illustrated in Figure 2.20.

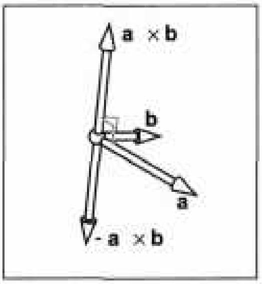


**Figure 2.20.** The "right-hand rule" for cross products. Imagine placing the base of your right palm where **a** and **b** join at their tails, and pushing the arrow of **a** toward **b**. Your extended right thumb should point toward **a** × **b**.

The cross product has the nice property that

$$\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c},$$

and

$$\mathbf{a} \times (k\mathbf{b}) = k(\mathbf{a} \times \mathbf{b}).$$

However, a consequence of the right-hand rule is

$$\mathbf{a} \times \mathbf{b} = -(\mathbf{b} \times \mathbf{a}).$$

In Cartesian coordinates, we can use an explicit expansion to compute the cross product:

$$\begin{aligned}
\mathbf{a} \times \mathbf{b} &= (x_a\mathbf{x} + y_a\mathbf{y} + z_a\mathbf{z}) \times (x_b\mathbf{x} + y_b\mathbf{y} + z_b\mathbf{z}) \\
&= x_a x_b \mathbf{x} \times \mathbf{x} + x_a y_b \mathbf{x} \times \mathbf{y} + x_a z_b \mathbf{x} \times \mathbf{z} \\
&\quad + y_a x_b \mathbf{y} \times \mathbf{x} + y_a y_b \mathbf{y} \times \mathbf{y} + y_a z_b \mathbf{y} \times \mathbf{z} \\
&\quad + z_a x_b \mathbf{z} \times \mathbf{x} + z_a y_b \mathbf{z} \times \mathbf{y} + z_a z_b \mathbf{z} \times \mathbf{z} \\
&= (y_a z_b - z_a y_b)\mathbf{x} + (z_a x_b - x_a z_b)\mathbf{y} + (x_a y_b - y_a x_b)\mathbf{z}.
\end{aligned} \tag{2.7}$$

So, in coordinate form,

$$\mathbf{a} \times \mathbf{b} = (y_a z_b - z_a y_b, z_a x_b - x_a z_b, x_a y_b - y_a x_b). \tag{2.8}$$

### 2.4.5 Orthonormal Bases and Coordinate Frames

Managing coordinate systems is one of the core tasks of almost any graphics program. Key to this is managing *orthonormal bases*. Any set of two 2D vectors $\mathbf{u}$ and $\mathbf{v}$ form an orthonormal basis provided they are orthogonal (at right angles) and are each of unit length. Thus,

$$\|\mathbf{u}\| = \|\mathbf{v}\| = 1,$$

and

$$\mathbf{u} \cdot \mathbf{v} = 0.$$

In 3D, three vectors $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ form an orthonormal basis if

$$\|\mathbf{u}\| = \|\mathbf{v}\| = \|\mathbf{w}\| = 1,$$

and

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{v} \cdot \mathbf{w} = \mathbf{w} \cdot \mathbf{u} = 0.$$

This orthonormal basis is *right-handed* provided

$$\mathbf{w} = \mathbf{u} \times \mathbf{v},$$

and otherwise it is left-handed.

Note that the Cartesian canonical orthonormal basis is just one of infinitely many possible orthonormal bases. What makes it special is that it and its implicit origin location are used for low-level representation within a program. Thus, the vectors $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ are never explicitly stored, and neither is the canonical origin location $\mathbf{o}$. The global model is typically stored in this canonical coordinate system, and it is thus often called the *global coordinate system*. However, if we

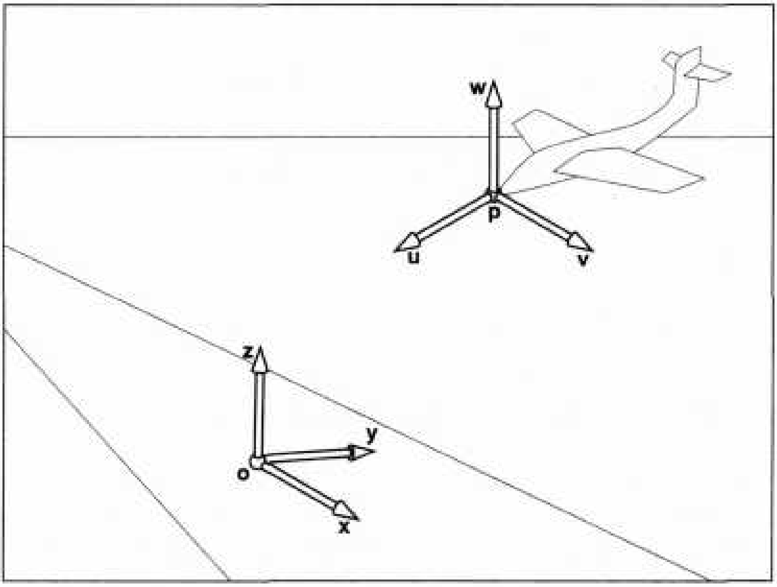


**Figure 2.21.** There is always a master or "canonical" coordinate system with origin **o** and orthonormal basis **x**, **y**, and **z**. This coordinate system is usually defined to be aligned to the global model and is thus often called the "global" or "world " coordinate system. This origin and basis vectors are never stored explicitly. All other vectors and locations are stored with coordinates that relate them to the global frame. The coordinate system associated with the plane are explicitly stored in terms of global coordinates.

want to use another coordinate system with origin $\mathbf{p}$ and orthonormal basis vectors $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$, then we *do* store those vectors explicitly. Such a system is called a *frame of reference* or *coordinate frame*. For example, in a flight simulator, we might want to maintain a coordinate system with the origin at the nose of the plane, and the orthonormal basis aligned with the airplane. Simultaneously we would have the master canonical coordinate system (Figure 2.21). The coordinate system associated with a particular object, such as the plane, is usually called a *local coordinate system*.

At a low level, the local frame is stored in canonical coordinates. For example,

$$\mathbf{u} = (x_u, y_u, z_u) \equiv x_u\mathbf{x} + y_u\mathbf{y} + z_u\mathbf{z}.$$

A location implicitly includes an offset from the canonical origin:

$$\mathbf{p} = (x_p, y_p, z_p) \equiv \mathbf{o} + x_p\mathbf{x} + y_p\mathbf{y} + z_p\mathbf{z}.$$

Note that if we store a vector $\mathbf{a}$ with respect to the *uvw* frame, we store a triple $(u_a, v_a, w_a)$ which we can interpret geometrically as:

$$(u_a, v_a, w_a) \equiv u_a\mathbf{u} + v_a\mathbf{v} + w_a\mathbf{w}.$$

To get the canonical coordinates of a vector $\mathbf{a}$ stored in the $\mathbf{u}$ $\mathbf{v}$ $\mathbf{w}$ coordinate system, simply recall that $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ are themselves stored in terms of Cartesian coordinates, so the expression $u_a\mathbf{u} + v_a\mathbf{v} + w_a\mathbf{w}$ is already in Cartesian coordinates if evaluated explicitly. Using matrices to manage changes of coordinate systems is discussed in Sections 6.2.1 and 6.5.

### 2.4.6 Constructing a Basis from a Single Vector

Often we must construct an orthonormal basis from a single vector, i.e., given a vector $\mathbf{a}$, we want an orthonormal $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ such that $\mathbf{w}$ points in the same direction as $\mathbf{a}$ (Hughes & Möller, 1999). This cannot be done uniquely, but typically all we need is a robust procedure to find any one of the possible bases. This can be done as follows:

$$\mathbf{w} = \frac{\mathbf{a}}{\|\mathbf{a}\|.}$$

To get $\mathbf{u}$ and $\mathbf{v}$, we need to find a vector $\mathbf{t}$ that is not collinear with $\mathbf{w}$. To do this, simply set $\mathbf{t}$ equal to $\mathbf{w}$ and change the smallest magnitude component of $\mathbf{t}$ to 1. For example, if $\mathbf{w} = (1/\sqrt{2}, -1/\sqrt{2}, 0)$ then $\mathbf{t} = (1/\sqrt{2}, -1/\sqrt{2}, 1)$. The $\mathbf{u}$ and $\mathbf{v}$ follow easily:

$$\mathbf{u} = \frac{\mathbf{t} \times \mathbf{w}}{\|\mathbf{t} \times \mathbf{w}\|,}$$
$$\mathbf{v} = \mathbf{w} \times \mathbf{u}.$$

## 2.5 2D Implicit Curves

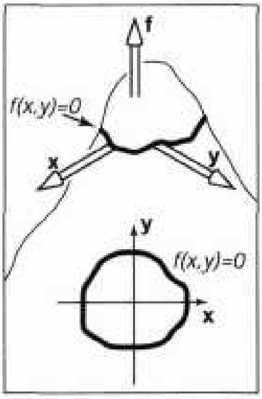


**Figure 2.22.** An implicit function *f(x,y) = 0* can be thought of as a height field where *f* is the height (top). A path where the height is zero is the implicit curve (bottom).

Intuitively, a *curve* is a set of points that can be drawn on a piece of paper without lifting the pen. A common way to describe a curve is using an *implicit equation*. An implicit equation in two dimensions has the form

$$f(x, y) = 0.$$

The function $f(x, y)$ returns a real value. Points $(x, y)$ where this value is zero are on the curve, and points where the value is non-zero are not on the curve. For example, let's say that $f(x, y)$ is

$$f(x, y) = (x - x_c)^2 + (y - y_c)^2 - r^2, \tag{2.9}$$

where $(x_c, y_c)$ is a 2D point and $r$ is a non-zero real number. If we take $f(x, y) = 0$, the points where this equality hold are on the circle with center $(x_c, y_c)$ and radius $r$. The reason that this is called an "implicit" equation is that the points $(x, y)$ on the curve cannot be immediately calculated from the equation, and instead must be determined by plugging $(x, y)$ into $f$ and finding out whether it is zero or by solving the equation. Thus, the points on the curve are not generated by the equation *explicitly*, but they are buried somewhere *implicitly* in the equation.

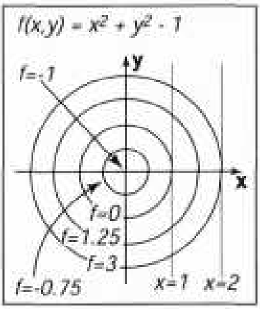


**Figure 2.23.** An implicit function *f(x,y) = 0* can be thought of as a height field where *f* is the height (top). A path where the height is zero is the implicit curve (bottom).

It is interesting to note that $f$ does have values for all $(x, y)$. We can think of $f$ as a terrain, with sea-level at $f = 0$ (Figure 2.22). The shore is the implicit curve. The value of $f$ is the altitude. Another thing to note is that the curve partitions space into regions where $f > 0$, $f < 0$ and $f = 0$. So you evaluate $f$ to decide whether a point is "inside" a curve. Note that $f(x, y) = c$ is a curve for any constant $c$, and $c = 0$ is just used as a convention. For example if $f(x, y) = x^2 + y^2 - 1$, varying $c$ just gives a variety of circles centered at the origin (Figure 2.23).

We can compress our notation using vectors. If we have $\mathbf{c} = (x_c, y_c)$ and $\mathbf{p} = (x, y)$, then our circle with center $\mathbf{c}$ and radius $r$ is defined by those position vectors that satisfy

$$(\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) - r^2 = 0.$$

This equation, if expanded algebraically, will yield Equation 2.9, but it is easier to see that this is an equation for a circle by "reading" the equation geometrically. It reads, "points $\mathbf{p}$ on the circle have the following property: the vector from $\mathbf{c}$ to $\mathbf{p}$ when dotted with itself has value $r^2$." It is also easier to implement vector equations than implementing fully expanded equations if you implement a vector type in your code; the cut-and-paste errors involving $x$, $y$, and $z$ will go away. Because a vector dotted with itself is just its own length squared, we could also read the equation as, "points $\mathbf{p}$ on the circle have the following property: the vector from $\mathbf{c}$ to $\mathbf{p}$ has squared length $r^2$."

Even better, is to observe that the squared length is just the squared distance from $\mathbf{c}$ to $\mathbf{p}$, which suggests the equivalent form

$$\|\mathbf{p} - \mathbf{c}\|^2 - r^2 = 0,$$

and, of course, this suggests

$$\|\mathbf{p} - \mathbf{c}\| - r = 0.$$

The above could be read "the points $\mathbf{p}$ on the circle are those a distance $r$ from the center point $\mathbf{c}$," which is as good a definition of circle as any. This illustrates that the vector form of an equation often suggests more geometry and intuition than the equivalent full-blown Cartesian form with $x$s and $y$s. For this reason, it is usually advisable to use vector forms when possible. In addition, you can support a vector class in your code; the code is cleaner when vector forms are used. It takes a little while to get used to vectors in these equations, but once you get the hang of it, the payoff is large.

### 2.5.1 The 2D Gradient

If we think of the function $f(x, y)$ as a height field with height $= f(x, y)$, the *gradient* vector points in the direction of maximum upslope, i.e., straight uphill. The gradient vector $\nabla f(x, y)$ is given by

$$\nabla f(x, y) = \left( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right).$$

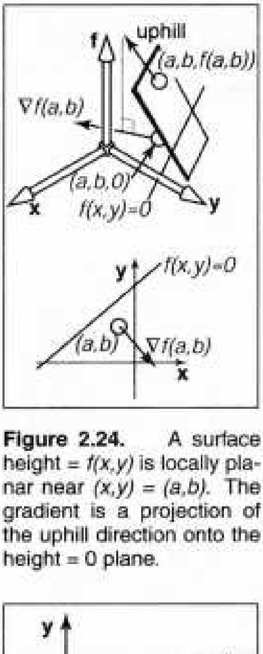


**Figure 2.24.** A surface height = *f(x,y)* is locally planar near *(x,y) = (a,b)*. The gradient is a projection of the uphill direction onto the height = 0 plane.

The gradient vector evaluated at a point on the implicit curve $f(x, y) = 0$ is perpendicular to the *tangent* vector of the curve at that point. This perpendicular vector is usually called the *normal vector* to the curve. In addition, since the gradient points uphill, it indicates the direction of the $f(x, y) > 0$ region.

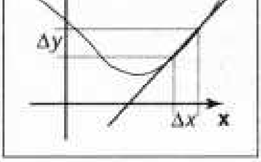


**Figure 2.25.** The derivative of a 1D function measures the slope of the line tangent to the curve.

In the context of height fields, the geometric meaning of partial derivatives and gradients is more visible than usual. Suppose that near the point $(a, b)$, $f(x, y)$ is a plane (Figure 2.24). There is a specific uphill and downhill direction. At right angles to this direction is a direction that is level with respect to the plane. Any intersection between the plane and the $f(x, y) = 0$ plane will be in the direction that is level. Thus the uphill/downhill directions will be perpendicular to the line of intersection $f(x, y) = 0$. To see why the partial derivative has something to do with this, we need to visualize its geometric meaning. Recall that the conventional derivative of a 1D function $y = g(x)$ is

$$\frac{dy}{dx} \equiv \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{g(x + \Delta x) - g(x)}{\Delta x}. \tag{2.10}$$

What this measures is is the *slope* of the *tangent* line to $g$ (Figure 2.25).

The partial derivative is a generalization of the 1D derivative. For a 2D function $f(x, y)$, we can't take the same limit for $x$ as in Equation 2.10, because $f$ can change in many ways for a given change in $x$. However, if we hold $y$ constant, we can define an analog of the derivative, called the *partial derivative* (Figure 2.26):

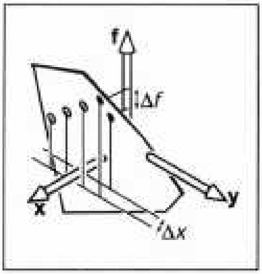


**Figure 2.26.** The partial derivative of a 2D function with respect to $f$ must hold $y$ constant to have a unique value, as shown by the dark point. The hollow points show other values of $f$ that do not hold $y$ constant.

$$\frac{\partial f}{\partial x} \equiv \lim_{\Delta x \to 0} \frac{f(x + \Delta x, y) - f(x, y)}{\Delta x}.$$

Why is it that the partial derivatives of $x$ and $y$ are the components of the gradient vector? Again, there is more obvious insight in the geometry than in the algebra. In Figure 2.27, we see the vector **a** travels along a path where **f** does not change. Note that this is again at a small enough scale that the surface height$(x, y) = f(x, y)$ can be considered locally planar. From the figure, we see that the vector $\mathbf{a} = (\Delta x, \Delta y)$.

Because the uphill direction is perpendicular to **a**, we know the dot product is equal to zero:

$$(\nabla f) \cdot \mathbf{a} \equiv (x_\nabla, y_\nabla) \cdot (x_a, y_a) = x_\nabla \Delta x + y_\nabla \Delta y = 0. \tag{2.11}$$

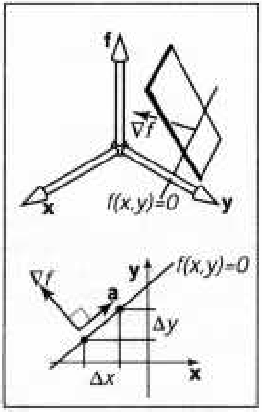


**Figure 2.27.** The vector **a** points in a direction where $f$ has no change and is thus perpendicular to the gradient vector $\nabla f$.

We also know that the change in $f$ in the direction $(x_a, y_a)$ equals zero:

$$\Delta f = \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial y}\Delta y \equiv \frac{\partial f}{\partial x}x_a + \frac{\partial f}{\partial y}y_a = 0. \tag{2.12}$$

Given any vectors $(x, y)$ and $(x', y')$ that are perpendicular, we know the angle between them is 90 degrees, and thus their dot product equals zero (recall that the dot product is proportional to the cosine of the angle between the two vectors). Thus we have $xx' + yy' = 0$. Given $(x, y)$, it is easy to construct valid vectors whose dot product with $(x, y)$ equals zero, the two most obvious being $(y, -x)$ and $(-y, x)$; you can verify that these vectors give the desired zero dot product with $(x, y)$. A generalization of this observation is that $(x, y)$ is perpendicular to $k(y, -x)$ where $k$ is any non-zero constant. This implies that

$$(x_a, y_a) = k\left(\frac{\partial f}{\partial y}, -\frac{\partial f}{\partial x}\right). \tag{2.13}$$

Combining Equations 2.11 and 2.13 gives

$$(x_\nabla, y_\nabla) = k'\left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right),$$

where $k'$ is any non-zero constant. By definition, "uphill" implies a positive change in $f$, so we would like $k' > 0$, and $k' = 1$ is a perfectly good convention.

As an example of the gradient, consider the implicit circle $x^2 + y^2 - 1 = 0$ with gradient vector $(2x, 2y)$, indicating that the outside of the circle is the positive region for the function $f(x, y) = x^2 + y^2 - 1$. Note that the length of the gradient vector can be different depending on the multiplier in the implicit equation. For example, the unit circle can be described by $Ax^2 + Ay^2 - A = 0$ for any non-zero $A$. The gradient for this curve is $(2Ax, 2Ay)$. This will be normal (perpendicular) to the circle, but will have a length determined by $A$. For $A > 0$, the normal will point outward from the circle, and for $A < 0$, it will point inward. This switch from outward to inward is as it should be, since the positive region switches inside the circle. In terms of the height-field view, $h = Ax^2 + Ay^2 - A$, the circle is the zero altitude point. For $A > 0$, the circle encloses a depression, and for $A < 0$, the circle encloses a bump. As $A$ becomes more negative, the bump increases in height, but the $h = 0$ circle doesn't change. The direction of maximum uphill doesn't change, but the slope increases. The length of the gradient reflects this change in degree of the slope. So intuitively, you can think of the gradient's direction as pointing uphill and its magnitude as measuring how uphill the slope is.

### 2.5.2 Implicit 2D Lines

The familiar "slope-intercept" form of the line is

$$y = mx + b. \tag{2.14}$$

This can be converted easily to implicit form (Figure 2.28):

$$y - mx - b = 0. \tag{2.15}$$

Here $m$ is the "slope" (ratio of rise to run) and $b$ is the $y$ value where the line crosses the $y$-axis, usually called the *y-intercept* . The line also partitions the 2D plane, but here "inside" and "outside" might be more intuitively called "over" and "under."

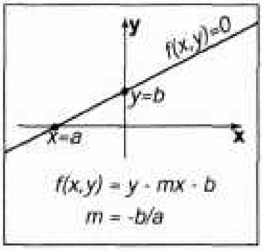


**Figure 2.28.** A 2D line can be described by the equation $y - mx - b = 0$.

Because we can multiply an implicit equation by any constant without changing the points where it is zero, $kf(x, y) = 0$ is the same curve for any non-zero $k$. This allows several implicit forms for the same line, for example,

$$2y - 2mx - 2b = 0.$$

One reason the slope-intercept form is sometimes awkward is that it can't represent some lines such as $x = 0$ because $m$ would have to be infinite. For this reason, a more general form is often useful:

$$Ax + By + C = 0, \tag{2.16}$$

for real numbers $A$, $B$, $C$. Suppose we know two points on the line $(x_0, y_0)$ and $(x_1, y_1)$. Because these points lie on the line, they must both satisfy Equation 2.16:

$$Ax_0 + By_0 + C = 0,$$
$$Ax_1 + By_1 + C = 0.$$

Unfortunately we have two equations and *three* unknowns: $A, B, C$. This problem arises because of the arbitrary multiplier we can have with an implicit equation. We could set $C = 1$ for convenience:

$$Ax + By + 1 = 0,$$

but we have a similar problem to the infinite slope case in slope-intercept form: lines through the origin will have $A(0) + B(0) + 1 = 0$ so $A$ or $B$ has to be infinite. For example, the equation for a 45 degree line through the origin can be written $x - y = 0$, or equally well $y - x = 0$, or even $17y - 17x = 0$, but it cannot be written in the form $Ax + By + 1 = 0$.

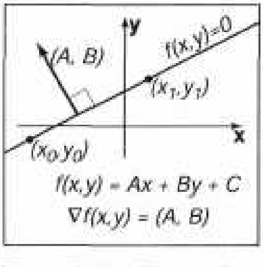


**Figure 2.29.** The gradient vector *(A,B)* is perpendicular to the implicit line *Ax + By + C = 0.*

Whenever we have such pesky algebraic problems, try to solve the problems using geometric intuition as a guide. One tool we have, as discussed in Section 2.5.1, is the gradient. For the line $Ax + By + C = 0$, the gradient vector is $(A, B)$. This vector is perpendicular to the line (Figure 2.29), and points to the side of the line where $Ax + By + C$ is positive. Given two points on the line $(x_0, y_0)$ and $(x_1, y_1)$, we know that the vector between them points in the same direction as the line. This vector is just $(x_1 - x_0, y_1 - y_0)$, and because it is parallel to the line, it must also be perpendicular to the gradient vector $(A, B)$. Recall that there are an infinite number of $(A, B, C)$ that describe the line because of the arbitrary scaling property of implicits. We want any one of the valid $(A, B, C)$.

We can start with any $(A, B)$ perpendicular to $(x_1 - x_0, y_1 - y_0)$. Such a vector is just $(A, B) = (y_0 - y_1, x_1 - x_0)$ by the same reasoning as in Section 2.5.1. This means the equation of the line through $(x_0, y_0)$ and $(x_1, y_1)$ is

$$(y_0 - y_1)x + (x_1 - x_0)y + C = 0. \tag{2.17}$$

Now we just need to find $C$. Because $(x_0, y_0)$ and $(x_1, y_1)$ are on the line, they must satisfy Equation 2.17. We can plug either in and solve for $C$. Doing this for $(x_0, y_0)$ yields $C = x_0y_1 - x_1y_0$, and thus the full equation for the line is

$$(y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 = 0. \tag{2.18}$$

Again, this is one of infinitely many valid implicit equations for the line through two points, but this form has no division operation, and thus no numerically degenerate cases for points with finite Cartesian coordinates. A nice thing about

Equation 2.18 is that we can always convert to slope-intercept form by moving the non-$y$ terms to the right-hand side of the equation, and dividing by the multiplier of the $y$ term:

$$y = \frac{y_1 - y_0}{x_1 - x_0}x + \frac{x_1 y_0 - x_0 y_1}{x_1 - x_0}.$$

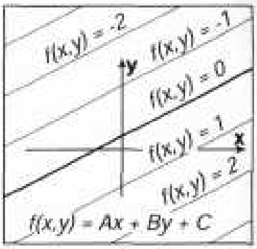


**Figure 2.30.** The value of the implicit function *f(x,y) = Ax + By + C* is a constant times the signed distance from *Ax + By +C = 0.*

An interesting property of the implicit line equation is that it can be used to find the signed distance from a point to the line. The value of $Ax + By + C$ is proportional to the distance from the line (Figure 2.30). As shown in Figure 2.31, the distance from a point to the line is the length of the vector $k(A, B)$, which is

$$\text{distance} = k\sqrt{A^2 + B^2}. \tag{2.19}$$

For the point $(x, y) + k(A, B)$, the value of $f(x, y) = Ax + By + C$ is

$$\begin{aligned} f(x + kA, y + kB) &= Ax + kA^2 + By + kB^2 + C \\ &= k(A^2 + B^2). \end{aligned} \tag{2.20}$$

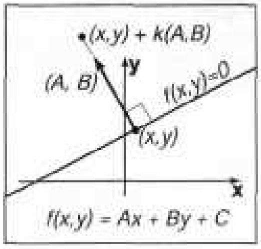


**Figure 2.31.** The vector *k(A,B)* connects a point *(x,y)* on the line closest to a point not on the line. The distance is proportional to *k*.

The simplification in that equation is a result of the fact that we know $(x, y)$ is on the line, so $Ax + By + C = 0$. From Equations 2.19 and 2.20, we can see that the signed distance from line $Ax + By + C = 0$ to a point $(a, b)$ is

$$\text{distance} = \frac{f(a, b)}{\sqrt{A^2 + B^2}}.$$

Here "signed distance" means that its magnitude (absolute value) is the distance, but it may be positive or negative. On one side of the line, distances are positive, and on the other they are negative. Note that if $(A, B)$ is a unit vector, then $f(a, b)$ is the signed distance. We can multiply Equation 2.18 by a constant that ensures $(A, B)$ is a unit vector:

$$\begin{aligned} f(x, y) = \frac{y_0 - y_1}{\sqrt{(x_1 - x_0)^2 + (y_0 - y_1)^2}}x &+ \frac{x_1 - x_0}{\sqrt{(x_1 - x_0)^2 + (y_0 - y_1)^2}}y \\ &+ \frac{x_0 y_1 - x_1 y_0}{\sqrt{(x_1 - x_0)^2 + (y_0 - y_1)^2}} = 0. \end{aligned} \tag{2.21}$$

Note that evaluating $f(x, y)$ in Equation 2.21 directly gives the signed distance, but does require a square root to set up the equation. Note also that although distances on one side of the line are positive, those on the other side of the line are negative. You can choose between the equally valid representations $f(x, y) = 0$ and $-f(x, y) = 0$ if your problem has some reason to prefer a particular side being positive. This will turn out to be very useful for triangle rasterization (Section 3.6). Other forms for 2D lines are discussed in Chapter 14.

### 2.5.3 Implicit Quadric Curves

For 2D quadric curves, i.e., ellipses and parabolas, as well as the special cases of hyperbolas, circles, and lines, we have the general implicit form:

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0.$$

The equation for the circle with center $(x_c, y_c)$ and radius $r$ is

$$(x - x_c)^2 + (y - y_c)^2 - r^2 = 0.$$

Equations for parabolas include

$$y - k(x - x_c)^2 = 0,$$

where $k$ is a non-zero constant and $x_c$ is the axis of symmetry for the parabola. There is an analogous form for parabolas with horizontal axes of symmetry. The equation for an axis-aligned ellipse is

$$\frac{(x - x_c)^2}{a^2} + \frac{(y - y_c)^2}{b^2} - 1 = 0,$$

where $(x_c, y_c)$ is the center of the ellipse, and $a$ and $b$ are the minor and major semi-axes (Figure 2.32).

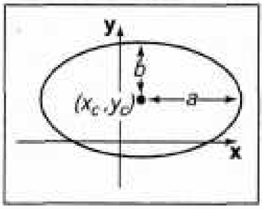


**Figure 2.32.** The ellipse with center *($x_c$, $y_c$)* and semi-axes of length *a* and *b*.

## 2.6 2D Parametric Curves

A *parametric* curve is controlled by a single *parameter* that can be considered a sort of index that moves continuously along the curve. Such curves have the form

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} g(t) \\ h(t) \end{bmatrix}.$$

Here $(x, y)$ is a point on the curve, and $t$ is the parameter that influences the curve. For a given $t$, there will be some point determined by the functions $g$ and $h$. For continuous $g$ and $h$, a small change in $t$ will yield a small change in $x$ and $y$. Thus, as $t$ continuously changes, points are swept out in a continuous curve. This is a nice feature because we can use the parameter $t$ to explicitly construct points on the curve. Often we can write a parametric curve in vector form:

$$\mathbf{p} = f(t),$$

where $f$ is a vector valued function: $f : \mathbb{R} \mapsto \mathbb{R}^2$. Such vector functions can generate very clean code, so they should be used when possible. Note that we can think of the curve with a position as a function of time. The curve can go anywhere and could loop and cross itself. We can also think of the curve as having a velocity at any point. For example, the point $\mathbf{p}(t)$ is travelling slowly near $t = -2$ and quickly between $t = 2$ and $t = 3$. This type of "moving point" vocabulary is often used when discussing parametric curves even when the curve is not describing a moving point.

### 2.6.1 2D Parametric Lines

A parametric line in 2D that passes through points $\mathbf{p}_0 = (x_0, y_0)$ and $\mathbf{p}_1 = (x_1, y_1)$ can be written

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_0 + t(x_1 - x_0) \\ y_0 + t(y_1 - y_0) \end{bmatrix}.$$

Because the formulas for $x$ and $y$ have such similar structure, we can use the vector form for $\mathbf{p} = (x, y)$ (Figure 2.33):

$$\mathbf{p}(t) = \mathbf{p}_0 + t(\mathbf{p}_1 - \mathbf{p}_0).$$

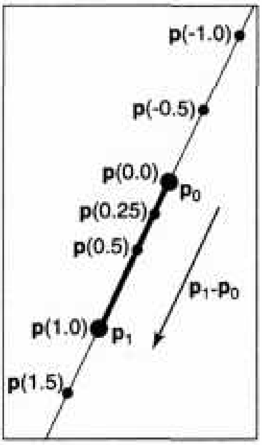


**Figure 2.33.** A 2D parametric line through $\mathbf{p}_0$ and $\mathbf{p}_1$. The line segment defined by t ∈ [0,1] is shown in bold.

You can read this in geometric form as: "start at point $\mathbf{p}_0$ and go some distance toward $\mathbf{p}_1$ determined by the parameter $t$." A nice feature of this form is that $\mathbf{p}(0) = \mathbf{p}_0$ and $\mathbf{p}(1) = \mathbf{p}_1$. Since the point changes linearly with $t$, the value of $t$ between $\mathbf{p}_0$ and $\mathbf{p}_1$ measures the fractional distance between the points. Points with $t < 0$ are to the "far" side of $\mathbf{p}_0$, and points with $t > 1$ are to the "far" side of $\mathbf{p}_1$.

Parametric lines can also be described as just a point $\mathbf{o}$ and a vector $\mathbf{d}$:

$$\mathbf{p}(t) = \mathbf{o} + t(\mathbf{d}).$$

When the vector $\mathbf{d}$ has unit length, the line is *arc-length parameterized.* This means $t$ is an exact measure of distance along the line. Any parametric curve can be arc-length parameterized, which is obviously a very convenient form, but not all can be converted analytically.

### 2.6.2 2D Parametric Circles

A circle with center $(x_c, y_c)$ and radius $r$ has a parametric form:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_c + r\cos\phi \\ y_c + r\sin\phi \end{bmatrix}.$$

To ensure that there is a unique parameter $\phi$ for every point on the curve, we can restrict its domain: $\phi \in [0, 2\pi)$ or $\phi \in (-\pi, \pi]$ or any other half open interval of length $2\pi$.

An axis-aligned ellipse can be constructed by scaling the $x$ and $y$ parametric equations separately:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_c + a\cos\phi \\ y_c + b\sin\phi \end{bmatrix}.$$

## 2.7 3D Implicit Surfaces

Implicit equations *implicitly* define a set of points that are on the surface

$$f(x, y, z) = 0.$$

Any point $(x, y, z)$ that is on the surface returns zero when given as an argument to $f$. Any point not on the surface returns some number other than zero. This is called implicit rather than explicit because you can check whether a point is on the surface by evaluating $f$, but you cannot always explicitly construct a set of points on the surface. As a convenient shorthand, I will write such functions of $\mathbf{p} = (x, y, z)$ as

$$f(\mathbf{p}) = 0.$$

### 2.7.1 Surface Normal to an Implicit Surface

A surface normal, which is needed for lighting computations, is a vector perpendicular to the surface. Each point on the surface may have a different normal vector. The surface normal at the intersection point $\mathbf{p}$ is given by the gradient of the implicit function

$$\mathbf{n} = \nabla f(\mathbf{p}) = \left( \frac{\partial f(\mathbf{p})}{\partial x}, \frac{\partial f(\mathbf{p})}{\partial y}, \frac{\partial f(\mathbf{p})}{\partial z} \right).$$

The gradient vector may point "into" the surface or may point "out" from the surface. If the particular form of $f$ creates inward facing gradients and outward facing gradients are desired, the surface $-f(\mathbf{p}) = 0$ is the same as surface $f(\mathbf{p}) = 0$ but has directionally reversed gradients, i.e., $\nabla f(\mathbf{p}) = -\nabla(-f(\mathbf{p}))$.

### 2.7.2 Implicit Planes

As an example, consider the infinite plane through point $\mathbf{a}$ with surface normal $\mathbf{n}$. The implicit equation to describe this plane is given by

$$(\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0. \tag{2.22}$$

Note that $\mathbf{a}$ and $\mathbf{n}$ are known quantities. The point $\mathbf{p}$ is any unknown point that satisfies the equation. In geometric terms this equation says "the vector from $\mathbf{a}$ to $\mathbf{p}$ is perpendicular to the plane normal." If $\mathbf{p}$ were not in the plane, then $(\mathbf{p} - \mathbf{a})$ would not make a right angle with $\mathbf{n}$ (Figure 2.34).

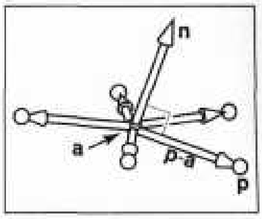


**Figure 2.34.** Any of the points **p** shown are in the plane with normal vector **n** that includes point **a** if Equation 2.22 is satisfied.

Sometimes we want the implicit equation for a plane through points $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$. The normal to this plane can be found by taking the cross product of any two vectors in the plane. One such cross product is

$$\mathbf{n} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a}).$$

This allows us to write the implicit plane equation:

$$(\mathbf{p} - \mathbf{a}) \cdot ((\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})) = 0.$$

A geometric way to read this equation is that the volume of the parallelepiped defined by $\mathbf{p} - \mathbf{a}$, $\mathbf{b} - \mathbf{a}$, and $\mathbf{c} - \mathbf{a}$ is zero, i.e., they are coplanar. This can only be true if $\mathbf{p}$ is in the same plane as $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$. The full-blown Cartesian representation for this is given by the determinant (this is discussed in more detail in Section 5.2.3):

$$\begin{vmatrix} x - x_a & y - y_a & z - z_a \\ x_b - x_a & y_b - y_a & z_b - z_a \\ x_c - x_a & y_c - y_a & z_c - z_a \end{vmatrix} = 0. \tag{2.23}$$

The determinant can be expanded (see Section 5.2.3 for the mechanics of expanding determinants) to the bloated form with many terms.

Equations 2.22 and 2.23 are equivalent, and comparing them is instructive. Equation 2.22 is easy to interpret geometrically and will yield efficient code. In addition, it is relatively easy to avoid a typographic error that compiles into incorrect code if it takes advantage of debugged cross and dot product code. Equation 2.23 is also easy to interpret geometrically and will be efficient provided an efficient 3 by 3 determinant function is implemented. It is also easy to implement without a typo provided a call of the type *determinant*$(\mathbf{a}, \mathbf{b}, \mathbf{c})$ exists. It will be especially easy for others to read your code if you rename the *determinant* function *volume*. So both Equations 2.22 and 2.23 map well into code. The full expansion of either equation is likely to generate typos. Such typos are likely to compile, and thus be especially pesky. This is an excellent example of clean math generating clean code, and bloated math generating bloated code.

### 2.7.3 3D Curves from Implicit Surfaces

One might hope that an implicit 3D curve could be created with the form $f(\mathbf{p}) = 0$. However, all such curves are just degenerate surfaces and are rarely useful in practice. 3D curves can be constructed from the intersection of two simultaneous implicit equations:

$$f(\mathbf{p}) = 0,$$
$$g(\mathbf{p}) = 0.$$

For example, a 3D line can be formed from the intersection of two implicit planes. Typically, it is more convenient to use 3D parametric curves, which are straightforward extensions of 2D parametric curves.

## 2.8 3D Parametric Curves

A 3D parametric curve operates much like a 2D parametric curve:

$$\begin{aligned} x &= f(t), \\ y &= g(t), \\ z &= h(t). \end{aligned}$$

For example, a spiral around the $z$-axis is:

$$\begin{aligned} x &= \cos t, \\ y &= \sin t, \\ z &= t. \end{aligned}$$

In this chapter we only discuss 3D parametric lines in detail. General 3D parametric curves are discussed more extensively in Chapter 15.

### 2.8.1 3D Parametric Lines

A 3D parametric line can be written as a straightforward extension of the 2D parametric line, e.g.,

$$\begin{aligned} x &= 2 + 7t, \\ y &= 1 + 2t, \\ z &= 3 - 5t. \end{aligned}$$

This is cumbersome and does not translate well to code variables, so we will write it in vector form:

$$\mathbf{p} = \mathbf{o} + t\mathbf{d},$$

where, for this example, $\mathbf{o}$ and $\mathbf{d}$ are given by

$$\begin{aligned}\mathbf{o} &= (2, 1, \quad 3),\\ \mathbf{d} &= (7, 2, -5).\end{aligned}$$

Note that this is very similar to the 2D case. The way to visualize this is to imagine that the line passes though $\mathbf{o}$ and is parallel to $\mathbf{d}$. Given any value of $t$, you get some point $\mathbf{p}(t)$ on the line. For example, at $t = 2$, $p(t) = (2, 1, 3) + 2(7, 2, -5) = (16, 5, -7)$. This general concept is the same as for two dimensions (Figure 2.30).

As in 2D, a *line segment* can be described by a 3D parametric line and an interval $t \in [t_a, t_b]$. The line segment between two points $\mathbf{a}$ and $\mathbf{b}$ is given by $\mathbf{p}(t) = \mathbf{a} + t(\mathbf{b} - \mathbf{a})$ with $t \in [0, 1]$. Here $\mathbf{p}(0) = \mathbf{a}$, $\mathbf{p}(1) = \mathbf{b}$, and $\mathbf{p}(0.5) = (\mathbf{a} + \mathbf{b})/2$, the midpoint between $\mathbf{a}$ and $\mathbf{b}$.

A *ray*, or *half-line*, is a 3D parametric line with a half-open interval, usually $[0, \infty)$. From now on we will refer to all lines, line segments, and rays as "rays." This is sloppy, but corresponds to common usage, and makes the discussion simpler.

## 2.9 3D Parametric Surfaces

Another way to specify 3D surfaces (surfaces in 3D space) is with 2D *parameters*. These surfaces have the form:

$$\begin{aligned}x &= f(u, v),\\ y &= g(u, v),\\ z &= h(u, v).\end{aligned}$$

### 2.9.1 Parametric Spheres

A point on the surface of the earth is given by the two parameters, longitude and latitude. For example, if we put a polar coordinate system on a radius $r$ sphere with center at the origin (Figure 2.35), we get the parametric equations

$$\begin{aligned}x &= r\cos\phi\sin\theta,\\ y &= r\sin\phi\sin\theta, \qquad (2.24)\\ z &= r\cos\theta.\end{aligned}$$

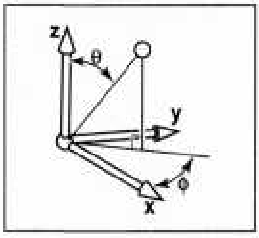


Figure 2.35. The geometry for spherical coordinates.

Ideally, we'd like to write this in vector form, but it isn't feasible for this particular parametric form. This use of $\theta$ and $\phi$ may or may not be backwards depending upon the reader's background. Unfortunately there is no standard for which angle uses which symbol across disciplines. Anyone who dismisses the importance of such standards should try to manipulate an equation of the form $\mathbf{bA} = \mathbf{x}$ where $\mathbf{b}$ is a square matrix and the other variables are column vectors, rather than the usual $\mathbf{Ax} = \mathbf{b}$. The more familiar they are with linear algebra, the worse their confusion will be. In graphics, we will always assume the meaning of $\theta$ and $\phi$ given in Equation 2.24. We will return to this equation when we texture map a sphere.

We would also like to be able to find the $(\theta, \phi)$ for a given $(x, y, z)$. If we assume that $\phi \in (-\pi, \pi]$ this is easy to do using the *atan2* function from Equation 2.2:

$$\begin{aligned} \theta &= \operatorname{acos}(z/\sqrt{x^2 + y^2 + z^2}), \\ \phi &= \operatorname{atan2}(y, x). \end{aligned} \tag{2.25}$$

## 2.10 Linear Interpolation

Perhaps the most common mathematical operation in graphics is *linear interpolation* . We have already seen an example of linear interpolation of position to form line segments in 2D and 3D, where two points $\mathbf{a}$ and $\mathbf{b}$ are associated with a parameter $t$ to form the line $\mathbf{p} = (1 - t)\mathbf{a} + t\mathbf{b}$. This is *interpolation* because $\mathbf{p}$ goes through $\mathbf{a}$ and $\mathbf{b}$ exactly at $t = 0$ and $t = 1$. It is *linear* interpolation because the weighting terms $t$ and $1 - t$ are linear polynomials of $t$.

Another common linear interpolation is among a set of positions on the $x$-axis: $x_0, x_1, \ldots, x_n$, and for each $x_i$ we have an associated height, $y_i$. We want to create a continuous function $y = f(x)$ that interpolates these positions, so that $f$ goes through every data point, i.e., $f(x_i) = y_i$. For linear interpolation, the points $(x_i, y_i)$ are connected by straight line segments. It is natural to use parametric line equations for these segments. The parameter $t$ is just the fractional distance between $x_i$ and $x_{i+1}$:

$$f(x) = y_i + \frac{x - x_i}{x_{i+1} - x_i}(y_{i+1} - y_i). \tag{2.26}$$

Because the weighting functions are linear polynomials of $x$, this is linear interpolation.

The two examples above have the common form of linear interpolation. Create a variable $t$ that varies from 0 to 1 as we move from data item $A$ to data item $B$. Intermediate values are just the function $(1-t)A + tB$. Notice that Equation 2.26 has this form with

$$t = \frac{x - x_i}{x_{i+1} - x_i}.$$

## 2.11 Triangles

Triangles in both 2D and 3D are the fundamental modeling primitive in many graphics programs. Often information such as color is tagged onto triangle vertices, and this information is interpolated across the triangle. The coordinate system that makes such interpolation straightforward is called *barycentric coordinates*, and we will develop these from scratch. We will also discuss 2D triangles, which must be understood before we can draw their pictures on 2D screens.

### 2.11.1 2D Triangles

If we have a 2D triangle defined by 2D points $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$, we can first find its area:

$$\begin{aligned} \text{area} &= \frac{1}{2}\begin{vmatrix} x_b - x_a & x_c - x_a \\ y_b - y_a & y_c - y_a \end{vmatrix} \\ &= \frac{1}{2}\left(x_a y_b + x_b y_c + x_c y_a - x_a y_c - x_b y_a - x_c y_b\right). \end{aligned} \tag{2.27}$$

The derivation of this formula can be found in Section 5.2.3. This area will have a positive sign if the points $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ are in counterclockwise order, and a negative sign, otherwise.

Often in graphics, we wish to assign a property, such as color, at each triangle vertex and smoothly interpolate the value of that property across the triangle. There are a variety of ways to do this, but the simplest is to use *barycentric* coordinates. One way to think of barycentric coordinates is as a non-orthogonal coordinate system as was discussed briefly in Section 2.4.2. Such a coordinate system is shown in Figure 2.36, where the coordinate origin is $\mathbf{a}$ and the vectors from $\mathbf{a}$ to $\mathbf{b}$ and $\mathbf{c}$ are the basis vectors. With that origin and those basis vectors, any point $\mathbf{p}$ can be written:

$$\mathbf{p} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}). \tag{2.28}$$

Note that we can reorder the terms in Equation 2.28 to get

$$\mathbf{p} = (1 - \beta - \gamma)\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}.$$

Often people define a new variable $\alpha$ to improve the symmetry of the equations:

$$\alpha \equiv 1 - \beta - \gamma,$$

which yields the equation

$$\mathbf{p}(\alpha, \beta, \gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}, \tag{2.29}$$

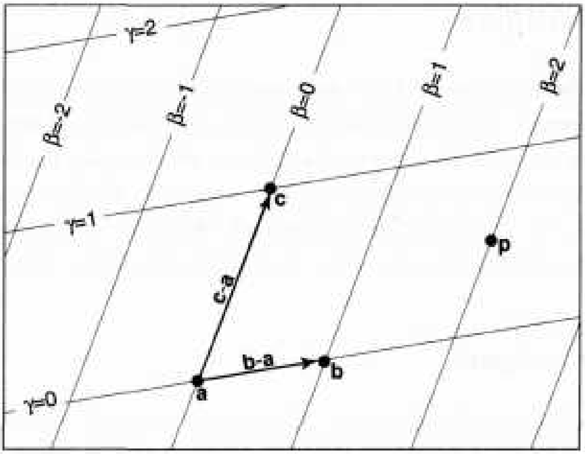


**Figure 2.36.** A 2D triangle with vertices **a**, **b**, **c** can be used to set up a non-orthogonal coordinate system with origin **a** and basis vectors (**b** - **a**) and (**c** - **a**). A point is then represented by an ordered pair $(\beta, \gamma)$. For example, the point **p** = (2.0, 0.5), i.e., **p** = **a** + 2.0 (**b**- **a**) + 0.5 (**c**- **a**).

with the constraint that

$$\alpha + \beta + \gamma = 1. \tag{2.30}$$

Barycentric coordinates seem like an abstract and unintuitive construct at first, but they turn out to be powerful and convenient. You may find it useful to think of how street addresses would work in a city where there were two sets of parallel streets, but where those sets were not at right angles. The natural system would essentially be barycentric coordinates, and you would quickly get used to them. Barycentric coordinates are defined for all points on the plane. A particularly nice feature of barycentric coordinates is that a point $\mathbf{p}$ is inside the triangle formed by $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ if and only if

$$\begin{aligned} 0 &< \alpha < 1, \\ 0 &< \beta < 1, \\ 0 &< \gamma < 1. \end{aligned}$$

If one of the coordinates is zero and the other two are between zero and one, then you are on an edge. If two are zero, then the other is one, and you are at a vertex. Another nice property of barycentric coordinates is that Equation 2.29 in effect mixes the the coordinates of the three vertices in a smooth way. The same mixing coefficients $(\alpha, \beta, \gamma)$ can be used to mix other properties, such as color, as we will see in the next chapter.

Given a point $\mathbf{p}$, how do we compute its barycentric coordinates? One way is to write Equation 2.28 as a linear system with unknowns $\beta$ and $\gamma$, solve, and set

$\alpha = 1 - \beta - \gamma$. That linear system is

$$\begin{bmatrix} x_b - x_a & x_c - x_a \\ y_b - y_a & y_c - y_a \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} x_p - x_a \\ y_p - y_a \end{bmatrix}. \tag{2.31}$$

Although it is straightforward to solve Equation 2.31 algebraically, it is often fruitful to compute a direct geometric solution.

One geometric property of barycentric coordinates is that they are the signed scaled distance from the lines through the triangle sides, as is shown for $\beta$ in Figure 2.37. Recall from Section 2.5.2 that evaluating the equation $f(x, y)$ for the line $f(x, y) = 0$ returns the scaled signed distance from $(x, y)$ to the line. Also recall that if $f(x, y) = 0$ is the equation for a particular line, so is $kf(x, y) = 0$ for any non-zero $k$. Changing $k$ scales the distance and controls which side of the line has positive signed distance, and which negative. We would like to choose $k$ such that, for example, $kf(x, y) = \beta$. Since $k$ is only one unknown, we can force this with one constraint, namely that at point $\mathbf{b}$ we know $\beta = 1$. So if the line $f_{ac}(x, y) = 0$ goes through both $\mathbf{a}$ and $\mathbf{c}$, then we can compute $\beta$ for a point $(x, y)$ as follows:

$$\beta = \frac{f_{ac}(x, y)}{f_{ac}(x_b, y_b)}, \tag{2.32}$$

and we can compute $\gamma$ and $\alpha$ in a similar fashion. For efficiency, it is usually wise to compute only two of the barycentric coordinates directly and to compute the third using Equation 2.30.

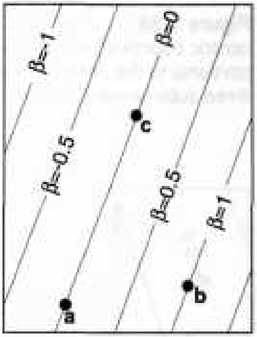


**Figure 2.37.** The barycentric coordinate $\beta$ is the signed scaled distance from the line through **a** and **c**.

To find this "ideal" form for the line through $\mathbf{p}_0$ and $\mathbf{p}_1$, we can first use the technique of Section 2.5.2 to find *some* valid implicit lines through the vertices. Equation 2.18 gives us

$$f_{ab}(x, y) \equiv (y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a = 0.$$

Note that $f_{ab}(x_c, y_c)$ probably does not equal one, so it is probably not the ideal form we seek. By dividing through by $f_{ab}(x_c, y_c)$ we get

$$\gamma = \frac{(y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a}{(y_a - y_b)x_c + (x_b - x_a)y_c + x_a y_b - x_b y_a}.$$

The presence of the division might worry us because it introduces the possibility of divide-by-zero, but this cannot occur for triangles with areas that are not near zero. There are analogous formulas for $\alpha$ and $\beta$, but typically only one is needed:

$$\beta = \frac{(y_a - y_c)x + (x_c - x_a)y + x_a y_c - x_c y_a}{(y_a - y_c)x_b + (x_c - x_a)y_b + x_a y_c - x_c y_a},$$
$$\alpha = 1 - \beta - \gamma.$$

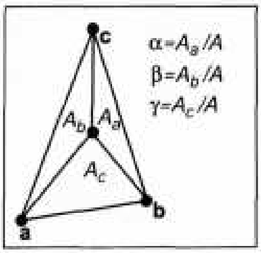


**Figure 2.38.** The barycentric coordinates are proportional to the areas of the three subtriangles shown.

Another way to compute barycentric coordinates is to compute the areas $A_a$, $A_b$, and $A_c$, of subtriangles as shown in Figure 2.38. Barycentric coordinates obey the rule

$$\begin{aligned} \alpha &= A_a/A, \\ \beta &= A_b/A, \\ \gamma &= A_c/A, \end{aligned} \tag{2.33}$$

where $A$ is the area of the triangle. Note that $A = A_a + A_b + A_c$, so it can be computed with two additions rather than a full area formula. This rule still holds for points outside the triangle if the areas are allowed to be signed. The reason for this is shown in Figure 2.39. Note that these are signed areas and will be computed correctly as long as the same signed area computation is used for both $A$ and the subtriangles $A_a$, $A_b$, and $A_c$.

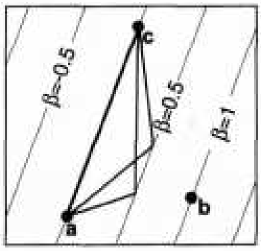


**Figure 2.39.** The area of the two triangles shown is base times height and are thus the same, as is any triangle with a vertex on the $\beta = 0.5$ line. The height and thus the area is proportional to $\beta$.

### 2.11.2 3D Triangles

One wonderful thing about barycentric coordinates is that they extend almost transparently to 3D. If we assume the points $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ are 3D, then we can still use the representation

$$\mathbf{p} = (1 - \beta - \gamma)\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}.$$

Now, as we vary $\beta$ and $\gamma$, we sweep out a plane.

The normal vector to a triangle can be found by taking the cross product of any two vectors in the plane of the triangle (Figure 2.40). It is easiest to use two of the three edges as these vectors, for example,

$$\mathbf{n} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a}). \tag{2.34}$$

Note that this normal vector is not necessarily of unit length, and it obeys the right-hand rule of cross products.

The area of the triangle can be found by taking the length of the cross product:

$$\text{area} = \frac{1}{2}\|(\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})\|. \tag{2.35}$$

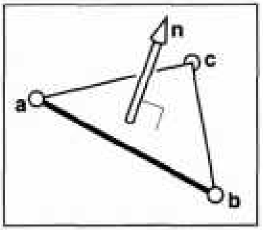


**Figure 2.40.** The normal vector of the triangle is perpendicular to all vectors in the plane of the triangle, and thus perpendicular to the edges of the triangle.

Note that this is *not* a signed area, so it cannot be used directly to evaluate barycentric coordinates. However, we can observe that a triangle with a "clockwise" vertex order will have a normal vector that points in the opposite direction to the normal of a triangle in the same plane with a "counterclockwise" vertex order. Recall that

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \; \|\mathbf{b}\| \; \cos\phi,$$

where $\phi$ is the angle between the vectors. If $\mathbf{a}$ and $\mathbf{b}$ are parallel, then $\cos\phi = \pm 1$, and this gives a test of whether the vectors point in the same or opposite directions. This, along with Equations 2.33, 2.34, and 2.35 suggest the formulas:

$$\alpha = \frac{\mathbf{n} \cdot \mathbf{n}_a}{\|\mathbf{n}\|^2},$$

$$\beta = \frac{\mathbf{n} \cdot \mathbf{n}_b}{\|\mathbf{n}\|^2},$$

$$\gamma = \frac{\mathbf{n} \cdot \mathbf{n}_c}{\|\mathbf{n}\|^2},$$

where $\mathbf{n}$ is Equation 2.34 evaluated with vertices $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$; $\mathbf{n}_a$ is Equation 2.34 evaluated with vertices $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{p}$, and so on, i.e.,

$$\begin{aligned} \mathbf{n}_a &= (\mathbf{c} - \mathbf{b}) \times (\mathbf{p} - \mathbf{b}), \\ \mathbf{n}_b &= (\mathbf{a} - \mathbf{c}) \times (\mathbf{p} - \mathbf{c}), \\ \mathbf{n}_c &= (\mathbf{b} - \mathbf{a}) \times (\mathbf{p} - \mathbf{a}). \end{aligned} \tag{2.36}$$

## Frequently Asked Questions

- Why isn't there vector division?

It turns out that there is no "nice" analogy of division for vectors. However, it is possible to motivate the quaternions by examining this questions in detail (see Hoffman's book referenced in the chapter notes).

- Is there something as clean as barycentric coordinates for polygons with more than three sides?

Unfortunately there is not. Even convex quadrilaterals are much more complicated. This is one reason triangles are such a common geometric primitive in graphics.

- Is there an implicit form for 3D lines?

No. However, the intersection of two 3D planes defines a 3D line, so a 3D line can be described by two simultaneous implicit 3D equations.

## Notes

The history of vector analysis is particularly interesting. It was largely invented by Grassman in the mid-1800s but was ignored and reinvented later (Crowe, 1994). Grassman now has a following in the graphics field of researchers who are developing *Geometric Algebra* based on some of his ideas (Doran & Lasenby, 2003). Readers interested in why the particular scaler and vector products are in some sense the right ones, and why we do not have a commonly-used vector division, will find enlightenment in the concise *About Vectors* (Hoffmann, 1975). Another important geometric tool is the *quaternion* invented by Hamilton in the mid-1800s. Quaternions are useful in many situations, but especially where orientations are concerned (Hanson, 2005).

## Exercises

1. The *cardinality* of a set is the number of elements it contains. Under IEEE floating point representation (Section 1.6), what is the cardinality of the *floats*?

2. Is it possible to implement a function that maps 32-bit integers to 64-bit integers that has a well defined inverse? Do all functions from 32-bit integers to 64-bit integers have well defined inverses?

3. Specify the unit cube ($x$, $y$, and $z$ coordinates all between 0 and 1 inclusive) in terms of the Cartesian product of three intervals.

4. If you have access to the natural log function $\ln(x)$, specify how you could use it to implement a $\log(b, x)$ function where $b$ is the base of the log. What should the function do for negative $b$ values? Assume an IEEE floating point implementation.

5. Solve the quadratic equation $4x^2 - 6x + 9 = 0$.

6. Implement a function that takes in coefficients $A$, $B$, and $C$ for the quadratic equation $Ax^2 + By + C = 0$ and computes the two solutions. Have the function return the number of valid (not NaN) solutions and fill in the return arguments so the smaller of the two solutions is first.

7. Show by counterexample that it is not always true that for 3D vectors $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$, $\mathbf{a} \times (\mathbf{b} \times \mathbf{c}) = (\mathbf{a} \times \mathbf{b}) \times \mathbf{c}$.

8. Given the non-parallel 3D vectors $\mathbf{a}$ and $\mathbf{b}$, compute a right-handed orthonormal basis such that $\mathbf{u}$ is parallel to $\mathbf{a}$ and $\mathbf{v}$ is in the the plane defined by $\mathbf{a}$ and $\mathbf{b}$.

9. What is the gradient of $f(x, y, z) = x^2 + y - 3z^3$?

10. What is a parametric form for the axis-aligned 2D ellipse?

11. What is the implicit equation of the plane through 3D points $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$? What is the parametric equation? What is the normal vector to this plane?

12. Given four 2D points $\mathbf{a}_0$, $\mathbf{a}_1$, $\mathbf{b}_0$, and $\mathbf{b}_1$, design a robust procedure to determine whether the line segments $\mathbf{a}_0\mathbf{a}_1$ and $\mathbf{b}_0\mathbf{b}_1$ intersect.

13. Design a robust procedure to compute the barycentric coordinates of a 2D point with respect to three 2D non-collinear points.

# 3

# Raster Algorithms

Most computer graphics images are presented to the user on a *raster* display. Such systems show images as rectangular arrays of *pixels*, which is short for "picture elements." These pixels are set using RGB (red-green-blue) color. In this chapter, we discuss the basics of raster displays, emphasizing the RGB color system and the non-linearities of standard image display.

## 3.1 Raster Displays

There are a variety of display technologies for desktop and projected display. These displays vary in *resolution* (the number of pixels) and physical size. Programmers can usually assume that the pixels are laid out in a rectangular array, also called a *raster*.

### 3.1.1 Pixels

Each displayable element in a raster display is called a *pixel*. Displays usually index pixels by an ordered pair $(i, j)$ indicating the row and column of the pixel. If a display has $n_x$ columns and $n_y$ rows of pixels, the bottom-left element is pixel $(0, 0)$ and the top-right is pixel $(n_x - 1, n_y - 1)$.[1]

[1] In many APIs the rows of an image will be addressed in the less intuitive manner from the top-to-bottom, so the top-left pixel has coordinates $(0, 0)$. This convention is common for historical reasons: it is the order that rows come in a standard television transmission.

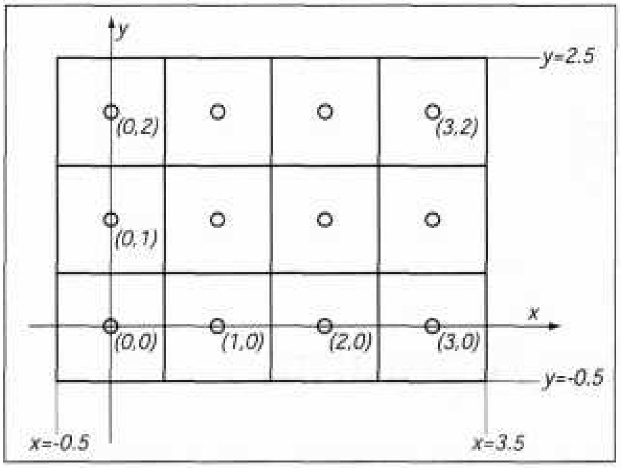


**Figure 3.1.** Coordinates of a four pixel by three pixel screen. Note that in some APIs the *y*-axis will point downwards.

We need 2D real screen coordinates to specify pixel positions. The details of such systems vary among APIs, but the most common is to use the integer lattice for pixel centers, as shown by the 4 by 3 screen in Figure 3.1. Because pixels have finite extent, note the 0.5 unit overshoot from the pixel centers.

Physical pixels, i.e., the actual displayed elements in hardware, will vary in shape from system to system. In CRTs (cathode ray tubes), the pixel is associated with a patch of phosphor in the CRT, and this phosphor glows based on how much an electron beam stimulates the phosphor. The shape of the pixel depends both on the details of how the electron beam sweeps the pixel, as well as the details of how the phosphor is distributed in the monitor. As a first approximation, we can assume the phosphor will have a "blobby" shape on the screen, with the highest intensity in the center and a gradual falloff toward the sides of the pixel. On an LCD (liquid crystal display) system, the pixels are approximately square filters which vary their opacity to darken a backlight. These pixels are almost perfect squares, and there is a small gap between squares to allow the control circuitry to get to the pixels. Most display systems other than CRTs or LCDs will behave somewhat like these two, with either blobby or square pixels.

## 3.2 Monitor Intensities and Gamma

All modern monitors take digital input for the "value" of a pixel and convert this to an intensity level. Real monitors have some non-zero intensity when they are off because the screen reflects some light. For our purposes we can consider this "black" and the monitor fully on as "white." We assume a numeric description of pixel color that ranges from zero to one. Black is zero, white is one, and a grey halfway between black and white is 0.5. Note that here "halfway" refers to the physical amount of light coming from the pixel, rather than the appearance. The human perception of intensity is non-linear and will not be part of the present discussion.

There are two key issues that must be understood to produce images on monitors. The first is that monitors are non-linear with respect to input. For example, if you give a monitor 0, 0.5, and 1.0 as inputs for three pixels, the intensities displayed might be 0, 0.25, and 1.0 (i.e., zero, one-quarter fully on, and fully on). As an approximate characterization of this non-linearity, most monitors are characterized by a $\gamma$ ("gamma") value. This value is the degree of freedom in the formula

$$\text{displayed intensity} = (\text{maximum intensity})a^{\gamma}, \tag{3.1}$$

where $a$ is the input intensity between zero and one. For example, if a monitor has a gamma of 2.0, and we input a value of $a = 0.5$, the displayed intensity will be one fourth the maximum possible intensity because $0.5^2 = 0.25$. Note that $a = 0$ maps to zero intensity and $a = 1$ maps to the maximum intensity regardless of the value of $\gamma$. Describing a display's non-linearity using $\gamma$ is just a first-order approximation; we do not need a great deal of accuracy in estimating the $\gamma$ of a device. A nice visual way to gauge the non-linearity is to find what value of $a$ gives an intensity halfway between black and white. This $a$ will be

$$0.5 = a^{\gamma}.$$

If we can find that $a$, we can deduce $\gamma$ by taking logs of both sides which yields

$$\gamma = \frac{\ln 0.5}{\ln a}.$$

We can find this $a$ by a standard technique where we display a checkerboard pattern of black and white pixels next to a square of grey pixels with input $a$ (Figure 3.2). When you look at this image from a distance (or without glasses if you are nearsighted), the two sides of the image will look about the same when $a$ is halfway between black and white. This is because the blurred checkerboard is mixing even numbers of white and black pixels so the overall effect is a uniform color halfway between white and black. To make this work, we must be able to try many values for $a$ until one matches. This can be done by giving the user a slider to control $a$, or by using many different gray squares simultaneously against a

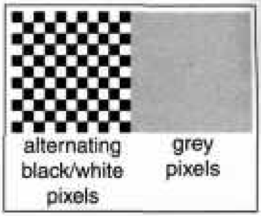


**Figure 3.2.** Alternating black and white pixels viewed from a distance are halfway between black and white. The gamma of a monitor can be inferred by finding a grey value that appears to have the same intensity as the black and white pattern.

large checkered region. Note that for CRTs, which have difficulty rapidly changing intensity along the horizontal direction, horizontal black and white stripes will work better than a checkerboard. This basic luminance matching strategy can be made more precise by taking advantage of human face recognition ability (Kindlmann, Reinhard, & Creem, 2002).

Once we know $\gamma$, we can *gamma correct* our input so that a value of $a = 0.5$ is displayed with intensity halfway between black and white. This is done with the transformation

$$a = a^{\frac{1}{\gamma}}.$$

When this formula is plugged into Equation 3.1 we get

$$\begin{aligned} \text{displayed intensity} &= \left(a^{\frac{1}{\gamma}}\right)^{\gamma} (\text{maximum intensity}) \\ &= a(\text{maximum intensity}). \end{aligned}$$

Another important characteristic of real displays is that they usually take quantized input values. So while we can manipulate intensities in the floating point range $[0, 1]$, the detailed input to a monitor is usually a fixed-size non-negative integer. The most common range for this integer is 0–255 which can be held in 8 bits of storage. This means that the possible values for $a$ are not any number in $[0, 1]$ but instead

$$\text{possible values for } a = \left\{ \frac{0}{255}, \frac{1}{255}, \frac{2}{255}, \ldots, \frac{254}{255}, \frac{255}{255} \right\}$$

This means the possible displayed intensity values are approximately

$$\left\{ M\left(\frac{0}{255}\right)^{\gamma}, M\left(\frac{1}{255}\right)^{\gamma}, M\left(\frac{2}{255}\right)^{\gamma}, \ldots, M\left(\frac{254}{255}\right)^{\gamma}, M\left(\frac{255}{255}\right)^{\gamma} \right\},$$

where $M$ is the maximum intensity. In applications where the exact intensities need to be controlled, we would have to actually measure the 256 possible intensities, and these intensities might be different at different points on the screen, especially for CRTs. They might also vary with viewing angle. Fortunately few applications require such accurate calibration.

## 3.3 RGB Color

Most computer graphics images are defined in terms of red-green-blue (RGB) color. RGB color is a simple space that allows straightforward conversion to the controls for most computer screens. In this section RGB color is discussed from a user's perspective, and operational facility is the goal. A more thorough discussion of color is given in Chapter 20, but the mechanics of RGB color space will allow us to write most graphics programs. The basic idea of RGB color space is that the color is displayed by mixing three *primary* lights: one red, one green, and one blue. The lights mix in an *additive* manner. Additive color mixing is fundamentally different from the more familiar *subtractive* color mixing that governs the mixing of paints and crayons. In those familiar media, red, yellow, and blue are the primaries, and they mix in familiar ways, such as yellow mixed with blue is green. In RGB additive color mixing we have (Figure 3.3):

$$\begin{aligned} \text{red} + \text{green} &= \text{yellow} \\ \text{green} + \text{blue} &= \text{cyan} \\ \text{blue} + \text{red} &= \text{magenta} \\ \text{red} + \text{green} + \text{blue} &= \text{white}. \end{aligned}$$

The color "cyan" is a blue-green, and the color "magenta" is a purple.

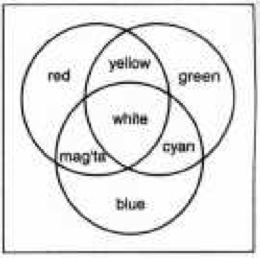


**Figure 3.3.** The additive mixing rules for colors red/green/blue.

If we are allowed to dim the primary lights from fully off to fully on, we can create all the colors that can be displayed on an RGB monitor. By convention, we write a color as the fraction of "fully on" it is for each monitor. This creates a three-dimensional *RGB color cube* that has a red, a green, and a blue axis. Allowable coordinates for the axes range from zero to one. The color cube is shown graphically in Figure 3.4.

The RGB coordinates of familiar colors are:

$$\begin{aligned} \text{black} &= (0, 0, 0) \\ \text{red} &= (1, 0, 0) \\ \text{green} &= (0, 1, 0) \\ \text{blue} &= (0, 0, 1) \\ \text{yellow} &= (1, 1, 0) \\ \text{magenta} &= (1, 0, 1) \\ \text{cyan} &= (0, 1, 1) \\ \text{white} &= (1, 1, 1). \end{aligned}$$

Actual RGB levels are often given in quantized form, just like the greyscales discussed in Section 3.2. Each component is specified with an integer. The most common size for these integers is one byte each, so each of the three RGB components is an integer between 0 and 255. The three integers together take up three bytes, which is 24 bits. Thus a system that has "24 bit color" has 256 possible levels for each of the three primary colors. Issues of gamma correction discussed in Section 3.2 also apply to each RGB component separately.

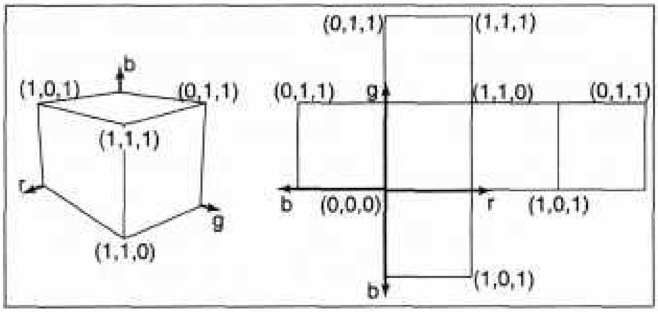


**Figure 3.4.** The RGB color cube in 3D and its faces unfolded. Any RGB color is a point in the cube. (See also Plate I.)

## 3.4 The Alpha Channel

Often we would like to only partially overwrite the contents of a pixel. A common example of this occurs in *compositing*, where we have a background and want to insert a foreground image over it. For opaque pixels in the foreground, we just replace the background pixel. For entirely transparent foreground pixels, we do not change the background pixel. For *partially* transparent pixels, some care must be taken. Partially transparent pixels can occur when the foreground object has partially transparent regions such as glass, or when there are sub-pixel holes in

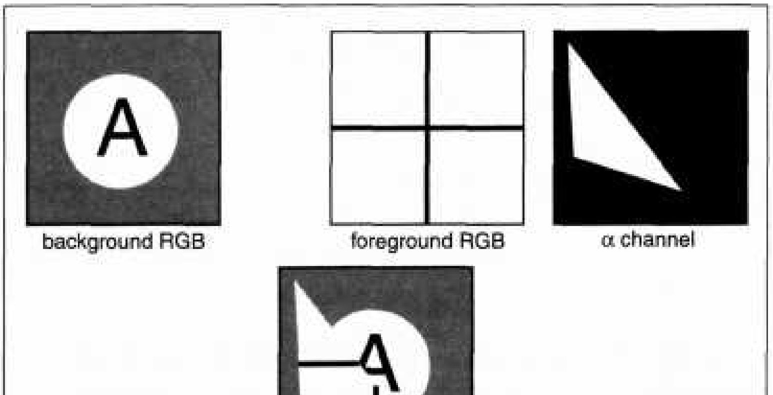


**Figure 3.5.** An example of compositing using Equation 3.2. The foreground image is in effect cropped by the $\alpha$ channel before being put on top of the background image. The resulting composite is shown on the bottom.

the foreground object such as in the leaves of a distant tree. To blend foreground and background in the case of holes, we want to measure the fraction of the pixel that should be foreground. We can call this fraction $\alpha$. If we want to composite a foreground color $\mathbf{c}_f$ over background color $\mathbf{c}_b$, and the fraction of the pixel covered by foreground is $\alpha$, then we can use the formula

$$\mathbf{c} = \alpha \mathbf{c}_f + (1 - \alpha)\mathbf{c}_b. \tag{3.2}$$

An example of using Equation 3.2 is shown in Figure 3.5. Note that the $\alpha$ image might be stored with the RGB image, or it might be stored as a separate greyscale (single channel) image.

Although Equation 3.2 is what is usually used, there are a variety of situations where $\alpha$ is used differently (Porter & Duff, 1984).

## 3.5 Line Drawing

Most graphics packages contain a line drawing command that takes two endpoints in screen coodinates (Figure 3.1) and draws a line between them. For example, the call for endpoints (1,1) and (3,2) would turn on pixels (1,1) and (3,2) and fill in one pixel between them. For general screen coordinate endpoints $(x_0, y_0)$ and $(x_1, y_1)$, the routine should draw some "reasonable" set of pixels that approximate a line between them. The values $x_0$, $x_1$, $y_0$, $y_1$ are often restricted to be integers (pixel centers) for simplicity, and because the lines themselves are coarse enough entities that subpixel accuracy is not appropriate. If you are implementing an API which calls for real number endpoint coordinates, rounding them to the nearest integer is usually a reasonable strategy that application programmers are unlikely to notice. Because the endpoint coordinates are integers, care should be taken to understand implicit conversions when these integers interact with floating point variables. Drawing such lines is based on line equations, and we have two types of equations to choose from: implicit and parametric. This section describes the two algorithms that result from these two types of equations.

### 3.5.1 Line Drawing Using Implicit Line Equations

The most common way to draw lines using implicit equations is the *midpoint* algorithm (Pitteway (1967); Van Aken and Novak (1985)). The midpoint algorithm ends up drawing the same lines as the *Bresenham algorithm* (Bresenham, 1965) but is somewhat more straightforward.

The first thing to do is find the implicit equation for the line as discussed in Section 2.5.2:

$$f(x, y) \equiv (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 = 0. \tag{3.3}$$

We assume that $x_0 \leq x_1$. If that is not true, we swap the points so that it is true. The slope $m$ of the line is given by

$$m = \frac{y_1 - y_0}{x_1 - x_0}.$$

The following discussion assumes $m \in (0, 1]$. Analogous discussions can be derived for $m \in (-\infty, -1]$, $m \in (-1, 0]$, and $m \in (1, \infty)$. The four cases cover all possibilities.

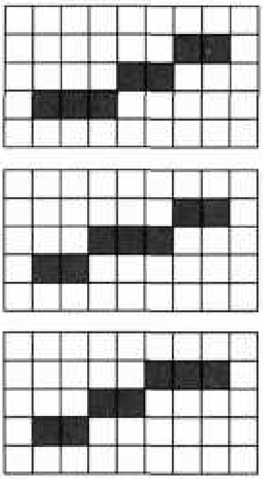


Figure 3.6. Three "reasonable" lines that go seven pixels horizontally and three pixels vertically.

For the case $m \in (0, 1]$, there is more "run" than "rise", i.e., the line is moving faster in $x$ than in $y$. If we have an API where the $y$-axis points downwards, we might have a concern about whether this makes the process harder, but, in fact, we can ignore that detail. We can ignore the geometric notions of "up" and "down," because the algebra is exactly the same for the two cases. Cautious readers can confirm that the resulting algorithm works for the $y$-axis downwards case. The key assumption of the midpoint algorithm is that we draw the thinnest line possible that has no gaps. A diagonal connection between two pixels is not considered a gap.

As the line progresses from the left endpoint to the right, there are only two possibilities: draw a pixel at the same height as the pixel drawn to its left, or draw a pixel one higher. There will always be exactly one pixel in each column of pixels between the endpoints. Zero would imply a gap, and two would be too thick a line. There may be two pixels in the same row for the case we are considering; the line is more horizontal than vertical so sometimes it will go right, and sometimes up. This concept is shown in Figure 3.6, where three "reasonable" lines are shown, each advancing more in the horizontal direction than in the vertical direction.

The midpoint algorithm for $m \in (0, 1]$ first establishes the leftmost pixel and the column number (x-value) of the rightmost pixel and then loops horizontally establishing the row (y-value) of each pixel. The basic form of the algorithm is:

```
$y = y_0$
for $x = x_0$ to $x_1$ do
    draw$(x, y)$
    if (some condition) then
        $y = y + 1$
```

Note that $x$ and $y$ are integers. In words this says, "keep drawing pixels from left to right and sometimes move upwards in the $y$-direction while doing so." The key is to establish efficient ways to make the decision in the *if* statement.

An effective way to make the choice is to look at the *midpoint* of the line between the two potential pixel centers. More specifically, the pixel just drawn is pixel $(x, y)$ whose center in real screen coordinates is at $(x, y)$. The candidate pixels to be drawn to the right are pixels $(x+1, y)$ and $(x+1, y+1)$.The midpoint between the centers of the two candidate pixels is $(x + 1, y + 0.5)$. If the line passes below this midpoint we draw the bottom pixel, and otherwise we draw the top pixel (Figure 3.7).

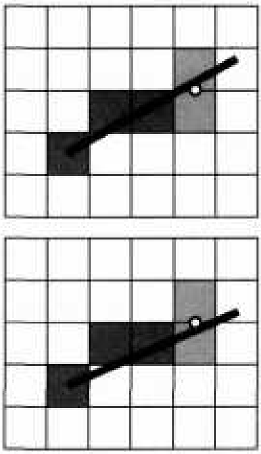


**Figure 3.7.** Top: the line goes above the midpoint so the top pixel is drawn. Bottom: the line goes below the midpoint so the bottom pixel is drawn.

To decide whether the line passes above or below $(x+1, y+0.5)$, we evaluate $f(x, y + 0.5)$ in Equation 3.3. Recall from Section 2.5 that $f(x, y) = 0$ for points $(x, y)$ on the line, $f(x, y) > 0$ for points on one side of the line, and $f(x, y) < 0$ for points on the other side of the line. Because $-f(x, y) = 0$ and $f(x, y) = 0$ are both perfectly good equations for the line, it is not immediately clear whether $f(x, y)$ being positive indicates that $(x, y)$ is above the line, or whether it is below. However, we can figure it out; the key term in Equation 3.3 is the $y$ term $(x_1 - x_0)y$. Note that $(x_1 - x_0)$ is definitely positive because $x_1 > x_0$. This means that as $y$ increases, the term $(x_1 - x_0)y$ gets larger (i.e., more positive or less negative). Thus, the case $f(x, +\infty)$ is definitely positive, and definitely above the line, implying points above the line are all positive. Another way to look at it is that the $y$ component of the gradient vector is positive. So above the line, where $y$ can increase arbitrarily, $f(x, y)$ must be positive. This means we can make our code more specific by filling in the *if* statement:

```
**if** $f(x + 1, y + 0.5) < 0$ **then**
    $y = y + 1$
```

The above code will work nicely for lines of the appropriate slope (i.e., between zero and one). The reader can work out the other three cases which differ only in small details.

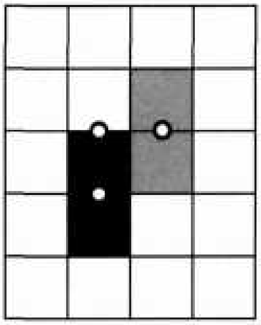


**Figure 3.8.** When using the decision point shown between the two light grey pixels, we just drew one of the dark grey pixels, so we evaluated *f* at one of the two left points shown.

If greater efficiency is desired, using an *incremental* method can help. An incremental method tries to make a loop more efficient by reusing computation from the previous step. In the midpoint algorithm as presented, the main computation is the evaluation of $f(x + 1, y + 0.5)$. Note that inside the loop, after the first iteration, either we already evaluated $f(x - 1, y + 0.5)$ or $f(x - 1, y - 0.5)$ (Figure 3.8). Note also this relationship:

$$f(x + 1, y) = f(x, y) + (y_0 - y_1)$$
$$f(x + 1, y + 1) = f(x, y) + (y_0 - y_1) + (x_1 - x_0).$$

This allows us to write an incremental version of the code:

```
$y = y_0$
$d = f(x_0 + 1, y_0 + 0.5)$
```

```
for $x = x_0$ to $x_1$ do
   draw$(x, y)$
   if $d < 0$ then
      $y = y + 1$
      $d = d + (x_1 - x_0) + (y_0 - y_1)$
   else
      $d = d + (y_0 - y_1)$
```

This code should run faster since it has little extra setup cost compared to the non-incremental version (that is not always true for incremental algorithms), but it may accumulate more numeric error because the evaluation of $f(x, y + 0.5)$ may be composed of many adds for long lines. However, given that lines are rarely longer than a few thousand pixels, such error is unlikely to be critical. Slightly longer setup cost, but faster loop execution, can be achieved by storing $(x_1 - x_0) + (y_0 - y_1)$ and $(y_0 - y_1)$ as variables. We might hope a good compiler would do that for us, but if the code is critical, it would be wise to examine the results of compilation to make sure.

In some cases, it is faster if an algorithm uses only integer operations. Because we have imposed the constraint that $x_0$, $x_1$, $y_0$, $y_1$ are all integers, the algorithm above is almost an integer-only algorithm. However, it does require the initialization $d = f(x_0 + 1, y_0 + 0.5)$. Note that this can be expanded as

$$f(x_0 + 1, y_0 + 0.5) = (y_0 - y_1)(x_0 + 1) + (x_1 - x_0)(y_0 + 0.5) + x_0 y_1 - x_1 y_0.$$

The $y_0 + 0.5$ is not an integer operation and results in a non-integer multiplier. But we can fix this: if $f(x, y) = 0$ is the equation of the line, then $2f(x, y) = 0$ is also a valid equation for the same line. So if we use $2f(x, y)$ instead of $f(x, y)$, the expression for $d$ becomes

$$\begin{aligned} 2f(x_0 + 1, y_0 + 0.5) = 2(y_0 - y_1)(x_0 + 1) + (x_1 - x_0)(2y_0 + 1) \\ + 2x_0 y_1 - 2x_1 y_0, \end{aligned}$$

which has all integer terms. The resulting code is:

```
$y = y_0$
$d = 2(y_0 - y_1)(x_0 + 1) + (x_1 - x_0)(2y_0 + 1) + 2x_0 y_1 - 2x_1 y_0$
for $x = x_0$ to $x_1$ do
   draw$(x, y)$
   if $d < 0$ then
      $y = y + 1$
      $d = d + 2(x_1 - x_0) + 2(y_0 - y_1)$
   else
      $d = d + 2(y_0 - y_1)$
```

The careful reader will note that before we can execute the all-integer code above, we must check whether $m \in [0, 1)$, and explicitly computing $m$ requires a divide. However, the following code, which is equivalent to the explicit slope check, can be used:

$$((y_1 \geq y_0) \text{ and } (x_1 - x_0 > y_1 - y_0)) \;\equiv\; (m \in [0, 1))$$

### 3.5.2 Line Drawing Using Parametric Line Equations

As derived in Section 2.6.1 a parametric line in 2D that goes through points $\mathbf{p}_0 = (x_0, y_0)$ and $\mathbf{p}_1 = (x_1, y_1)$ can be written

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_0 + t(x_1 - x_0) \\ y_0 + t(y_1 - y_0) \end{bmatrix},$$

or equivalently in vector form,

$$\mathbf{p}(t) = \mathbf{p}_0 + t(\mathbf{p}_1 - \mathbf{p}_0).$$

If the slope of the line $m \in [-1, 1]$, then the "for" loop can advance in $x$, and we get remarkably compact code. A similar algorithm results for $m$ outside $[-1, 1]$, so only two cases are needed. Because t progresses constantly along the distance of the line, so do the $x$ and $y$ components of the line; thus, we can compute $t$ as a function of $x$:

$$t = \frac{x - x_0}{x_1 - x_0}.$$

The resulting code is:

```
**for** $x = x_0$ to $x_1$ **do**
    $t = (x - x_0)/(x_1 - x_0)$
    $y = y_0 + t(y_1 - y_0)$
    draw($x$, round($y$))
```

A nice property of this algorithm is that it works whether or not $x_1 > x_0$. So the code for parametric lines turns out to be very simple. However, it cannot be made integer-only as we shall see. In many computer languages, conversion from float to integer is implemented as truncation, so the term round($y$) can be implemented as an integer conversion of $(y + 0.5)$.

This code can also be made incremental because $t$ and therefore $y$ change by a constant amount in each iteration:

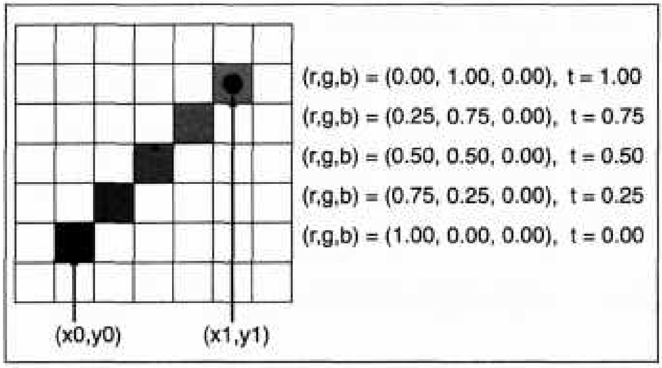

(r,g,b) = (0.00, 1.00, 0.00), t = 1.00
(r,g,b) = (0.25, 0.75, 0.00), t = 0.75
(r,g,b) = (0.50, 0.50, 0.00), t = 0.50
(r,g,b) = (0.75, 0.25, 0.00), t = 0.25
(r,g,b) = (1.00, 0.00, 0.00), t = 0.00
(x0,y0)
(x1,y1)

**Figure 3.9.** A colored line switching from red to green. The middle pixel is half red and half green which is a "dark yellow". (See also Plate II.)

```
$\Delta y = (y_1 - y_0)/(x_1 - x_0)$
$y = y_0$
**for** $x = x_0$ to $x_1$ **do**
   draw($x$,round($y$))
   $y = y + \Delta y$
```

Sometimes lines are specified with RGB colors $\mathbf{c}_0$ and $\mathbf{c}_1$ at either end, and we would like to change the color smoothly along the line. If we can parameterize the line segment in terms of a $t \in [0, 1]$, we can use the formula

$$\mathbf{c} = (1 - t)\mathbf{c}_0 + t\mathbf{c}_1.$$

This allows us to compute a color at each pixel. An example of such a colored line which shifts from red to green is shown in Figure 3.9. Note that the expression for $\mathbf{c}$ and for $t$ can also be computed incrementally if desired. The code for this is:

```
$\Delta y = (y_1 - y_0)/(x_1 - x_0)$
$\Delta r = (r_1 - r_0)/(x_1 - x_0)$
$\Delta g = (g_1 - g_0)/(x_1 - x_0)$
$\Delta b = (b_1 - b_0)/(x_1 - x_0)$
$y = y_0, r = r_0, g = g_0, b = b_0$
**for** $x = x_0$ to $x_1$ **do**
   draw($x$,round($y$), $r, g, b$)
   $y = y + \Delta y$
   $r = r + \Delta r$
   $g = g + \Delta g$
   $b = b + \Delta b$
```

A similar change can be made to the midpoint (implicit) algorithm, but it would be difficult to do using only integer operations. In infrastructures where floating point division is expensive, the four divides above can be replaced by one divide and four multiplies.

## 3.6 Triangle Rasterization

We often want to draw a 2D triangle with 2D points $\mathbf{p}_0 = (x_0, y_0)$, $\mathbf{p}_1 = (x_1, y_1)$, and $\mathbf{p}_2 = (x_2, y_2)$ in screen coordinates. This is similar to the line drawing problem, but it has some of its own subtleties. It will turn out that there is no advantage to integer coordinates for endpoints, so we will allow the $(x_i, y_i)$ to have floating point values. As with line drawing, we may wish to interpolate color or other properties from values at the vertices. This is straightforward if we have the barycentric coordinates (Section 2.11). For example, if the vertices have colors $\mathbf{c}_0$, $\mathbf{c}_1$, and $\mathbf{c}_2$, the color at a point in the triangle with barycentric coordinates $(\alpha, \beta, \gamma)$ is

$$\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2.$$

This type of interpolation of color is known in graphics as *Gouraud* interpolation after its inventor (Gouraud, 1971).

Another subtlety of rasterizing triangles is that we are usually rasterizing triangles that share vertices and edges. This means we would like to rasterize adjacent triangles so there are no holes. We could do this by using the midpoint algorithm to draw the outline of each triangle and then fill in the interior pixels. This would mean adjacent triangles both draw the same pixels along each edge. If the adjacent triangles have different colors, the image will depend on the order in which the two triangles are drawn. The most common way to rasterize triangles that avoids the order problem and eliminates holes is to use the convention that pixels are drawn if and only if their centers are inside the triangle, i.e., the barycentric coordinates of the pixel center are all in the interval $(0, 1)$. This raises the issue of what to do if the center is exactly on the edge of the triangle. There are several ways to handle this as will be discussed later in this section. The key observation is that barycentric coordinates allow us to decide whether to draw a pixel and what color that pixel should be if we are interpolating colors from the vertices. So our problem of rasterizing the triangle boils down to efficiently finding the barycentric coordinates of pixel centers (Pineda, 1988). The brute-force rasterization algorithm is:

```
for all $x$ do
  for all $y$ do
```

```
    compute $(\alpha, \beta, \gamma)$ for $(x, y)$
    if ($\alpha \in [0, 1]$ and $\beta \in [0, 1]$ and $\gamma \in [0, 1]$) then
      $\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2$
      drawpixel $(x, y)$ with color $\mathbf{c}$
```

The rest of the algorithm limits the outer loops to a smaller set of candidate pixels and makes the barycentric computation efficient.

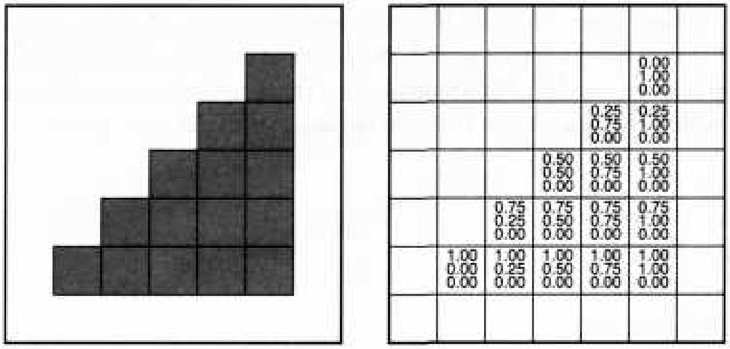
We can add a simple efficiency by finding the bounding rectangle of the three vertices and only looping over this rectangle for candidate pixels to draw. We can compute barycentric coordinates using Equation 2.32. This yields the algorithm:

```
$x_{\text{min}} = \text{floor}\ (x_i)$
$x_{\text{max}} = \text{ceiling}\ (x_i)$
$y_{\text{min}} = \text{floor}\ (y_i)$
$y_{\text{max}} = \text{ceiling}\ (y_i)$
for $y = y_{\text{min}}$ to $y_{\text{max}}$ do
  for $x = x_{\text{min}}$ to $x_{\text{max}}$ do
    $\alpha = f_{12}(x, y)/f_{12}(x_0, y_0)$
    $\beta = f_{20}(x, y)/f_{20}(x_1, y_1)$
    $\gamma = f_{01}(x, y)/f_{01}(x_2, y_2)$
    if ($\alpha > 0$ and $\beta > 0$ and $\gamma > 0$) then
      $\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2$
      drawpixel $(x, y)$ with color $\mathbf{c}$
```

Here $f_{ij}$ is the line given by Equation 3.3 with the appropriate vertices:

$$\begin{aligned} f_{01}(x, y) &= (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0, \\ f_{12}(x, y) &= (y_1 - y_2)x + (x_2 - x_1)y + x_1y_2 - x_2y_1, \\ f_{20}(x, y) &= (y_2 - y_0)x + (x_0 - x_2)y + x_2y_0 - x_0y_2. \end{aligned}$$

Note that we have exchanged the test $\alpha \in (0, 1)$ with $\alpha > 0$ etc., because if all of $\alpha$, $\beta$, $\gamma$ are positive, then we know they are all less than one because $\alpha + \beta + \gamma = 1$. We could also compute only two of the three barycentric variables and get the third from that relation, but it is not clear that this saves computation once the algorithm is made incremental, which is possible as in the line drawing algorithms; each of the computations of $\alpha$, $\beta$, and $\gamma$ does an evaluation of the form $f(x, y) = Ax + By + C$. In the inner loop, only $x$ changes, and it changes by one. Note that $f(x + 1, y) = f(x, y) + A$. This is the basis of the incremental algorithm. In the outer loop, the evaluation changes for $f(x, y)$ to $f(x, y + 1)$, so a similar efficiency can be achieved. Because $\alpha$, $\beta$, and $\gamma$ change by constant



| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | | |
| | | | | | 0.00<br>1.00<br>0.00 | |
| | | | | 0.25<br>0.75<br>0.00 | 0.25<br>1.00<br>0.00 | |
| | | | 0.50<br>0.50<br>0.00 | 0.50<br>0.75<br>0.00 | 0.50<br>1.00<br>0.00 | |
| | | 0.75<br>0.25<br>0.00 | 0.75<br>0.50<br>0.00 | 0.75<br>0.75<br>0.00 | 0.75<br>1.00<br>0.00 | |
| | 1.00<br>0.00<br>0.00 | 1.00<br>0.25<br>0.00 | 1.00<br>0.50<br>0.00 | 1.00<br>0.75<br>0.00 | 1.00<br>1.00<br>0.00 | |
| | | | | | | |

**Figure 3.10.** A colored triangle with barycentric interpolation. Note that the changes in color components are linear in each row and column as well as along each edge. In fact it is constant along every line, such as the diagonals, as well. (See also Plate III.)

increments in the loop, so does the color $\mathbf{c}$. So this can be made incremental as well. For example, the red value for pixel $(x + 1, y)$ differs from the red value for pixel $(x, y)$ by a constant amount that can be precomputed. An example of a triangle with color interpolation is shown in Figure 3.10.

### 3.6.1 Dealing With Pixels on Triangle Edges

We have still not discussed what to do for pixels whose centers are exactly on the edge of a triangle. If a pixel is exactly on the edge of a triangle, then it is also on the edge of the adjacent triangle if there is one. There is no obvious way to award the pixel to one triangle or the other. The worst decision would be to not draw the pixel because a hole would result between the two triangles. Better, but still not good, would be to have both triangles draw the pixel. If the triangles are transparent, this will result in a double-coloring. We would really like to award the pixel to exactly one of the triangles, and we would like this process to be simple; which triangle is chosen does not matter as long as the choice is well defined.

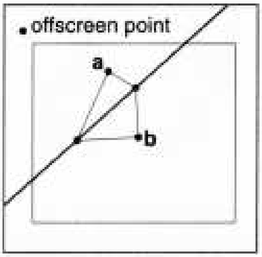


**Figure 3.11.** The off-screen point will be on one side of the triangle edge or the other. Exactly one of the non-shared vertices **a** and **b** will be on the same side.

One approach is to note that any off-screen point is definitely on exactly one side of the shared edge and that is the edge we will draw. For two non-overlapping triangles, the vertices not on the edge are on opposite sides of the edge from each other. Exactly one of these vertices will be on the same side of the edge as the off-screen point (Figure 3.11). This is the basis of the test. The test if numbers $p$ and $q$ have the same sign can be implemented as the test $pq > 0$, which is very efficient in most environments.

Note that the test is not perfect because the line through the edge may also go through the offscreen point, but we have at least greatly reduced the number of problematic cases. Which off-screen point is used is arbitrary, and $(x, y) = (-1, -1)$ is as good a choice as any. We will need to add a check for the case of a point exactly on an edge. We would like this check not to be reached for common cases, which are the completely inside or outside tests. This suggests:

```
$x_{\min} = \text{floor}\ (x_i)$
$x_{\max} = \text{ceiling}\ (x_i)$
$y_{\min} = \text{floor}\ (y_i)$
$y_{\max} = \text{ceiling}\ (y_i)$
$f_\alpha = f_{12}(x_0, y_0)$
$f_\beta = f_{20}(x_1, y_1)$
$f_\gamma = f_{01}(x_2, y_2)$
for $y = y_{\min}$ to $y_{\max}$ do
  for $x = x_{\min}$ to $x_{\max}$ do
    $\alpha = f_{12}(x, y)/f_\alpha$
    $\beta = f_{20}(x, y)/f_\beta$
    $\gamma = f_{01}(x, y)/f_\gamma$
    if ($\alpha \geq 0$ and $\beta \geq 0$ and $\gamma \geq 0$) then
      if ($\alpha > 0$ or $f_\alpha f_{12}(-1, -1) > 0$) and ($\beta > 0$ or $f_\beta f_{20}(-1, -1) > 0$)
      and ($\gamma > 0$ or $f_\gamma f_{01}(-1, -1) > 0$) then
        $\mathbf{c} = \alpha \mathbf{c}_0 + \beta \mathbf{c}_1 + \gamma \mathbf{c}_2$
        drawpixel $(x, y)$ with color $\mathbf{c}$
```

We might expect that the above code would work to eliminate holes and double-draws only if we use exactly the same line equation for both triangles. In fact, the line equation is the same only if the two shared vertices have the same order in the draw call for each triangle. Otherwise the equation might flip in sign. This could be a problem depending on whether the compiler changes the order of operations. So if a robust implementation is needed, the details of the compiler and arithmetic unit may need to be examined. The first four lines in the pseudocode above must be coded carefully to handle cases where the edge exactly hits the pixel center.

In addition to being amenable to an incremental implementation, there are several potential early exit points. For example, if $\alpha$ is negative, there is no need to compute $\beta$ or $\gamma$. While this may well result in a speed improvement, profiling is always a good idea; the extra branches could reduce pipelining or concurrency and might slow down the code. So as always, test any attractive looking optimizations if the code is a critical section.

Another detail of the above code is that the divisions could be divisions by zero for degenerate triangles, i.e., if $f_\gamma = 0$. Either the floating point error conditions should be accounted for properly, or another test will be needed.

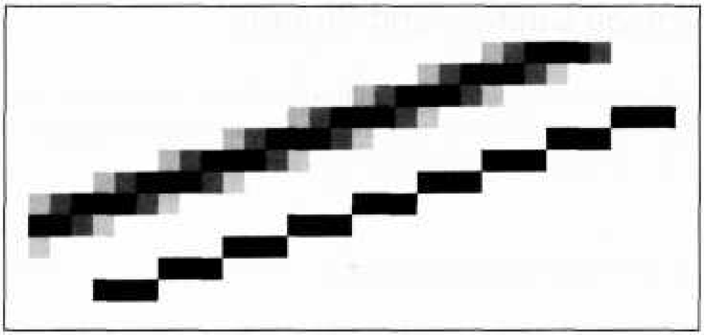


Figure 3.12. An antialiased and a jaggy line viewed at close range so individual pixels are visible.

## 3.7 Simple Antialiasing

One problem with the line and triangle drawing algorithms presented earlier is that they have fairly jaggy appearances. We can lessen this visual artifact by allowing a pixel to be "partially" on (Crow, 1978). For example, if the center of a pixel is *almost* inside a black triangle on a white background, we can color it halfway between white and black. The top line in the figure is drawn this way. In practice this form of blurring helps visual quality, especially in animations. This is shown as the bottom line of Figure 3.12.

The most straightforward way to create such "unjaggy" images is to use a *box filter*, where the pixel is set to the average color of the regions inside it. This means we have to think of all drawable entities as having well-defined areas. For example, a line is just a rectangle as shown in Figure 3.13. More sophisticated methods of blurring for visual quality are discussed in Chapter 4. Jaggy artifacts, like the sawtoothed lines the midpoint algorithm generates, are a result of *aliasing*, a term from signal processing. Thus, the general technique of carefully selecting pixel values to avoid jaggy artifacts is called *antialiasing*. The box filter will suffice for most applications that do not have extremely high visual quality requirements.

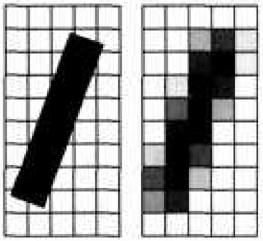


Figure 3.13. An antialiased line can be created by using an underlying rectangle.

The easiest way to implement box-filter antialiasing is to create images at very high resolutions and then downsample. For example, if our goal is a 256 by 256 pixel image of a line with width 1.2 pixels, we could rasterize a rectangle version of the line with width 4.8 pixels on a 1024 by 1024 screen, and then average 4 by 4 groups of pixels to get the colors for each of the 256 by 256 pixels in the "shrunken" image. This is an approximation of the actual box-filtered image, but works well when objects are not extremely small relative to the distance between pixels.

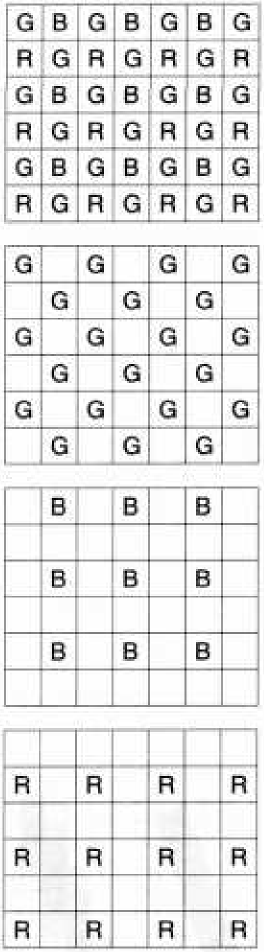
## 3.8 Image Capture and Storage

Almost all graphics software deals with some "real" images that are captured using digital cameras or flatbed scanners. This section deals with the practicalities of acquiring, storing, and manipulating such images.

### 3.8.1 Scanners and Digital Cameras

Scanners and digital cameras use some type of light-sensitive chip to record light. The dominant technologies are CCD and CMOS arrays. These devices are sensitive to light intensity across all wavelengths. To get color images, either the light is split into three components and then filtered through red, green, and blue filters (so three chips are needed), three passes are made with different filters and the same chip, or the sensors on the chip are individually coated with different colored filters.

In most current digital cameras, a single light-sensitive CCD is used with colored filters in the *Bayer mosaic* (Bayer, 1976) (Figure 3.14). This pattern devotes half of the sensors to the green channel and a quarter each to red and blue. The pattern makes the green channels a regular array at a forty-five degree angle to the chip lattice. For natural scenes this pattern works well in practice, but for some man-made scenes color aliasing can result. Camera manufacturers have different proprietary algorithms for creating a single RGB image from an image captured using the Bayer mosaic that are somewhat more complicated than the obvious strategy of linear interpolation in each separate channel.



| | | | | | | |
|---|---|---|---|---|---|---|
| G | B | G | B | G | B | G |
| R | G | R | G | R | G | R |
| G | B | G | B | G | B | G |
| R | G | R | G | R | G | R |
| G | B | G | B | G | B | G |
| R | G | R | G | R | G | R |

| | | | | | | |
|---|---|---|---|---|---|---|
| G | | G | | G | | G |
| | G | | G | | G | |
| G | | G | | G | | G |
| | G | | G | | G | |
| G | | G | | G | | G |
| | G | | G | | G | |

| | | | | | | |
|---|---|---|---|---|---|---|
| | B | | B | | B | |
| | | | | | | |
| | B | | B | | B | |
| | | | | | | |
| | B | | B | | B | |
| | | | | | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | | |
| R | | R | | R | | R |
| | | | | | | |
| R | | R | | R | | R |
| | | | | | | |
| R | | R | | R | | R |

**Figure 3.14.** The top image shows the mosaic of RGB sensors in a typical digital camera. The bottom three images show the same pattern with individual colors highlighted.

### 3.8.2 Image Storage

Most RGB image formats use eight bits for each of the red, green, and blue channels. This results in approximately three megabytes of raw information for a single million-pixel image. To reduce the storage requirement, most image formats allow for some kind of compression. At a high level, such compression is either *lossless* or *lossy*. No information is discarded in lossless compression, while some information is lost unrecoverably in a lossy system. Popular image storage formats include:

**gif** This lossy format indexes only 256 possible colors. The quality of the image depends on how carefully these 256 colors are chosen. This format typically works well for natural diagrams.

**jpeg** This lossy format compresses image blocks based on thresholds in the human visual system. This format works well for natural images.

**tiff** This lossless format is usually a compressed 24-bit per pixel although many other options exist.

**ppm** This lossless format is typically a 24-bit per pixel uncompressed format although many options exist.

**png** This is a set of lossless formats with a good set of open source management tools.

Because of compression and variants, writing input/output routines for images can be involved. For non-commercial applications it is advisable to just use raw ppm if no read/write libraries are readily available.

## Frequently Asked Questions

### • Why don't they just make monitors linear and avoid all this gamma business?

Ideally the 256 possible intensities of a monitor should *look* evenly spaced as opposed to being linearly spaced in energy. Because human perception of intensity is itself non-linear, a gamma between 1.5 and 3 (depending on viewing conditions) will make the intensities approximately uniform in a subjective sense. In this way gamma is a feature. Otherwise the manufacturers would make the monitors linear.

### • How are polygons that are not triangles rasterized?

These can either be done directly scan-line by scan-line, or they can be broken down into triangles. The latter appears to be the more popular technique.

### • Is it always better to antialias?

No. Some images look crisper without antialiasing. Many programs use unantialiased "screen fonts" because they are easier to read.

## Exercises

1. Derive the incremental form of the midpoint line drawing algorithm with colors at endpoints for $0 < m \leq 1$.

2. Modify the triangle drawing algorithm so that it will draw exactly one pixel for points on a triangle edge which goes through $(x, y) = (-1, -1)$.

3. Simulate an image acquired from the Bayer mosaic by taking a natural image (preferably a scanned photo rather than a digital photo where the Bayer mosaic may already have been applied) and creating a greyscale image composed of interleaved red/green/blue channels. This simulates the raw output of a digital camera. Now create a true RGB image from that output and compare with the original.

Stephen R. Marschner

# 4

# Signal Processing

In graphics, we often deal with functions of a continuous variable: an image is the first example you have seen, but you will encounter many more as you continue your exploration of graphics. By their nature continuous functions can't be directly represented in a computer; we have to somehow represent them using a finite number of bits. One of the most useful approaches to representing continuous functions is to use *samples* of the function: just store the values of the function at many different points and *reconstruct* the values in between when and if they are needed.

You are by now familiar with the idea of representing an image using a two-dimensional grid of pixels—so you have already seen a sampled representation! Think of an image captured by a digital camera: the actual image of the scene that was formed by the camera's lens is a continuous function of the position on the image plane, and the camera converted that function into a two-dimensional grid of samples. Mathematically, the camera converted a function of type $\mathbb{R}^2 \rightarrow \mathbf{C}$ (where $\mathbf{C}$ is the set of colors) to a two-dimensional array of color samples, or a function of type $\mathbb{Z}^2 \rightarrow \mathbf{C}$.

Another example of a sampled representation is a 2D digitizing tablet such as the screen of a tablet computer or PDA. In this case the original function is the motion of the stylus, which is a time-varying 2D position, or a function of type $\mathbb{R} \rightarrow \mathbb{R}^2$. The digitizer measures the position of the stylus at many points in time, resulting in a sequence of 2D coordinates, or a function of type $\mathbb{Z} \rightarrow \mathbb{R}^2$. A *motion capture* system does exactly the same thing for a special marker attached to an actor's body: it takes the 3D position of the marker over time ($\mathbb{R} \to \mathbb{R}^3$) and makes it into a series of instantaneous position measurements ($\mathbb{Z} \to \mathbb{R}^3$).

Going up in dimension, a medical CT scanner, used to non-invasively examine the interior of a person's body, measures density as a function of position inside the body. The output of the scanner is a 3D grid of density values: it converts the density of the body ($\mathbb{R}^3 \to \mathbb{R}$) to a 3D array of real numbers ($\mathbb{Z}^3 \to \mathbb{R}$).

These examples seem different, but in fact they can all be handled using exactly the same mathematics. In all cases a function is being sampled at the points of a *lattice* in one or more dimensions, and in all cases we need to be able to reconstruct that original continuous function from the array of samples.

From the example of a 2D image, it may seem that the pixels are enough, and we never need to think about continuous functions again once the camera has discretized the image. But what if we want to make the image larger or smaller on the screen, particularly by non-integer scale factors? It turns out that the simplest algorithms to do this perform badly, introducing obvious visual artifacts known as *aliasing*. Explaining why aliasing happens and understanding how to prevent it requires the mathematics of sampling theory. The resulting algorithms are rather simple, but the reasoning behind them, and the details of making them perform well, can be subtle.

Representing continuous functions in a computer is, of course, not unique to graphics; nor is the idea of sampling and reconstruction. Sampled representations are used in applications from digital audio to computational physics, and graphics is just one (and by no means the first) user of the related algorithms and mathematics. The fundamental facts about how to do sampling and reconstruction have been known in the field of communications since the 1920s and were stated in exactly the form we use them by the 1940s (Shannon & Weaver, 1964).

This chapter starts by summarizing sampling and reconstruction using the concrete one-dimensional example of digital audio. Then we go on to present the basic mathematics and algorithms that underlie sampling and reconstruction in one and two dimensions. Finally we go into the details of the frequency-domain viewpoint, which provides many insights into the behavior of these algorithms.

## 4.1 Digital Audio: Sampling in 1D

Although sampled representations had already been in use for years in telecommunications, the introduction of the compact disc in 1982, following the increased use of digital recording for audio in the previous decade, was the first highly visible consumer application of sampling.

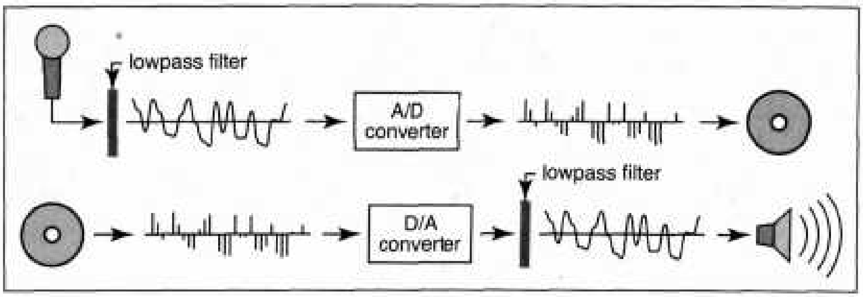


**Figure 4.1.** Sampling and reconstruction in digital audio.

In audio recording, a microphone converts sound, which exists as pressure waves in the air, into a time-varying voltage that amounts to a measurement of the changing air pressure at the point wherethe microphone is located. This electrical signal needs to be stored somehow so that it may be played back at a later time and sent to a loudspeaker that converts the voltage back into pressure waves by moving a diaphragm in synchronization with the voltage.

The digital approach to recording the audio signal (Figure 4.1) uses sampling: an *analog-to-digital converter* (*A/D converter*, or *ADC*) measures the voltage many thousand times per second, generating a stream of integers that can easily be stored on any number of media, say a disk on a computer in the recording studio, or transmitted to another location, say the memory in a portable audio player. At playback time, the data is read out at the appropriate rate and sent to a *digital-to-analog comverter* (*D/A canverter*, or *DAC*). The DAC produces a voltage according to the rumbers it receives, and, provided we take enough samples to fairly represent the variation in voltage, the resulting electrical signal is, for all practical purposes, idntical to the input.

It turns out that tle number of samples per second required to end up with a good reproduction dpends on how high-pitched the sounds are that we are trying to record. A samplerate that works fine for reproducing a string bass or a kick drum produces bizare-sounding results if we try to record a piccolo or a cymbal; but those sounds are eproduced just fine with a higher sample rate. To avoid these *undersampling artifcts* the digital audio recorder *filters* the input to the ADC to remove high frequecies that can cause problems.

Another kind o problem arises on the output side. The DAC produces a voltage that changes whenever a nev sample comes in, but stays constant until the next sample, pducing a stair-step shaped graph. These stair-steps act like noise, adding a hig-frequency, sigal-dependent buzzing sound. To remove this *reconstruction artiact*, the digital audio player filters the output from the DAC to smooth out the waveform.

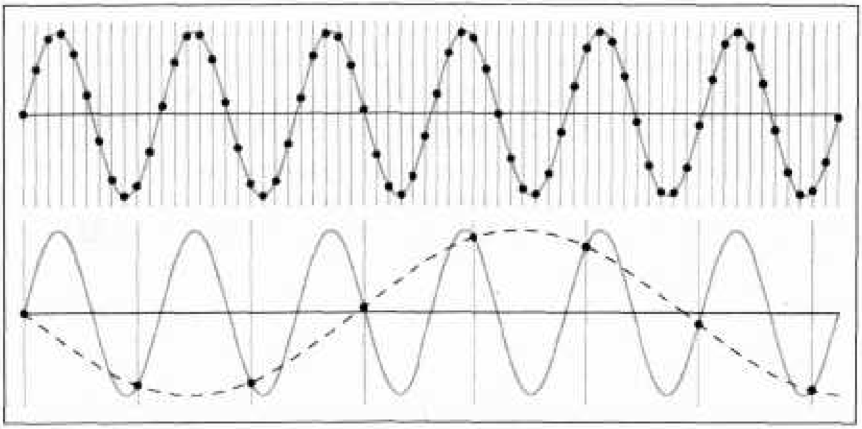


Figure 4.2. A sine wave (gray curve) sampled at two different rates. Top: at a high sample rate, the resulting samples (black dots) represent the signal well. Bottom: a lower sample rate produces an ambiguous result: the samples are exactly the same as would result from sampling a wave of much lower frequency (dashed curve).

### 4.1.1 Sampling Artifacts and Aliasing

The digital audio recording chain can serve as a concrete model for the sampling and reconstruction processes that happen in graphics. The same kind of undersampling and reconstruction artifacts also happen with images or other sampled signals in graphics, and the solution is the same: filtering before sampling and filtering again during reconstruction.

A concrete example of the kind of artifacts that can arise from too-low sample frequencies is shown in Figure 4.2. Here we are sampling a simple sine wave using two different sample frequencies: 10.8 samples per cycle on the top and 1.2 samples per cycle on the bottom. The higher rate produces a set of samples that obviously capture the signal well, but the samples resulting from the lower sample rate are indistinguishable from samples of a low-frequency sine wave—in fact, faced with this set of samples the low-frequency sinusoid seems the more likely interpretation.

Once the sampling has been done, it is impossible to know which of the two signals—the fast or the slow sine wave—was the original, and therefore there is no single method that can properly reconstruct the signal in both cases. Because the high frequency signal is "pretending to be" a low-frequency signal, this phenomenon is known as *aliasing*.

Aliasing shows up whenever flaws in sampling and recontruction lead to artifacts at surprising frequencies. In audio, aliasing takes the form of odd-sounding extra tones—a bell ringing at 10KHz, after being sampled at 8KHz, turns into a

6KHz tone. In images, aliasing often takes the form of *moiré patterns* that result from the interaction of the sample grid with regular features in an image, for instance the window blinds in Figure 4.34.

Another example of aliasing in a synthetic image is the familiar stair-stepping on straight lines that are rendered with only black and white pixels (Figure 4.34). This is an example of small-scale features (the sharp edges of the lines) creating artifacts at a different scale (for shallow-slope lines the stair steps are very long).

The basic issues of sampling and reconstruction can be understood simply based on features being too small or too large, but some more quantitative questions are harder to answer:

- What sample rate is high enough to ensure good results?
- What kinds of filters are appropriate for sampling and reconstruction?
- What degree of smoothing is required to avoid aliasing?

Solid answers to these questions will have to wait until we have developed the theory fully in Section 4.5

## 4.2 Convolution

Before we discuss algorithms for sampling and reconstruction, we'll first examine the mathematical concept on which they are based—*convolution*. Convolution is a simple mathematical concept that underlies the algorithms that are used for sampling, filtering, and reconstruction. It also is the basis of how we will analyze these algorithms later in the chapter.

Convolution is an operation on functions: it takes two functions and combines them to produce a new function. In this book, the convolution operator is denoted by a star: the result of applying convolution to the functions $f$ and $g$ is $f \star g$. We say that $f$ is convolved with $g$, and $f \star g$ is the convolution of $f$ and $g$.

Convolution can be applied either to continuous functions (functions $f(x)$ that are defined for any real argument $x$) or to discrete sequences (functions $a[i]$ that are defined only for integer arguments $i$). It can also be applied to functions defined on one-dimensional, two-dimensional, or higher-dimensional domains (that is, functions of one, two, or more arguments). We will start with the discrete, one-dimensional case first, then continue to continuous functions and two- and three-dimensional functions.

For convenience in the definitions, we generally assume that the functions' domains go on forever, though of course in practice they will have to stop somewhere, and we have to handle the end points in a special way.

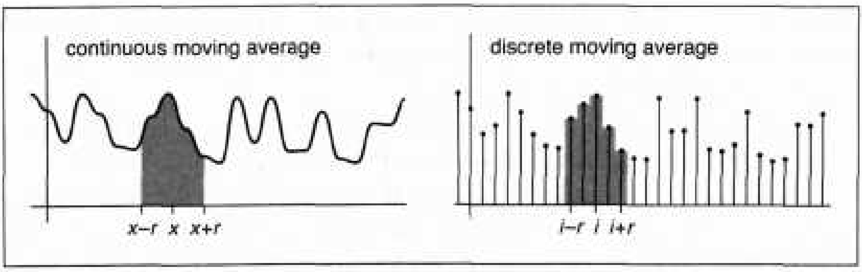


**Figure 4.3.** Smoothing using a moving average.

### 4.2.1 Moving Averages

To get a basic picture of convolution, consider the example of smoothing a 1D function using a moving average (Figure 4.3). To get a smoothed value at any point, we compute the average of the function over a range extending a distance $r$ in each direction. The distance $r$, called the *radius* of the smoothing operation, is a parameter that controls how much smoothing happens.

We can state this idea mathematically for discrete or continuous functions. If we're smoothing a continuous function $g(x)$, averaging means integrating $g$ over an interval and then dividing by the length of the interval:

$$h(x) = \frac{1}{2r} \int_{x-r}^{x+r} g(t)\, dt.$$

On the other hand, if we're smoothing a discrete function $b[i]$, averaging means summing $b$ for a range of indices and dividing by the number of values:

$$c[i] = \frac{1}{2r+1} \sum_{j=i-r}^{i+r} b[j]. \tag{4.1}$$

In each case, the normalization constant is chosen so that ifwe smooth a constant function the result will be the same function.

This idea of a moving average is the essence of convoltion; the only difference is that in convolution the moving average is a weightec average.

### 4.2.2 Discrete Convolution

We will start with the most concrete case ofconvolution: convolving a discrete sequence $a[i]$ with another discrete sequence $bi]$. The result isa discrete sequence $(a \star b)[i]$. The process is just like smoothing$b$ with a movig average, but this

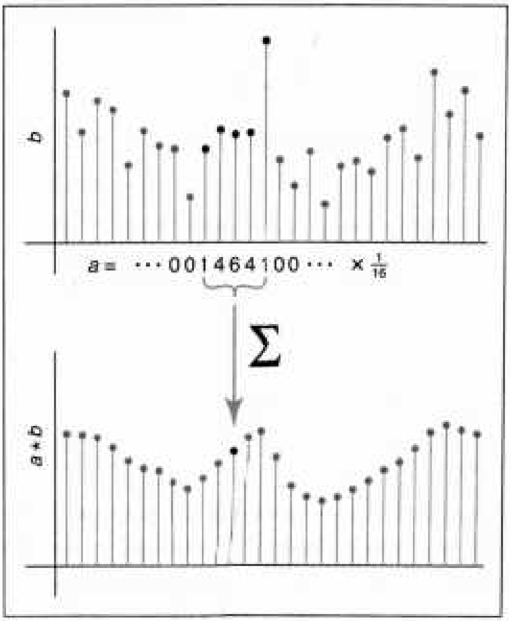


**Figure 4.4.** Computing one value in the dscrete convolution of a sequence *b* with a filter *a* that has support five samples wide. Eachsample in *a* ⋆ *b* is an average of nearby samples in *b*, weighted by the values of *a*.

time instead of equally weighting allsamples within a distance $r$, we use a second sequence $a$ to give a weight to eachsample (Figure 4.4). The value $a[j]$ gives the weight for a sample that is a distance $j$ from the index $i$ where we are evaluating the convolution. Hereis the definiton of $(a \star b)$, expressed as a formula:

$$(a \star b)[i] = \sum_j a[j]b[i-j]. \tag{4.2}$$

By omitting boundson $j$, we indicat: that this sum runs over all integers (that is, from $-\infty$ to $+\infty$). Figure 4.4 llustrates how one output sample is computed, using the exaaple of $a = \frac{1}{16}[\ldots, 0, 1, 4, 6, 4, 1, 0, \ldots]$—that is, $a[0] = \frac{6}{16}$, $a[\pm 1] = \frac{4}{16}$, etc.

In graphics, oneof the two funcions will usually have *finite support* (as does the example in Figre 4.4), whicb means that it is non-zero only over a finite interval of argumet values. If we ıssume that $a$ has finite support, there is some *radius* $r$ such that $[j] = 0$ whene'er $|j| > r$. In that case, we can write the sum above as

$$(a \star b)[i] = \sum_{j=-r}^{r} a[j]b[i-j],$$

and we can express the definition in code as

```
function convolve(sequence $a$, sequence $b$, int $r$, int $i$)
  $s = 0$
  for $j = -r$ to $r$
    $s = s + a[j]b[i-j]$
  return $s$
```

## Convolution Filters

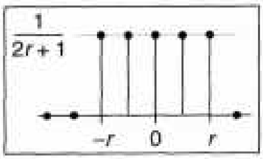


**Figure 4.5.** A discrete box filter.

Convolution is important because we can use it to perform filtering. Looking back at our first example of filtering, the moving average, we can now reinterpret that smoothing operation as convolution with a particular sequence. When we compute an average over some limited range of indices, that is the same as weighting the points in the range all identically and weighting the rest of the points with zeros. This kind of filter, which has a constant value over the interval where it is non-zero, is known as a *box filter* (because it looks like a rectangle if you draw its graph—see Figure 4.5). For a box filter of radius $r$ the weight is $1/(2r+1)$:

$$a[j] = \begin{cases} \frac{1}{2r+1} & -r \leq j \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

If you substitute this filter into Equation 4.2 you will find that it reduces to the moving average in Equation 4.1.

As in this example, convolution filters are usually desgned so that they sum to 1. That way, they don't affect the overall level of the signal.

**Example: Convolving a box and a step.** For a simple eample of filtering, let the signal be the *step function*

$$b[i] = \begin{cases} 1 & i \geq 0, \\ 0 & i < 0, \end{cases}$$

and the filter be the five-point box filter centered at zero,

$$a[j] = \frac{1}{5}\begin{cases} 1 & -2 \leq j \leq 2, \\ 0 & \text{othewise.} \end{cases}$$

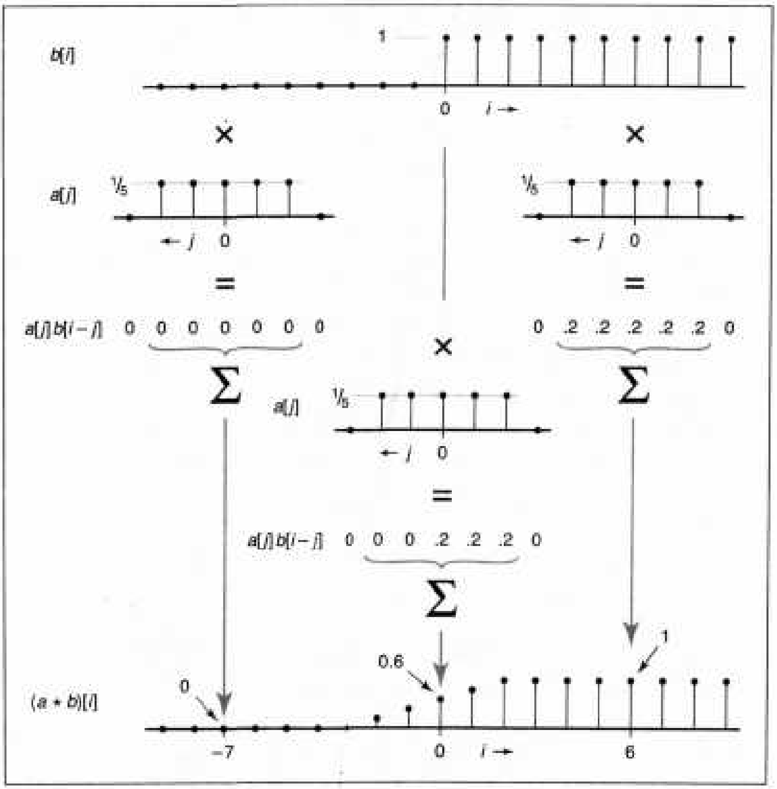


**Figure 4.6.** Discrete convolution of a box function with a step function.

What is the result of convolving $a$ and $b$? At a particular index $i$, as shown in Figure 4.6, the result is the average of the step function over the range from $i-2$ to $i+2$. If $i < -2$, we are averaging all zeros and the result is zero. If $i \geq 2$, we are averaging all ones and the result is one. In between there are $i+3$ ones, resulting in the value $\frac{i+3}{5}$. The output is a linear ramp that goes from 0 to 1 over five samples: $\frac{1}{5}[\ldots, 0, 0, 1, 2, 3, 4, 5, 5, \ldots]$.

### Properties of Convolution

The way we've written it so far, convolution seems like an asymmetric operation: $b$ is the sequence were smoothing, and $a$ provides the weights. But one of the nice properties of convolution is that it actually doesn't make any difference which is which: the filter and the signal are interchangeable. To see this, just rethink the sum in Equation 4.2 with the indices counting from the origin of the sequence $b$, rather than from the origin of $a$ where we are computing the value. That is, we replace $j$ with $i - k$. The result of this change of variable is

$$\begin{aligned}(a \star b)[i] &= \sum_k a[i-k]b[i-(i-k)] \\ &= \sum_k b[k]a[i-k].\end{aligned}$$

This is exactly the same as Equation 4.2 but with $b$ acting as the filter and $a$ acting as the signal. So for any sequences $a$ and $b$, $(a \star b) = (b \star a)$, and we say that convolution is a *commutative* operation.[1]

More generally, convolution is a "multiplication-like" operation. Like multiplication or addition of numbers or functions, neither the order of the arguments nor the placement of parentheses affects the result. Also, convolution relates to addition in the same way that multiplication does. To be precise, convolution is *commutative* and *associative*, and it is *distributive* over addition.

| | |
|---|---|
| commutative: | $(a \star b)[i] = (b \star a)[i]$ |
| associative: | $(a \star (b \star c))[i] = ((a \star b) \star c)[i]$ |
| distributive: | $(a \star (b + c))[i] = (a \star b + a \star c)[i]$ |

These properties are very natural if we think of convolution as being like multiplication, and they are very handy to know about because they can help us save work by simplifying convolutions before we actually compute them. For instance, suppose we want to take a sequence $b$ and convolve it with three filters, $a_1$, $a_2$, and $a_3$—that is, we want $a_3 \star (a_2 \star (a_1 \star b))$. If the sequence is long and the filters are short (that is, they have small radi), it is much faster to first convolve the three filters together (computing $a_1 \star a_2 \star a_3$) and finally to convolve the result with the signal, computing $(a_1 \star a_2 \star a_3) \star b$, which we know from commutativity and associativity gives the same result.

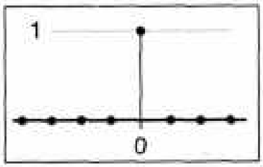


**Figure 4.7.** The discrete identity filter.

A very simple filter serves as an *identity* for discrete convolution: it is the discrete filter of radius zero, or the sequence $d[i] = \ldots, 0, 0, 1, 0, 0, \ldots$ (Figure 4.7). If we convolve $d$ with a signal $b$, there will be only one non-zero term in the sum:

$$\begin{aligned}(d \star b)[i] &= \sum_{j=0}^{j=0} d[j]b[i-j] \\ &= b[i].\end{aligned}$$

[1] You may have noticed that one of the functions in th convolution sum eems to be flipped over—that is, $a[j]$ gives the weight for the sample $j$ units *eafier* in the sequene, while $a[-j]$ gives the weight for the sample $j$ units *later* in the sequence. Te reason for thishas to do with ensuring associativity; see Exercise 4. Most of the filters we use ure symmetric, sc you hardly ever need to worry about this.

So clearly, convolving $b$ with $d$ just gives back $b$ again. The sequence $d$ is known as the *discrete impluse*. It is occasionally useful in expressing a filter: for instance, the process of smoothing a signal $b$ with a filter $a$ and then subtracting that from the original could be expressed as a single convolution with the filter $d - a$:

$$c = b - a \star b = d \star b - a \star b = (d - a) \star b.$$

### 4.2.3 Convolution as a Sum of Shifted Filters

There is a second, entirely equivalent, way of interpreting Equation 4.2. Looking at the samples of $a \star b$ one at a time leads to the weighted-average interpretation that we have already seen. But if we omit the $[i]$, we can instead think of the sum as adding together entire sequences. One piece of notation is required to make this work: if $b$ is a sequence, then the same sequence shifted to the right by $j$ places is called $b_{\rightarrow j}$ (Figure 4.8):

$$b_{\rightarrow j}[i] = b[i - j].$$

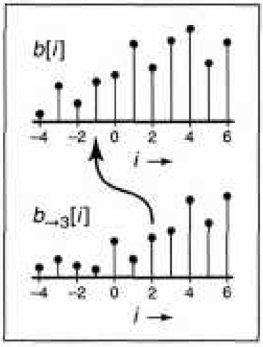


**Figure 4.8.** Shifting a sequence $b$ to get $b_{\rightarrow j}$.

Then, we can write Equation 4.2 as a statement about the whole sequence $(a \star b)$ rather than element-by-element:

$$(a \star b) = \sum_j a[j] b_{\rightarrow j}.$$

Looking at it this way, the convolution is a sum of shifted copies of $b$, weighted by the entries of $a$ (Figure 4.9). Because of commutativity, we can pick either $a$

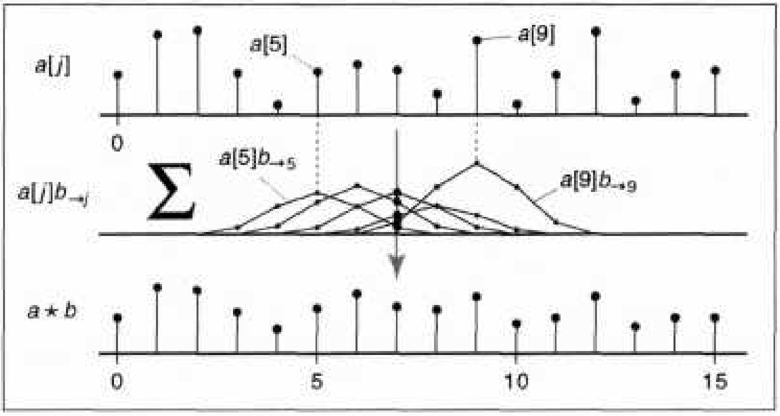


**Figure 4.9.** Discrete convolution as a sum of shifted copies of the filter.

or $b$ as the filter; if we choose $b$, then we are adding up one copy of the filter for every sample in the input.

### 4.2.4 Convolution with Continuous Functions

While it is true that discrete sequences are what we actually work with in a computer program, these sampled sequences are supposed to represent continuous functions, and often we need to reason mathematically about the continuous functions in order to figure out what to do. For this reason it is useful to define convolution between continuous functions and also between continuous and discrete functions.

The convolution of two continuous functions is the obvious generalization of Equation 4.2, with an integral replacing the sum:

$$(f \star g)(x) = \int_{-\infty}^{+\infty} f(t)g(x-t)\,dt. \tag{4.3}$$

One way of interpreting this definition is that the convolution of $f$ and $g$, evaluated at the argument $x$, is the area under the curve of the product of the two functions

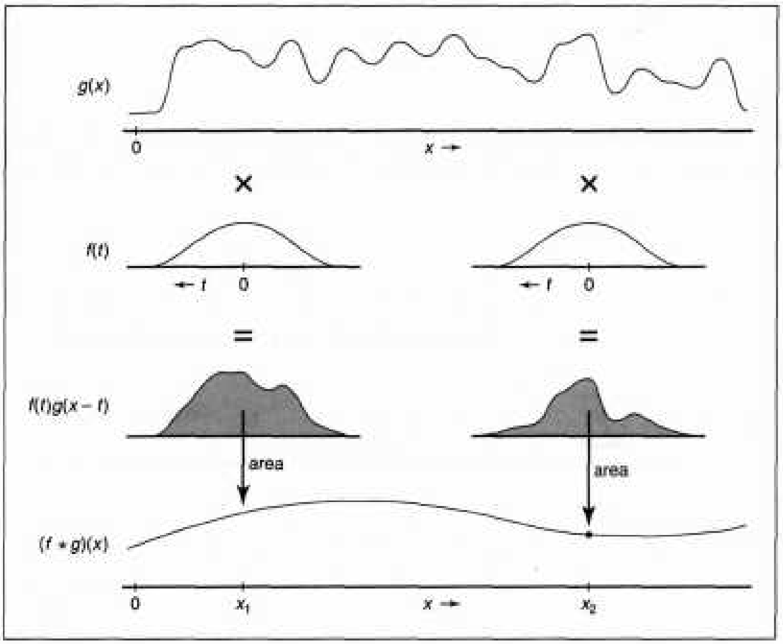


**Figure 4.10.** Continuous convolution.

after we shift $f$ so that $f(0)$ lines up with $g(x)$. Just like in the discrete case, the convolution is a moving average, with the filter providing the weights for the average (See Figure 4.10).

Like discrete convolution, convolution of continuous functions is commutative and associative, and it is distributive over addition. Also as with the discrete case, the continuous convolution can be seen as a sum of copies of the filter rather than the computation of weighted averages. Except, in this case, there are infinitely many copies of the filter:

$$(f \star g) = \int_{-\infty}^{+\infty} f(t) g_{\to t} \, dt.$$

**Example: Convolution of two box functions.** Let $f$ be a box function:

$$f(x) = \begin{cases} 1 & -\frac{1}{2} \leq x < \frac{1}{2}, \\ 0 & \text{otherwise.} \end{cases}$$

Then what is $f \star f$? The definition (Equation 4.3) gives

$$(f \star f)(x) = \int_{-\infty}^{\infty} f(t) f(x - t) \, dt.$$

Figure 4.11 shows the two cases of this integral. The two boxes might have zero overlap, which happens when $x \leq -1$ or $x \geq 1$; in this case the result is zero. When $-1 < x < 1$, the overlap depends on the separation between the two boxes,

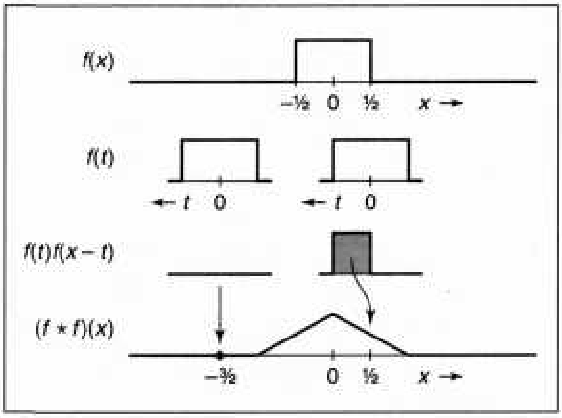


**Figure 4.11.** Convolving two boxes yields a tent function.

which is $|x|$; the result is $1 - |x|$. So

$$(f \star f)(x) = \begin{cases} 1 - |x| & -1 < x < 1, \\ 0 & \text{otherwise.} \end{cases}$$

This function, known as the *tent function*, is another common filter (see Section 4.3.1).

### The Dirac Delta Function

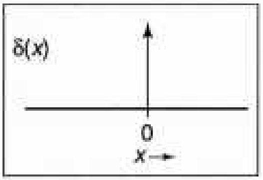


**Figure 4.12.** The Dirac delta function $\delta(x)$.

In discrete convolution, we saw that the discrete impulse $d$ acted as an identity: $d \star a = a$. In the continuous case, there is also an identity function, called the *Dirac impulse* or *Dirac delta* function, denoted $\delta(x)$.

Intuitively, the delta function is a very narrow, very tall spike that has infinitesimal width but still has area equal to 1 (Figure 4.12). The key defining property of the delta function is that multiplying it by a function selects out the value exactly at zero:

$$\int_{-\infty}^{\infty} \delta(x) f(x) dx = f(0).$$

The delta function does not have a well-defined value at 0 (you can think of its value loosely as $+\infty$), but it does have the value $\delta(x) = 0$ for all $x \neq 0$.

From this property of selecting out single values, it follows that the delta function is the identity for continuous convolution (Figure 4.13). The convolution of

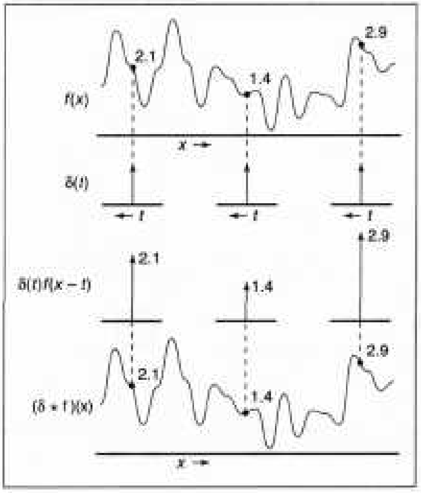


**Figure 4.13.** Convolving a function with $\delta(x)$ returns a copy of the same function.

$\delta$ with a function $f$ is

$$(\delta \star f)(x) = \int_{-\infty}^{\infty} \delta(t) f(x-t)dt = f(x).$$

So $\delta \star f = f$.

### 4.2.5 Discrete-Continuous Convolution

There are two ways to connect the discrete and continuous worlds. One is sampling: we convert a continuous function into a discrete one by writing down the function's value at all integer arguments and forgetting about the rest. Given a continuous function $f(x)$, we can sample it to convert to a discrete sequence $a[i]$:

$$a[i] = f(i).$$

Going the other way, from a discrete function, or sequence, to a continuous function, is called *reconstruction*. This is accomplished using yet another form of convolution, the discrete-continuous form. In this case, we are filtering a discrete sequence $a[i]$ with a continuous filter $f(x)$:

$$(a \star f)(x) = \sum_i a[i] f(x-i).$$

The value of the reconstructed function $a \star f$ at $x$ is a weighted sum of the samples $a[i]$ for values of $i$ near $x$ (Figure 4.14). The weights come from the filter $f$, which is evaluated at a set of points spaced one unit apart. For example, if $x = 5.3$ and

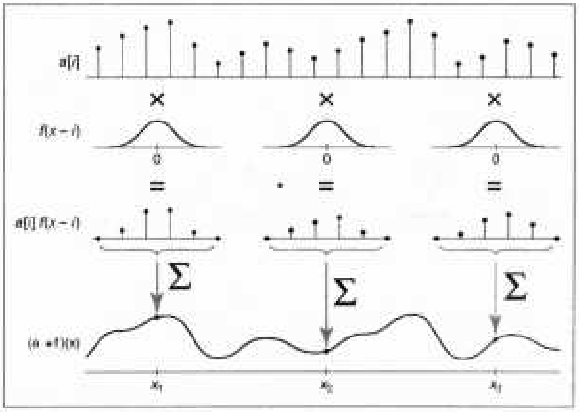


**Figure 4.14.** Discrete-continuous convolution.

$f$ has radius 2, $f$ is evaluated at 1.3, 0.3, $-0.7$, and $-1.7$. Note that for discrete-continuous convolution we generally write the sequence first and the filter second, so that the sum is over integers.

As with discrete convolution, we can put bounds on the sum if we know the filter's radius, $r$, eliminating all points where the difference between $x$ and $i$ is at least $r$:

$$(a \star f)(x) = \sum_{i=\lceil x-r \rceil}^{\lfloor x+r \rfloor} a[i]f(x-i).$$

Note, that if a point falls exactly at distance $r$ from $x$ (i.e., if $x - r$ turns out to be an integer), it will be left out of the sum. This is in contrast to the discrete case, where we included the point at $i - r$.

Expressed in code, this is:

```
function reconstruct(sequence $a$, filter $f$, real $x$)
$s = 0$
$r = f$.radius
for $i = \lceil x - r \rceil$ to $\lfloor x + r \rfloor$ do
    $s = s + a[i]f(x - i)$
return $s$
```

As with the other forms of convolution, discrete-continuous convolution may be seen as summing shifted copies of the filter (Figure 4.15):

$$(a \star f) = \sum_i a[i]f_{\rightarrow i}.$$

Discrete-continuous convolution is closely related to splines. For uniform splines (a uniform B-spline, for instance), the parameterized curve for the spline

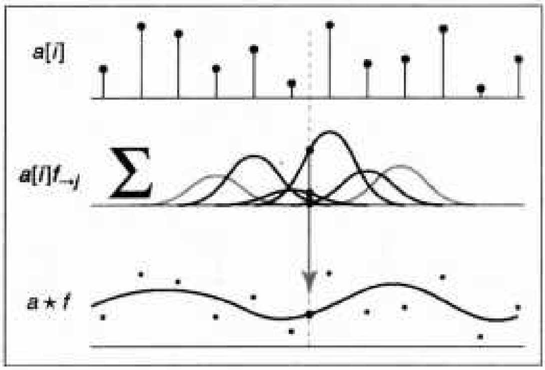


**Figure 4.15.** Reconstruction (discrete-continuous convolution) as a sum of shifted copies of the filter.

is exactly the convolution of the spline's basis function with the control point sequence (see Section 15.6.2).

### 4.2.6 Convolution in More than One Dimension

So far, everything we have said about sampling and reconstruction has been one-dimensional: there has been a single variable $x$ or a single sequence index $i$. Many of the important applications of sampling and reconstruction in graphics, though, are applied to two-dimensional functions—in particular, to 2D images. Fortunately, the generalization of sampling algorithms and theory from 1D to 2D, 3D, and beyond is conceptually very simple.

Beginning with the definition of discrete convolution, we can generalize it to two dimensions by making the sum into a double sum:

$$(a \star b)[i,j] = \sum_{i'} \sum_{j'} a[i',j']b[i-i',j-j'].$$

If $a$ is a finitely supported filter of radius $r$ (that is, it has $(2r+1)^2$ values), then we can write this sum with bounds (Figure 4.16):

$$(a \star b)[i,j] = \sum_{i'=-r}^{i'=r} \sum_{j'=-r}^{j'=r} a[i',j']b[i-i',j-j']$$

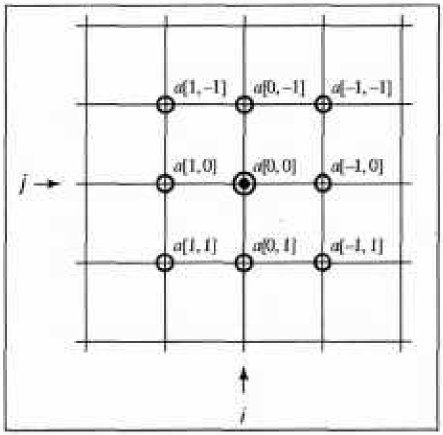


**Figure 4.16.** The weights for the nine input samples that contribute to the discrete convolution at point $(i, j)$ with a filter $a$ of radius 1.

and express it in code:

```
function convolve2d(filter2d $a$, filter2d $b$, int $i$, int $j$)
$s = 0$
$r = a$.radius
for $i' = -r$ to $r$ do
    for $j' = -r$ to $r$ do
        $s = s + a[i'][j']b[i - i'][j - j']$
return $s$
```

This definition can be interpreted in the same way as in the 1D case: each output sample is a weighted average of an area in the input, using the 2D filter as a "mask" to determine the weight of each sample in the average.

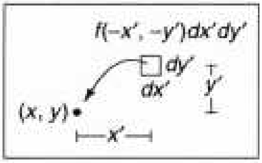


**Figure 4.17.** The weight for an infinitesimal area in the input signal resulting from continuous convolution at (*x*, *y*).

Continuing the generalization, we can write continuous-continuous (Figure 4.17) and discrete-continuous (Figure 4.18) convolutions in 2D as well:

$$(f \star g)(x, y) = \int\int f(x', y')g(x - x', y - y')\, dx'\, dy';$$

$$(a \star f)(x, y) = \sum_i \sum_j a[i, j]f(x - i, y - j).$$

In each case, the result at a particular point is a weighted average of the input near that point. For the continuous-continuous case, it is a weighted integral over a region centered at that point, and in the discrete-continuous case it is a weighted average of all the samples that fall near the point.

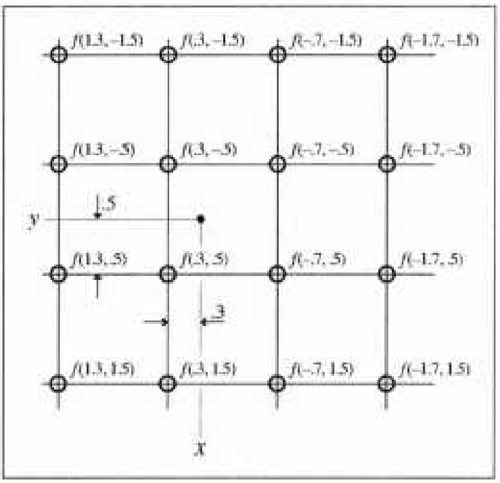


**Figure 4.18.** The weights for the 16 input samples that contribute to the discrete-continuous convolution at point (*x*, *y*) for a reconstruction filter of radius 2.

Once we have gone from 1D to 2D, it should be fairly clear how to generalize further to 3D or even to higher dimensions.

## 4.3 Convolution Filters

Now that we have the machinery of convolution, let's examine some of the particular filters commonly used in graphics.

### 4.3.1 A Gallery of Convolution Filters

#### The Box Filter

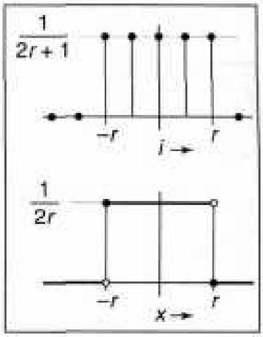


Figure 4.19. The discrete and continuous box filters.

The box filter (Figure 4.19) is a piecewise constant function whose integral is equal to one. As a discrete filter, it can be written as

$$a_{\text{box},r}[i] = \begin{cases} 1/(2r+1) & |i| \leq r, \\ 0 & \text{otherwise}. \end{cases}$$

Note that for symmetry we include both endpoints.

As a continuous filter, we write

$$f_{\text{box},r}(x) = \begin{cases} 1/(2r) & -r \leq x < r, \\ 0 & \text{otherwise}. \end{cases}$$

In this case, we exclude one endpoint which makes the box of radius 0.5 usable as a reconstruction filter. It is because the box filter is discontinuous that these boundary cases are important, and so for this particular filter, we need to pay attention to them. We write just $f_{\text{box}}$ for the common case of $r = \frac{1}{2}$.

#### The Tent Filter

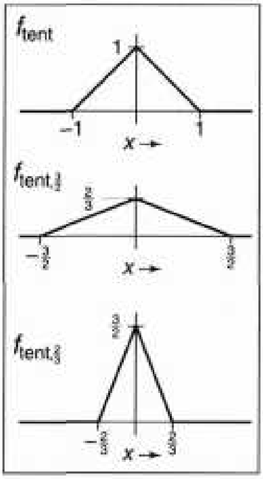


Figure 4.20. The tent filter and two scaled versions.

The tent, or linear filter (Figure 4.20) is a continuous, piecewise linear function:

$$f_{\text{tent}}(x) = \begin{cases} 1 - |x| & |x| < 1, \\ 0 & \text{otherwise}; \end{cases}$$

$$f_{\text{tent},r}(x) = \frac{f_{\text{tent}}(x/r)}{r}.$$

For filters that are at least $C^0$ (that is, there are no sudden jumps in the value, as there are with the box), we no longer need to separate the definitions of the discrete and continuous filters: the discrete filter is just the continuous filter sampled at the integers. Also note that for simplicity we define $f_{\text{tent},r}$ by scaling the "standard size" tent filter $f_{\text{tent}}$. From now on, we'll take this scaling for granted: once we define a filter $f$, then we can use $f_r$ to mean "the filter $f$ stretched out by $r$ and also scaled down by $r$." Note that $f_r$ has the same integral as $f$, and we will always make sure that the value of the integral is equal to 1.0.

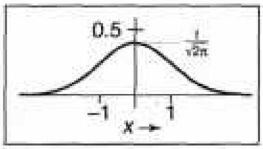


**Figure 4.21.** The Gaussian filter.

### The Gaussian Filter

The Gaussian function (Figure 4.21), also known as the normal distribution, is an important filter theoretically and practically. We'll see more of its special properties as the chapter goes on:

$$f_g(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}.$$

The Gaussian does not have finite support, although because of the exponential decay, its values rapidly become small enough to ignore. When necessary, then, we can trim the tails from the function by setting it to zero outside some radius. The Gaussian makes a good sampling filter because it is very smooth; we'll make this statement more precise later in the chapter.

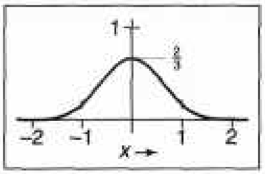


**Figure 4.22.** The B-spline filter.

### The B-Spline Cubic Filter

Many filters are defined as piecewise polynomials, and cubic filters with four pieces are often used as reconstruction filters. One such filter is known as the B-spline filter (Figure 4.22) because of its origins as a blending function for spline curves (see Chapter 15):

$$f_B(x) = \frac{1}{6}\begin{cases} -3(1-|x|)^3 + 3(1-|x|)^2 + 3(1-|x|) + 1 & -1 \le x \le 1, \\ (2-|x|)^3 & 1 \le |x| \le 2, \\ 0 & \text{otherwise.} \end{cases}$$

Among piecewise cubics, the B-spline is special because it has continuous first and second derivatives—that is, it is $C^2$. A more concise way of defining this filter is $F_B = f_{\text{box}} \star f_{\text{box}} \star f_{\text{box}} \star f_{\text{box}}$; proving that the longer form above is equivalent is a nice exercise in convolution (see Exercise 3).

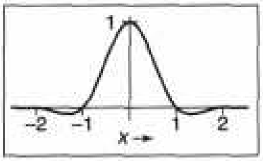


**Figure 4.23.** The Catmull-Rom filter.

### The Catmull-Rom Cubic Filter

Another piecewise cubic filter named for a spline, the Catmull-Rom filter (Figure 4.23), has the value zero at $x = -2, -1, 1,$ and 2, which means it will *interpolate* the samples when used as a reconstruction filter (Section 4.3.2):

$$f_C(x) = \frac{1}{2}\begin{cases} -3(1-|x|)^3 + 4(1-|x|)^2 + (1-|x|) & -1 \leq x \leq 1, \\ (2-|x|)^3 - (2-|x|)^2 & 1 \leq |x| \leq 2, \\ 0 & \text{otherwise.} \end{cases}$$

### The Mitchell-Netravali Cubic Filter

For the all-important application of resampling images, Mitchell and Netravali (Mitchell & Netravali, 1988) made a study of cubic filters and recommended one part way between the previous two filters as the best all-around choice (Figure 4.24). It is simply a weighted combination of the previous two filters:

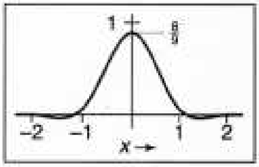


**Figure 4.24.** The Mitchell-Netravali filter.

$$\begin{aligned} f_M(x) &= \frac{1}{3}f_B(x) + \frac{2}{3}f_C(x) \\ &= \frac{1}{18}\begin{cases} -21(1-|x|)^3 + 27(1-|x|)^2 + 9(1-|x|) + 1 & -1 \leq x \leq 1, \\ 7(2-|x|)^3 - 6(2-|x|)^2 & 1 \leq |x| \leq 2, \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

### 4.3.2 Properties of Filters

Filters have some traditional terminology that goes with them, which we use to describe the filters and compare them to one another.

The *impulse response* of a filter is just another name for the function: it is the response of the filter to a signal that just contains an impluse (and recall that convolving with an impulse just gives back the filter).

A continuous filter is *interpolating* if, when it is used to reconstruct a continuous function from a discrete sequence, the resulting function takes on exactly the values of the samples at the sample points—that is, it "connects the dots" rather than producing a function that only goes near the dots. Interpolating filters are exactly those filters $f$ for which $f(0) = 1$ and $f(i) = 0$ for all non-zero integers $i$ (Figure 4.25).

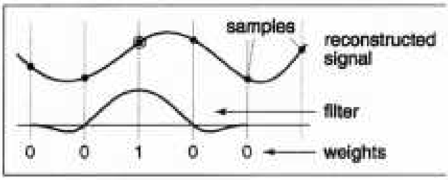


**Figure 4.25.** An interpolating filter reconstructs the sample points exactly because it has the value zero at all non-zero integer offsets from the center.

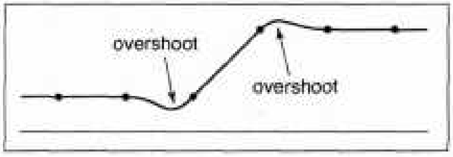


Figure 4.26. A filter with negative lobes will always produce some overshoot when filtering or reconstructing a sharp discontinuity.

A filter that takes on negative values has *ringing* or *overshoot*: it will produce extra oscillations in the value around sharp changes in the value of the function being filtered.

For instance, the Catmull-Rom filter has negative lobes on either side, and if you filter a step function with it, it will exaggerate the step a bit, resulting in function values that undershoot 0 and overshoot 1 (Figure 4.26).

A continuous filter is *ripple free* if, when used as a reconstruction filter, it will reconstruct a constant sequence as a constant function (Figure 4.27). This is equivalent to the requirement that the filter sum to one on any integer-spaced grid:

$$\sum_i f(x+i) = 1 \quad \text{for all } x.$$

A continuous filter has a *degree of continuity*, which is the highest-order derivative that is defined everywhere. A filter, like the box filter, that has sudden jumps in its value is not continuous at all. A filter that is continuous but has sharp corners (discontinuities in the first derivative), such as the tent filter, has order of continuity zero, and we say it is $C^0$. A filter that has a continuous derivative (no sharp corners), such as the piecewise cubic filters in the previous section, is $C^1$; if its second derivative is also continuous, as is true of the B-spline filter, it is $C^2$. The order of continuity of a filter is particularly important for a reconstruction filter because the reconstructed function inherits the continuity of the filter.

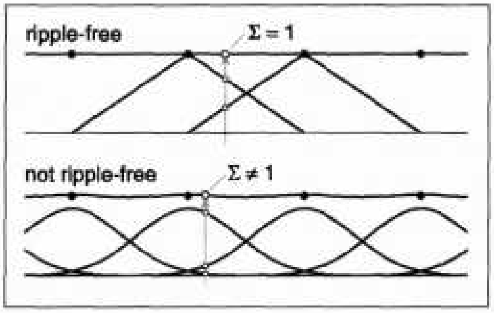


Figure 4.27. The tent filter of radius 1 is a ripple-free reconstruction filter; the Gaussian filter with standard deviation 1/2 is not.

### Separable Filters

So far we have only discussed filters for 1D convolution, but for images and other multidimensional signals we need filters too. In general, any 2D function could be a 2D filter, and occasionally it is useful to define them this way. But, in most cases, we can build suitable 2D (or higher-dimensional) filters from the 1D filters we have already seen.

The most useful way of doing this is by using a *separable* filter. The value of a separable filter $f_2(x, y)$ at a particular $x$ and $y$ is simply the product of $f_1$ (the 1D filter) evaluated at $x$ and at $y$:

$$f_2(x, y) = f_1(x)f_1(y).$$

Similarly, for discrete filters,

$$a_2[i, j] = a_1[i]a_1[j].$$

Any horizontal or vertical slice through $f_2$ is a scaled copy of $f_1$. The integral of $f_2$ is the square of the integral of $f_1$, so in particular if $f_1$ is normalized, then so is $f_2$.

**Example: The separable tent filter.** If we choose the tent function for $f_1$, the resulting piecewise bilinear function (Figure 4.28) is

$$f_{2,\text{tent}}(x, y) = \begin{cases} (1 - |x|)(1 - |y|) & |x| < 1 \quad \text{and} \quad |y| < 1, \\ 0 & \text{otherwise.} \end{cases}$$

The profiles along the coordinate axes are tent functions, but the profiles along the diagonals are quadratics (for instance, along the line $x = y$ in the positive quadrant, we see the quadratic function $(1 - x)^2$).

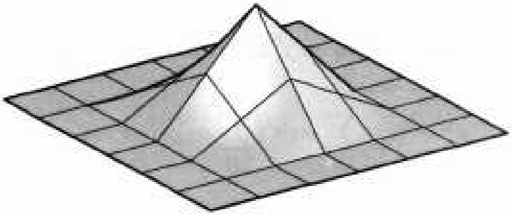


**Figure 4.28.** The separable 2D tent filter.

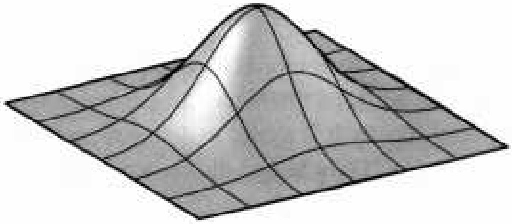


**Figure 4.29.** The 2D Gaussian filter, which is both separable and radially symmetric.

**Example: The 2D Gaussian filter.** If we choose the Gaussian function for $f_1$, the resulting 2D function (Figure 4.29) is

$$\begin{aligned} f_{2,g}(x,y) &= \frac{1}{2\pi}\left(e^{-x^2/2}e^{-y^2/2}\right), \\ &= \frac{1}{2\pi}\left(e^{-(x^2+y^2)/2}\right), \\ &= \frac{1}{2\pi}e^{-r^2/2}. \end{aligned}$$

Notice that this is (up to a scale factor) the same function we would get if we revolved the 1D Gaussian around the origin to produce a circularly symmetric function. The property of being both circularly symmetric and separable at the same time is unique to the Gaussian function. The profiles along the coordinate axes are Gaussians, but so are the profiles along any direction at any offset from the center.

The key advantage of separable filters over other 2D filters has to do with efficiency in implementation. Let's substitute the definition of $a_2$ into the definition of discrete convolution:

$$(a_2 \star b)[i,j] = \sum_{i'}\sum_{j'} a_1[i']a_1[j']b[i-i',j-j'].$$

Note that $a_1[i']$ does not depend on $j'$ and can be factored out of the inner sum:

$$= \sum_{i'} a_1[i'] \sum_{j'} a_1[j']b[i-i',j-j'].$$

Let's abbreviate the inner sum as $S[k]$:

$$S[k] = \sum_{j'} a_1[j']b[k, j - j'];$$
$$(a_2 \star b)[i, j] = \sum_{i'} a_1[i']S[i - i']. \tag{4.4}$$

With the equation in this form, we can first compute and store $S[i - i']$ for each value of $i'$, and then compute the outer sum using these stored values. At first glance this does not seem remarkable, since we still had to do work proportional to $(2r + 1)^2$ to compute all the inner sums. However, it's quite different if we want to compute the value at many points $[i, j]$.

Suppose we need to compute $a_2 \star b$ at $[2, 2]$ and $[3, 2]$, and $a_1$ has a radius of 2. Examining Equation 4.4, we can see that we will need $S[0], \ldots, S[4]$ to compute the result at $[2, 2]$, and we will need $S[1], \ldots, S[5]$ to compute the result at $[3, 2]$. So, in the separable formulation, we can just compute all six values of $S$ and share $S[1], \ldots, S[4]$ (Figure 4.30).

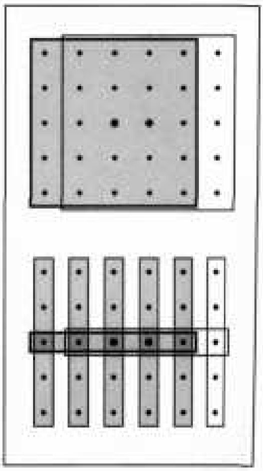


**Figure 4.30.** Computing two output points using separate 2D arrays of 25 samples (above) vs. filtering once along the columns, then using separate 1D arrays of five samples (below).

This savings has great significance for large filters. Filtering an $m$ by $n$ 2D image with a filter of radius $r$ in the general case requires computation of $(2r+1)^2$ products per pixel, while filtering the image with a separable filter of the same size requires $2(2r + 1)$ products (at the expense of some intermediate storage). This change in asymptotic complexity from $O(r^2)$ to $O(r)$ enables the use of much larger filters.

The algorithm is:

```
function filterImage(image $I$, filter $f$)
$r = f$.radius
$n_x = I$.width
$n_y = I$.height
allocate storage array $S[0, \ldots, n_x - 1]$
allocate image $I_{\text{out}}[r, \ldots, n_x - r - 1][r, \ldots, n_y - r - 1]$
initialize $S$ and $I_{\text{out}}$ to all zero
for $y = r$ to $n_y - r - 1$ do
   for $x = 0$ to $n_x - 1$ do
      $S[x] = 0$
      for $i = -r$ to $r$ do
         $S[x] = S[x] + f[i]I[x][y - i]$
   for $x = r$ to $n_x - r - 1$ do
      for $i = -r$ to $r$ do
         $I_{\text{out}}[x][y] = I_{\text{out}}[x][y] + f[i]S[x - i]$
return $I_{\text{out}}$
```

For simplicity, this function avoids all questions of boundaries by trimming $r$ pixels off all four sides of the output image. In practice there are various ways to handle the boundaries; see Section 4.4.3.

## 4.4 Signal Processing for Images

We have discussed sampling, filtering, and reconstruction in the abstract so far, using mostly 1D signals for examples. But as we observed at the beginning of the chapter, the most important and most common application of signal processing in graphics is for sampled images. Let us look carefully at how all this applies to images.

### 4.4.1 Image Filtering Using Discrete Filters

Perhaps the simplest application of convolution is processing images using discrete convolution. Some of the most widely used features of image manipulation programs are simple convolution filters. Blurring of images can be achieved by convolving with many common lowpass filters, ranging from the box to the Gaussian (Figure 4.31). A Gaussian filter creates a very smooth-looking blur and is

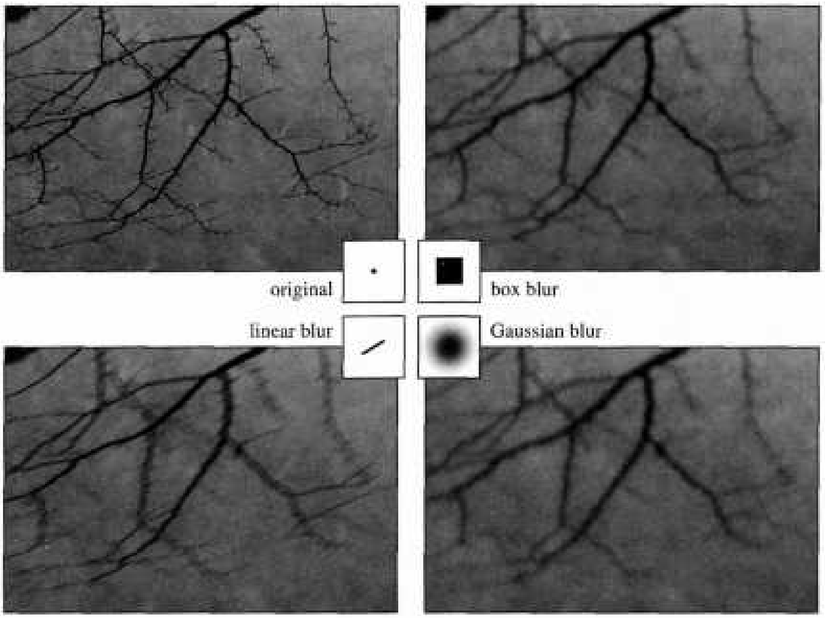


**Figure 4.31.** Blurring an image by convolution with each of three different filters.

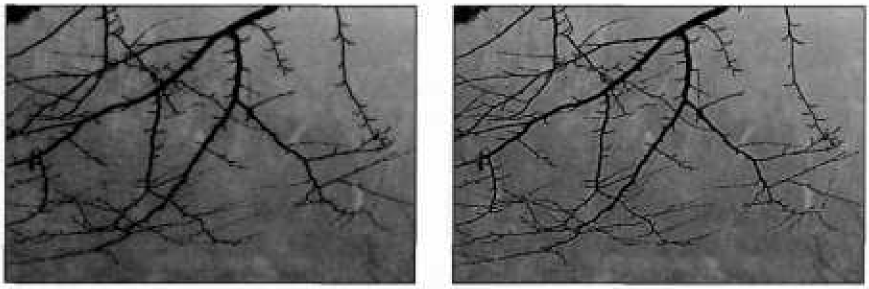


Figure 4.32. Sharpening an image using a convolution filter.

commonly used for this purpose.

The opposite of blurring is sharpening, and one way to do this is by using the "unsharp mask" procedure: subtract a fraction $\alpha$ of a blurred image from the original. With a rescaling to avoid changing the overall brightness, we have

$$\begin{aligned} I_{\text{sharp}} &= (1+\alpha)I - \alpha(f_{g,\sigma} \star I) \\ &= \left((1+\alpha)d - \alpha f_{g,\sigma}\right) \star I \\ &= f_{\text{sharp}}(\sigma, \alpha) \star I, \end{aligned}$$

where $f_{g,\sigma}$ is the Gaussian filter of width $\sigma$. Using the discrete impluse $d$ and the distributive property of convolution, we were able to write this whole process as a single filter that depends on both the width of the blur and the degree of sharpening (Figure 4.32).

Another example of combining two discrete filters is a drop shadow. It's common to take a blurred, shifted copy of an object's outline to create a soft drop shadow (Figure 4.33). We can express the shifting operation as convolution with an off-center impulse:

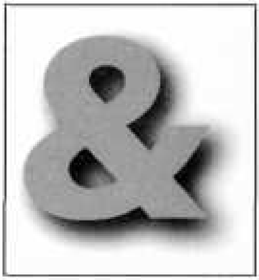


Figure 4.33. A soft drop shadow.

$$d_{m,n}(i,j) = \begin{cases} 1 & i = m \text{ and } j = n, \\ 0 & \text{otherwise.} \end{cases}$$

Shifting, then blurring, is achieved by convolving with both filters:

$$\begin{aligned} I_{\text{shadow}} &= f_{g,\sigma} \star (d_{m,n} \star I) \\ &= (f_{g,\sigma} \star d_{m,n}) \star I \\ &= f_{\text{shadow}}(m, n, \sigma) \star I. \end{aligned}$$

Here we have used associativity to group the two operations into a single filter with three parameters.

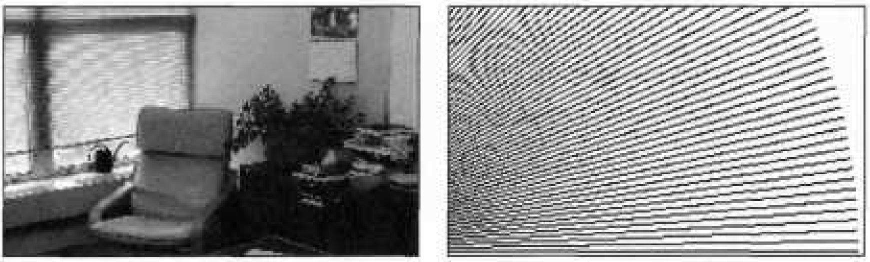


**Figure 4.34.** Two artifacts of aliasing in images: moiré patterns in periodic textures (left), and "jaggies" on straight lines (right).

### 4.4.2 Antialiasing in Image Sampling

In image synthesis, we often have the task of producing a sampled representation of an image for which we have a continuous mathematical formula (or at least a procedure we can use to compute the color at any point, not just at integer pixel positions). Ray tracing is a common example; we'll learn more about ray tracing and the specific methods for antialiasing in Chapter 10. In the language of signal processing, we have a continuous 2D signal (the image) that we need to sample on a regular 2D lattice. If we go ahead and sample the image without any special measures, the result will exhibit various aliasing artifacts (Figure 4.34). At sharp edges in the image, we see stair-step artifacts known as "jaggies." In areas where there are repeating patterns, we see wide bands known as *moiré patterns*.

The problem here is that the image contains too many small-scale features; we need to smooth it out by filtering it before sampling. Looking back at the definition of continuous convolution in Equation 4.3, we need to average the image over an area around the pixel location, rather than just taking the value at a single

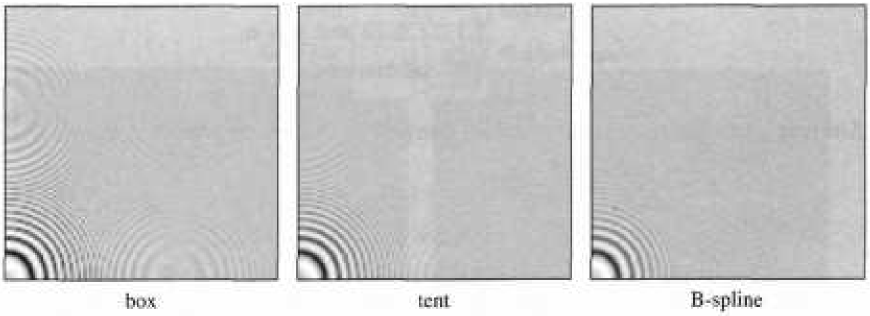


**Figure 4.35.** A comparison of three different sampling filters being used to antialias a difficult test image that contains circles that are spaced closer and closer as they get larger.

point. The specific methods for doing this are discussed in Chapter 10. A simple filter like a box will improve the appearance of sharp edges, but it still produces some moiré patterns (Figure 4.35). The Gaussian filter, which is very smooth, is much more effective against the moiré patterns, at the expense of overall somewhat more blurring. These two examples illustrate the tradeoff between sharpness and aliasing that is fundamental to choosing antialiasing filters.

### 4.4.3 Reconstruction and Resampling

One of the most common image operations where careful filtering is crucial is *resampling*—changing the sample rate, or changing the image size.

Suppose we have taken an image with a digital camera that is 3000 by 2000 pixels in size, and we want to display it on a monitor that has only 1280 by 1024 pixels. In order to make it fit, while maintaining the 3:2 aspect ratio, we need to resample it to 1278 by 852 pixels. How should we go about this?

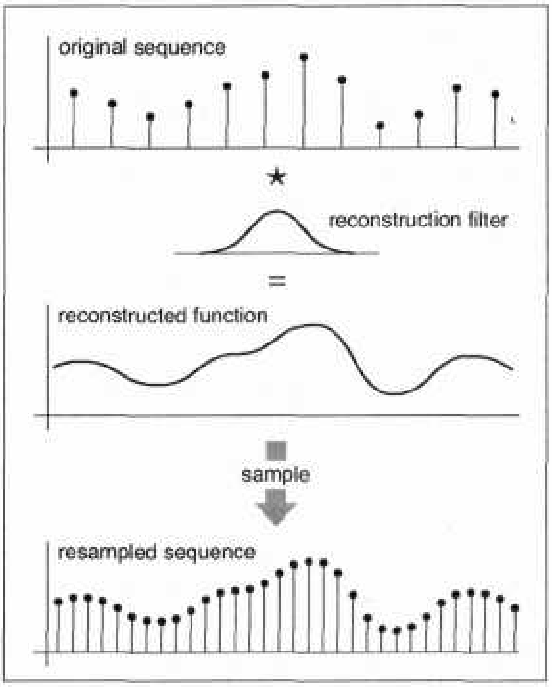


**Figure 4.36.** Resampling an image consists of two logical steps that are combined into a single operation in code. First, we use a reconstruction filter to define a smooth, continuous function from the input samples. Then, we sample that function on a new grid to get the output samples.

One way to approach this problem is to think of the process as dropping pixels: the size ratio is between 2 and 3, so we'll have to drop out one or two pixels between pixels that we keep. It's possible to shrink an image in this way, but the quality of the result is low—the images in Figure 4.34 were made using pixel dropping. Pixel dropping is very fast, however, and it is a reasonable choice to make a preview of the resized image during an interactive manipulation.

The way to think about resizing images is as a *resampling* operation: we want a set of samples of the image on a particular grid that is defined by the new image dimensions, and we get them by sampling a continuous function that is reconstructed from the input samples (Figure 4.36). Looking at it this way, it's just a sequence of standard image processing operations: first we reconstruct a continuous function from the input samples, and then we sample that function just as we would sample any other continuous image. To avoid aliasing artifacts, appropriate filters need to be used at each stage.

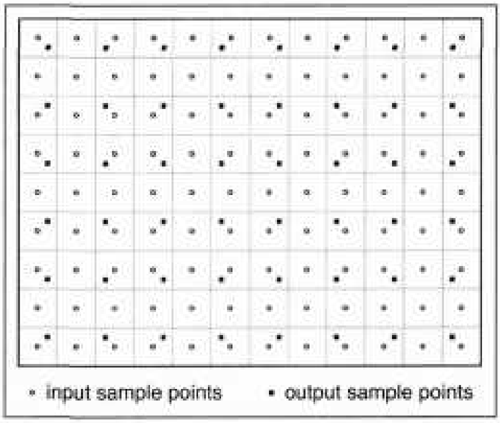


**Figure 4.37.** The sample locations for the input and output grids in resampling a 12 by 9 image to make an 8 by 6 one.

A small example is shown in Figure 4.37: if the original image is 12 by 9 pixels and the new one is 8 by 6 pixels, there are 2/3 as many output pixels as input pixels in each dimension, so their spacing across the image is 3/2 the spacing of the original samples.

In order to come up with a value for each of the output samples, we need to somehow compute values for the image in between the samples. The pixel-dropping algorithm gives us one way to do this: just take the value of the closest sample in the input image and make that the output value. This is exactly equivalent to reconstructing the image with a 1-pixel-wide box filter and then point sampling.

Of course, if the main reason for choosing pixel dropping or other very simple filtering is performance, one would never *implement* that method as a special case of the general reconstruction-and-resampling procedure. In fact, because of the discontinuities, it's difficult to make box filters work in a general framework. But, for high-quality resampling, the reconstruction/sampling framework provides valuable flexibility.

To work out the algorithmic details it's simplest to drop down to 1D and discuss resampling a sequence. The simplest way to write an implementation is in terms of the *reconstruct* function we defined in Section 4.2.5.

```
**function** resample(sequence $a$, float $x_0$, float $\Delta x$, int $n$, filter $f$)
create sequence $b$ of length $n$
**for** $i = 0$ to $n - 1$ **do**
    $b[i] = \text{reconstruct}(a, f, x_0 + i\Delta x)$
**return** $b$
```

The parameter $x_0$ gives the position of the first sample of the new sequence in terms of the samples of the old sequence. That is, if the first output sample falls midway between samples 3 and 4 in the input sequence, $x_0$ is 3.5.

This procedure reconstructs a continuous image by convolving the input sequence with a continuous filter and then point samples it. That's not to say that these two operations happen sequentially—the continuous function exists only in principle and its values are computed only at the sample points. But mathematically, this function computes a set of point samples of the function $a \star f$.

This point sampling seems wrong, though, because we just finished saying that a signal should be sampled with an appropriate smoothing filter to avoid aliasing. We should be convolving the reconstructed function with a sampling filter $g$ and point sampling $g \star (f \star a)$. But since this is the same as $(g \star f) \star a$, we can roll the sampling filter together with the reconstruction filter; one convolution operation is all we need (Figure 4.38). This combined reconstruction and sampling filter is known as a *resampling filter*.

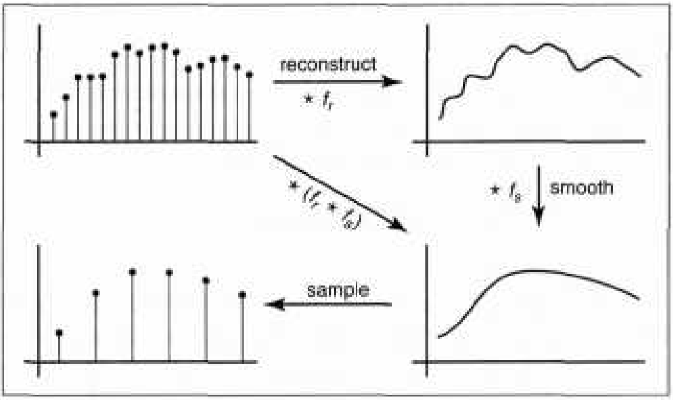


**Figure 4.38.** Resampling involves filtering for reconstruction and for sampling. Since two convolution filters applied in sequence can be replaced with a single filter, we only need one resampling filter, which serves the roles of reconstruction and sampling.

When resampling images, we usually specify a *source rectangle* in the units of the old image that specifies the part we want to keep in the new image. For example, using the pixel sample positioning convention from Chapter 3, the rectangle we'd use to resample the entire image is $(-0.5, n_x^{\text{old}} - 0.5) \times (-0.5, n_y^{\text{old}} - 0.5)$. Given a source rectangle $(x_l, x_h) \times (y_l, y_h)$, the sample spacing for the new image is $\Delta x = (x_h - x_l)/n_x^{\text{new}}$ in $x$ and $\Delta y = (y_h - y_l)/n_y^{\text{new}}$ in $y$. The lower left sample is positioned at $(x_l + \Delta x/2, y_l + \Delta y/2)$.

Modifying the 1D pseudocode to use this convention, and expanding the call to the reconstruct function into the double loop that is implied, we arrive at:

```
**function** resample(sequence $a$, float $x_l$, float $x_h$, int $n$, filter $f$)
create sequence $b$ of length $n$
$r = f$.radius
$x_0 = x_l + \Delta x/2$
**for** $i = 0$ to $n - 1$ **do**
  $s = 0$
  $x = x_0 + i\Delta x$
  **for** $j = \lceil x - r \rceil$ to $\lfloor x + r \rfloor$ **do**
    $s = s + a[j]f(x - j)$
  $b[i] = s$
**return** $b$
```

This routine contains all the basics of resampling an image. One last issue that remains to be addressed is what to do at the edges of the image, where the simple version here will access beyond the bounds of the input sequence. There are several things we might do:

- Just stop the loop at the ends of the sequence. This is equivalent to padding the image with zeros on all sides;
- Clip all array accesses to the end of the sequence—that is, return $a[0]$ when we would want to access $a[-1]$. This is equivalent to padding the edges of the image by extending the last row or column;
- Modify the filter as we approach the edge so that it does not extend beyond the bounds of the sequence.

The first option leads to dim edges when we resample the whole image, which is not really satisfactory. The second option is easy to implement; the third is probably the best performing. The simplest way to modify the filter near the edge of the image is to *renormalize* it: divide the filter by the sum of the part of the filter that falls within the image. This way, the filter always adds up to 1 over the actual image samples, so it preserves image intensity. For performance, it is desirable

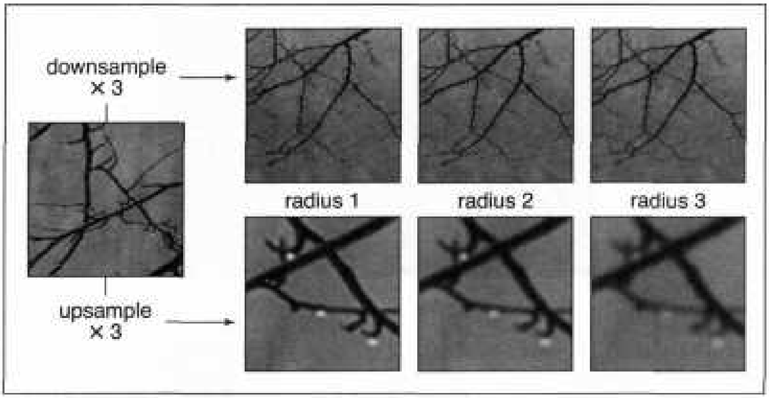


**Figure 4.39.** The effects of using different sizes of a filter for upsampling (enlarging) or downsampling (reducing) an image.

to handle the band of pixels within a filter radius of the edge (which require this renormalization) separately from the center (which contains many more pixels and does not require renormalization).

The choice of filter for resampling is important. There are two separate issues: the shape of the filter and the size (radius). Because the filter serves both as a reconstruction filter and a sampling filter, the requirements of both roles affect the choice of filter. For reconstruction, we would like a filter smooth enough to avoid aliasing artifacts when we enlarge the image, and the filter should be ripple-free. For sampling, the filter should be large enough to avoid undersampling and smooth enough to avoid moiré artifacts. Figure 4.39 illustrates these two different needs.

Generally we will choose one filter shape and scale it according to the relative resolutions of the input and output. The lower of the two resolutions determines the size of the filter: when the output is more coarsely sampled than the input (downsampling, or shrinking the image), the smoothing required for proper sampling is greater than the smoothing required for reconstruction, so we size the filter according to the output sample spacing (radius 3 in Figure 4.39). On the other hand, when the output is more finely sampled (upsampling, or enlarging the image) then the smoothing required for reconstruction dominates (the reconstructed function is already smooth enough to sample at a higher rate than it started), so the size of the filter is determined by the input sample spacing (radius 1 in Figure 4.39).

Choosing the filter itself is a tradeoff between speed and quality. Common choices are the box filter (when speed is paramount), the tent filter (moderate quality), or a piecewise cubic (excellent quality). In the piecewise cubic case, the

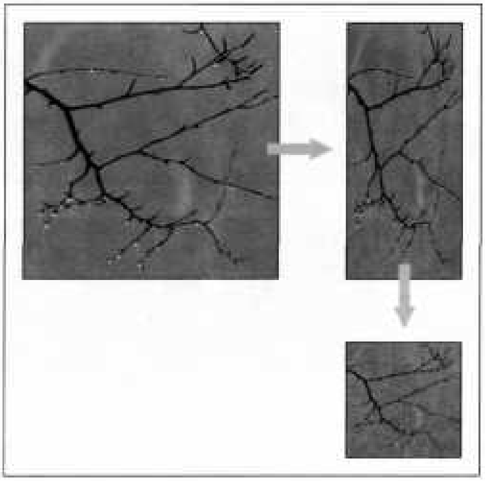


**Figure 4.40.** Resampling an image using a separable approach.

degree of smoothing can be adjusted by interpolating between $f_B$ and $f_C$; the Mitchell-Netravali filter is a good choice.

Just as with image filtering, separable filters can provide a significant speedup. The basic idea is to resample all the rows first, producing an image with changed width but not height, then to resample the columns of that image to produce the final result (Figure 4.40). Modifying the pseudocode given earlier so that it takes advantage of this optimization is reasonably straightforward.

## 4.5 Sampling Theory

If you are only interested in implementation, you can stop reading here; the algorithms and recommendations in the previous sections will let you implement programs that perform sampling and reconstruction and achieve excellent results. However, there is a deeper mathematical theory of sampling with a history reaching back to the first uses of sampled representations in telecommunications. Sampling theory answers many questions that are difficult to answer with reasoning based strictly on scale arguments.

But most important, sampling theory gives valuable insight into the workings of sampling and reconstruction. It gives the student who learns it an extra set of intellectual tools for reasoning about how to achieve the best results with the most efficient code.

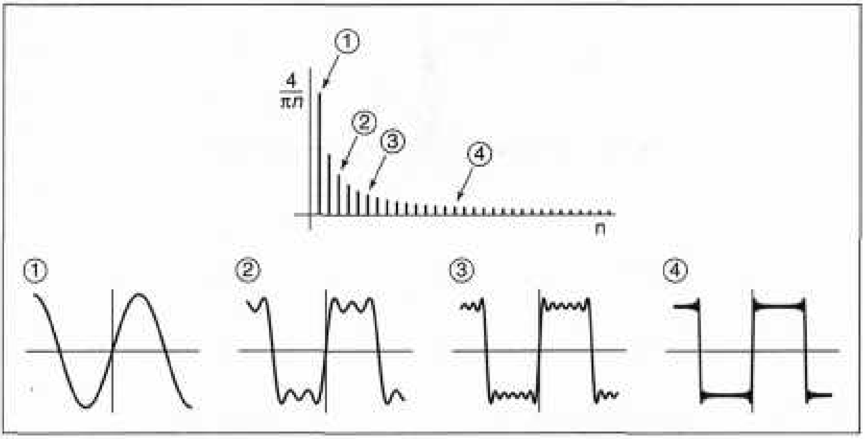


**Figure 4.41.** Approximating a square wave with finite sums of sines.

### 4.5.1 The Fourier Transform

The Fourier transform, along with convolution, is the main mathematical concept that underlies sampling theory. You can read about the Fourier transform in many math books on analysis, as well as in books on signal processing.

The basic idea behind the Fourier transform is to express any function by adding together sine waves (sinusoids) of all frequencies. By using the appropriate weights for the different frequencies, we can arrange for the sinusoids to add up to any (reasonable) function we want.

As an example, the square wave in Figure 4.41 can be expressed by a sequence of sine waves:

$$\sum_{n=1,3,5,\ldots}^{\infty} \frac{4}{\pi n} \sin 2\pi n x.$$

This *Fourier series* starts with a sine wave ($\sin 2\pi x$) that has frequency 1.0—same as the square wave—and the remaining terms add smaller and smaller corrections to reduce the ripples and, in the limit, reproduce the square wave exactly. Note that all the terms in the sum have frequencies that are integer multiples of the frequency of the square wave. This is because other frequencies would produce results that don't have the same period as the square wave.

A surprising fact is that a signal does not have to be periodic in order to be expressed as a sum of sinusoids in this way: a non-periodic signal just requires more sinusoids. Rather than summing over a discrete sequence of sinusoids, we will instead integrate over a continuous family of sinusoids. For instance, a box

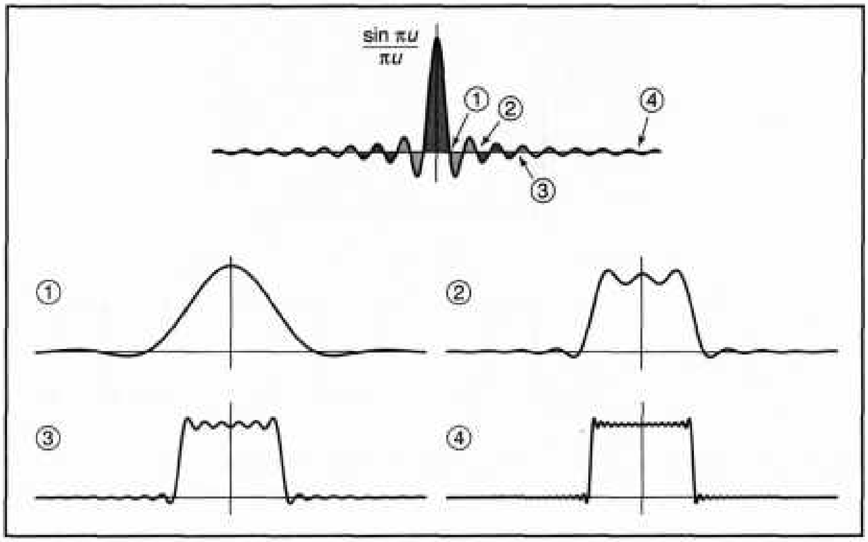


**Figure 4.42.** Approximating a box function with integrals of cosines up to each of four cutoff frequencies.

function can be written as the integral of a family of cosine waves:

$$\int_{-\infty}^{\infty} \frac{\sin \pi u}{\pi u} \cos 2\pi u x \, du. \tag{4.5}$$

This integral in Equation 4.5 is adding up infinitely many cosines, weighting the cosine of frequency $u$ by the weight $(\sin \pi u)/\pi u$. The result, as we include higher and higher frequencies, converges to the box function (see Figure 4.42). When a function $f$ is expressed in this way, this weight, which is a function of the frequency $u$, is called the *Fourier transform* of $f$, denoted $\hat{f}$. The function $\hat{f}$ tells us how to build $f$ by integrating over a family of sinusoids:

$$f(x) = \int_{-\infty}^{\infty} \hat{f}(u) e^{2\pi i u x} du. \tag{4.6}$$

Equation 4.6 is known as the *inverse Fourier transform* (IFT) because it starts with the Fourier transform of $f$ and ends up with $f$.[2]

Note that in Equation 4.6 the complex exponential $e^{2\pi i u x}$ has been substituted for the cosine in the previous equation. Also, $\hat{f}$ is a complex-valued function.

[2] Note that the term "Fourier transform" is used both for the function $\hat{f}$ and for the operation that computes $\hat{f}$ from $f$. Unfortunately, this rather ambiguous usage is standard.

The machinery of complex numbers is required to allow the phase, as well as the frequency, of the sinusoids to be controlled; this is necessary to represent any functions that are not symmetric across zero. The magnitude of $\hat{f}$ is known as the *Fourier spectrum*, and, for our purposes, this is sufficient—we won't need to worry about phase or use any complex numbers directly.

It turns out that computing $\hat{f}$ from $f$ looks very much like computing $f$ from $\hat{f}$:

$$\hat{f}(u) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i u x} dx. \tag{4.7}$$

Equation 4.7 is known as the (forward) *Fourier transform* (FT). The sign in the exponential is the only difference between the forward and inverse Fourier transforms, and it is really just a technical detail. For our purposes, we can think of the FT and IFT as the same operation.

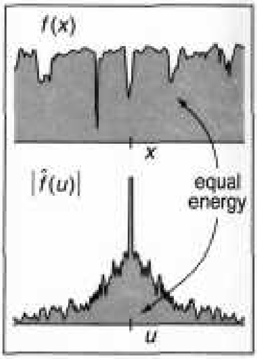


**Figure 4.43.** The Fourier transform preserves the squared integral of the signal.

Sometimes the $f$–$\hat{f}$ notation is inconvenient, and then we will denote the Fourier transform of $f$ by $\mathcal{F}\{f\}$ and the inverse Fourier transform of $\hat{f}$ by $\mathcal{F}^{-1}\{\hat{f}\}$.

A function and its Fourier transform are related in many useful ways. A few facts (most of them easy to verify) that we will use later in the chapter are:

- A function and its Fourier transform have the same squared integral:

  $$\int (f(x))^2 dx = \int (\hat{f}(u))^2 du.$$

  The physical interpretation is that the two have the same energy (Figure 4.43).

  In particular, scaling a function up by $a$ also scales its Fourier transform by $a$. That is, $\mathcal{F}\{af\} = a\mathcal{F}\{f\}$.

- Stretching a function along the $x$-axis squashes its Fourier transform along the $u$-axis by the same factor (Figure 4.44):

  $$\mathcal{F}\{f(x/b)\} = b\hat{f}(bx).$$

  (The renormalization by $b$ is needed to keep the energy the same.)

  This means that if we are interested in a family of functions of different width and height (say all box functions centered at zero), then we only need to know the Fourier transform of one canonical function (say the box function with width and height equal to one), and we can easily know the Fourier transforms of all the scaled and dilated versions of that function.

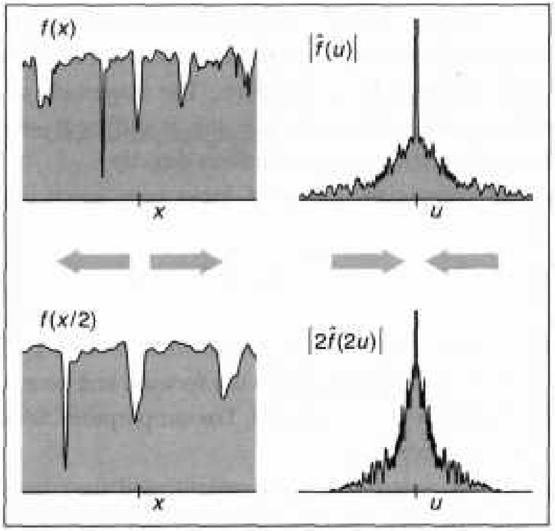


**Figure 4.44.** Scaling a signal along the *x*-axis in the space domain causes an inverse scale along the *u*-axis in the frequency domain.

For example, we can instantly generalize Equation 4.5 to give the Fourier transform of a box of width $b$ and height $a$:

$$ab\frac{\sin \pi bu}{\pi bu}.$$

- The average value of $f$ is equal to $\hat{f}(0)$. This makes sense since $\hat{f}(0)$ is supposed to be the zero-frequency component of the signal (the DC component if we are thinking of an electrical voltage).
- If $f$ is real (which it always is for us), $\hat{f}$ is an even function—that is, $\hat{f}(u) = \hat{f}(-u)$. Likewise, if $f$ is an even function then $\hat{f}$ will be real (this is not usually the case in our domain, but remember that we really are only going to care about the magnitude of $\hat{f}$).

### 4.5.2 Convolution and the Fourier Transform

One final property of the Fourier transform that deserves special mention is its relationship to convolution (Figure 4.45). Briefly,

$$\mathcal{F}\{f \star g\} = \hat{f}\hat{g}.$$

$$\begin{tikzcd}
f \arrow[rr] \arrow[drr] \arrow[ddddd, "\mathcal{F}"] & & \star \arrow[rr] & & f \star g \arrow[ddddd, "\mathcal{F}"] \\
& g \arrow[r] \arrow[ur] \arrow[ddd, "\mathcal{F}"] & \times \arrow[r] & fg \arrow[ddd, "\mathcal{F}"] & \\
& & \text{space domain} & & \\
& & \text{frequency domain} & & \\
& \hat{g} \arrow[r] \arrow[dr] & \star \arrow[r] & \hat{f} \star \hat{g} & \\
\hat{f} \arrow[rr] \arrow[urr] & & \times \arrow[rr] & & \hat{f}\hat{g}
\end{tikzcd}$$

**Figure 4.45.** A commutative diagram to show visually the relationship between convolution and multiplication. If we multiply *f* and *g* in space, then transform to frequency, we end up in the same place as if we transformed *f* and *g* to frequency and then convolved them. Likewise, if we convolve *f* and *g* in space and then transform into frequency, we end up in the same place as if we transformed *f* and *g* to frequency, then multiplied them.

The Fourier transform of the convolution of two functions is the product of the Fourier transforms. Following the by now familiar symmetry,

$$\hat{f} \star \hat{g} = \mathcal{F}\{fg\}.$$

The convolution of two Fourier transforms is the Fourier transform of the product of the two functions. These facts are fairly straightforward to derive from the definitions.

This relationship is the main reason Fourier transforms are useful in studying the effects of sampling and reconstruction. We've seen how sampling, filtering, and reconstruction can be seen in terms of convolution; now the Fourier transform gives us a new domain—the frequency domain—in which these operations are simply products.

### 4.5.3 A Gallery of Fourier Transforms

Now that we have some facts about Fourier transforms, let's look at some examples of individual functions. In particular, we'll look at some filters from Section 4.3.1, which are shown with their Fourier transforms in Figure 4.46. We have already seen the box function:

$$\mathcal{F}\{f_{\text{box}}\} = \frac{\sin \pi u}{\pi u} = \operatorname{sinc} \pi u.$$

The function[3] $\sin x/x$ is important enough to have its own name, sinc $x$.

[3] You may notice that $\sin \pi u/\pi u$ is undefined for $u = 0$. It is, however, continuous across zero, and we take it as understood that we use the limiting value of this ratio, 1, at $u = 0$.

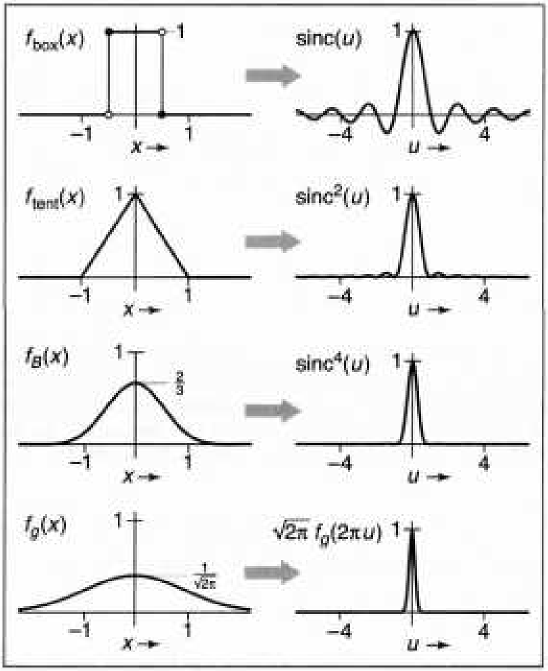


**Figure 4.46.** The Fourier transforms of the box, tent, B-spline, and Gaussian filters.

The tent function is the convolution of the box with itself, so its Fourier transform is just the square of the Fourier transform of the box function:

$$\mathcal{F}\{f_{\text{tent}}\} = \frac{\sin^2 \pi u}{\pi^2 u^2} = \text{sinc}^2 \pi u.$$

We can continue this process to get the Fourier transform of the B-spline filter (see Exercise 3):

$$\mathcal{F}\{f_{\text{B}}\} = \frac{\sin^4 \pi u}{\pi^4 u^4} = \text{sinc}^4 \pi u.$$

The Gaussian has a particularly nice Fourier transform:

$$\mathcal{F}\{f_{\text{G}}\} = e^{-(2\pi u)^2/2}.$$

It is another Gaussian! The Gaussian with standard deviation 1.0 becomes a Gaussian with standard deviation $1/2\pi$.

### 4.5.4 Dirac Impulses in Sampling Theory

The reason impulses are useful in sampling theory is that we can use them to talk about samples in the context of continuous functions and Fourier transforms. We represent a sample, which has a position and a value, by an impulse translated to that position and scaled by that value. A sample at position $a$ with value $b$ is represented by $b\delta(x-a)$. This way we can express the operation of sampling the function $f(x)$ at $a$ as multiplying $f$ by $\delta(x-a)$. The result is $f(a)\delta(x-a)$.

Sampling a function at a series of equally spaced points is therefore expressed as multiplying the function by the sum of a series of equally spaced impulses, called an *impulse train* (Figure 4.47). An impulse train with period $T$, meaning that the impulses are spaced a distance $T$ apart is

$$s_T(x) = \sum_{i=-\infty}^{\infty} \delta(x - Ti).$$

The Fourier transform of $s_1$ is the same as $s_1$: a sequence of impulses at all integer frequencies. You can see why this should be true by thinking about what happens when we multiply the impulse train by a sinusoid and integrate. We wind up adding up the values of the sinusoid at all the integers. This sum will exactly cancel to zero for non-integer frequencies, and it will diverge to $+\infty$ for integer frequencies.

Because of the dilation property of the Fourier transform, we can guess that the Fourier transform of an impulse train with period $T$ (which is like a dilation of $s_1$) is an impulse train with period $1/T$. Making the sampling finer in the space domain makes the impulses farther apart in the frequency domain.

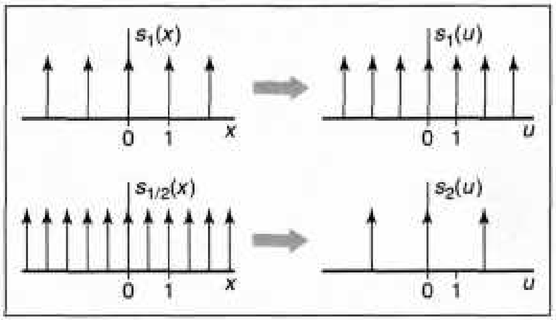


**Figure 4.47.** Impulse trains. The Fourier transform of an impulse train is another impulse train. Changing the period of the impulse train in space causes an inverse change in the period in frequency.

### 4.5.5 Sampling and Aliasing

Now we have built the mathematical machinery we need to understand the sampling and reconstruction process from the viewpoint of the frequency domian. The key advantage of introducing Fourier transforms is that it makes the effects of convolution filtering on the signal much clearer, and it provides more precise explanations of why we need to filter when sampling and reconstructing.

We start the process with the original, continuous signal. In general its Fourier transform could include components at any frequency, although for most kinds of signals (especially images), we expect the content to decrease as the frequency gets higher. Images also tend to have a large component at zero frequency—remember that the zero-frequency, or DC, component is the integral of the whole image, and since images are all positive values this tends to be a large number.

Let's see what happens to the Fourier transform if we sample and reconstruct without doing any special filtering (Figure 4.48). When we sample the signal, we model the operation as multiplication with an impulse train; the sampled signal is $fs_T$. Because of the multiplication-convolution property, the FT of the sampled signal is $\hat{f} \star \hat{s_T} = \hat{f} \star s_{1/T}$.

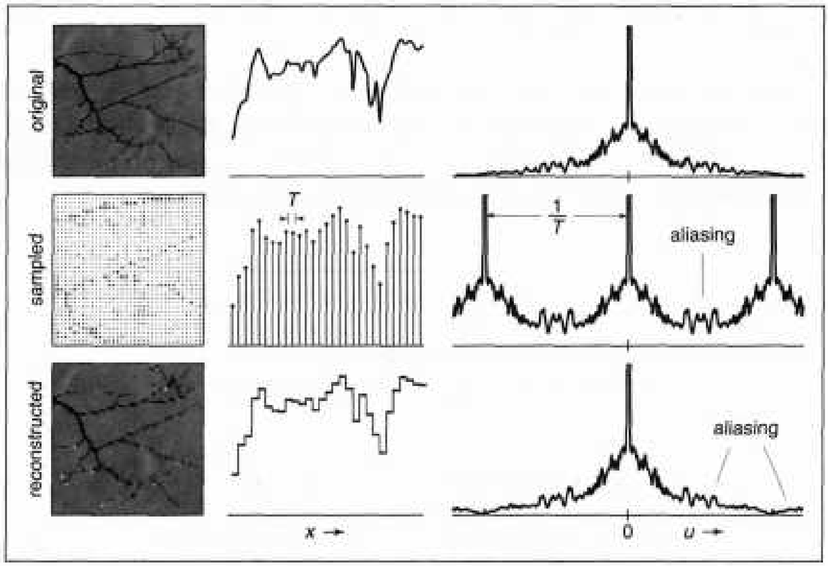


**Figure 4.48.** Sampling and reconstruction with no filtering. Sampling produces alias spectra that overlap and mix with the base spectrum. Reconstruction with a box filter collects even more information from the alias spectra. The result is a signal that has serious aliasing artifacts.

Recall that $\delta$ is the identity for convolution. This means that

$$(\hat{f} \star s_{1/T})(u) = \sum_{i=-\infty}^{\infty} \hat{f}(u - i/T);$$

that is, convolving with the impulse train makes a whole series of equally spaced copies of the spectrum of $f$. A good intuitive interpretation of this seemingly odd result is that all those copies just express the fact (as we saw back in Section 4.1.1) that frequencies that differ by an integer multiple of the sampling frequency are indistinguishable once we have sampled—they will produce exactly the same set of samples. The original spectrum is called the *base spectrum* and the copies are known as *alias spectra*.

The trouble begins if these copies of the signal's spectrum overlap, which will happen if the signal contains any significant content beyond half the sample frequency. When this happens, the spectra add, and the information about different frequencies is irreversibly mixed up. This is the first place aliasing can occur, and if it happens here, it's due to undersampling—using too low a sample frequency for the signal.

Suppose we reconstruct the signal using the nearest-neighbor technique. This is equivalent to convolving with a box of width 1. (The discrete-continuous convolution used to do this is the same as a continuous convolution with the series of impulses that represent the samples.) The convolution-multiplication property means that the spectrum of the reconstructed signal will be the product of the spectrum of the sampled signal and the spectrum of the box. The resulting reconstructed Fourier transform contains the base spectrum (though somewhat attenuated at higher frequencies), plus attenuated copies of all the alias spectra. Because the box has a fairly broad Fourier transform, these attenuated bits of alias spectra are significant, and they are the second form of aliasing, due to an inadequate reconstruction filter. These alias components manifest themselves in the image as the pattern of squares that is characteristic of nearest-neighbor reconstruction.

### Preventing Aliasing in Sampling

To do high quality sampling and reconstruction, we have seen that we need to choose sampling and reconstruction filters appropriately. From the standpoint of the frequency domain, the purpose of lowpass filtering when sampling is to limit the frequency range of the signal so that the alias spectra do not overlap the base spectrum. Figure 4.49 shows the effect of sample rate on the Fourier transform of the sampled signal. Higher sample rates move the alias spectra farther apart, and eventually whatever overlap is left does not matter.

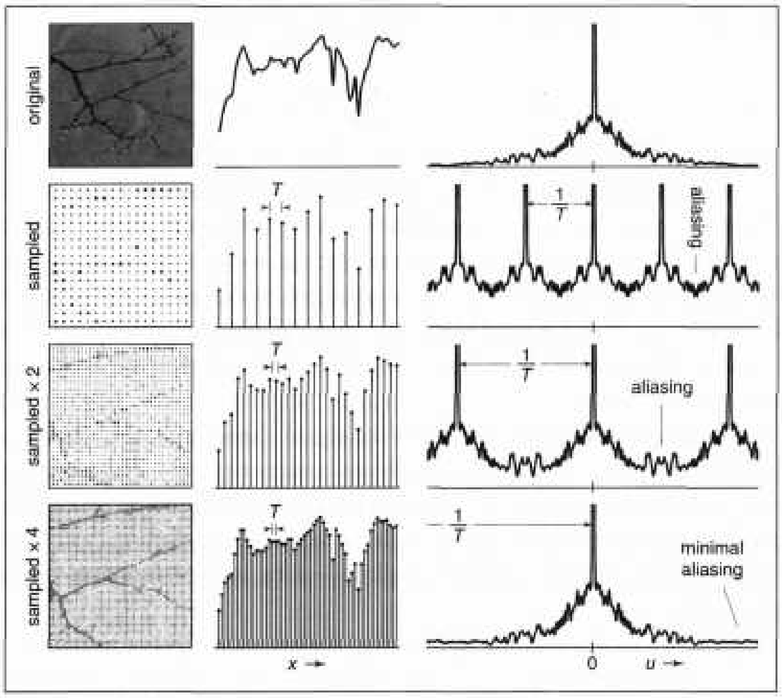


**Figure 4.49.** The effect of sample rate on the frequency spectrum of the sampled signal. Higher sample rates push the copies of the spectrum apart, reducing problems caused by overlap.

The key criterion is that the width of the spectrum must be less than the distance between the copies—that is, the highest frequency present in the signal must be less than half the sample frequency. This is known as the *Nyquist criterion*, and the highest allowable frequency is known as the *Nyquist frequency* or *Nyquist limit*. The *Nyquist-Shannon sampling theorem* states that a signal whose frequencies do not exceed the Nyquist limit (or, said another way, a signal that is bandlimited to the Nyquist frequency) can, in principle, be reconstructed exactly from samples.

With a high enough sample rate for a particular signal, we don't need to use a sampling filter. But if we are stuck with a signal that contains a wide range of frequencies (such as an image with sharp edges in it), we must use a sampling filter to bandlimit the signal before we can sample it. Figure 4.50 shows the effects of three lowpass (smoothing) filters in the frequency domain, and Figure 4.51 shows the effect of using these same filters when sampling. Even if the spectra overlap without filtering, convolving the signal with a lowpass filter can narrow the spectrum enough to eliminate overlap and produce a well-sampled

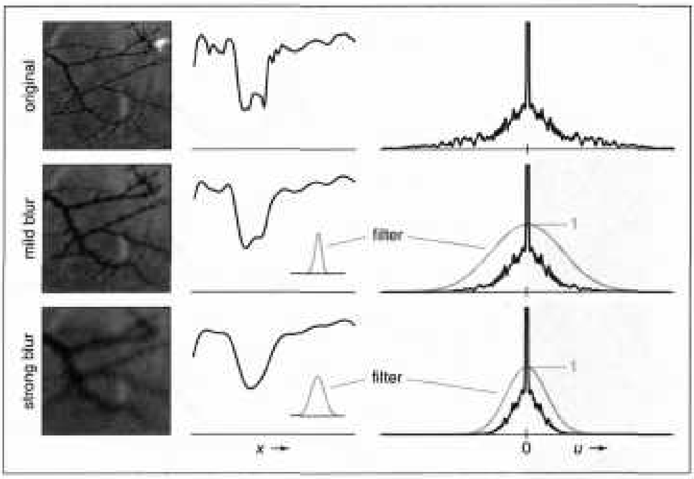


**Figure 4.50.** Applying lowpass (smoothing) filters narrows the frequency spectrum of a signal.

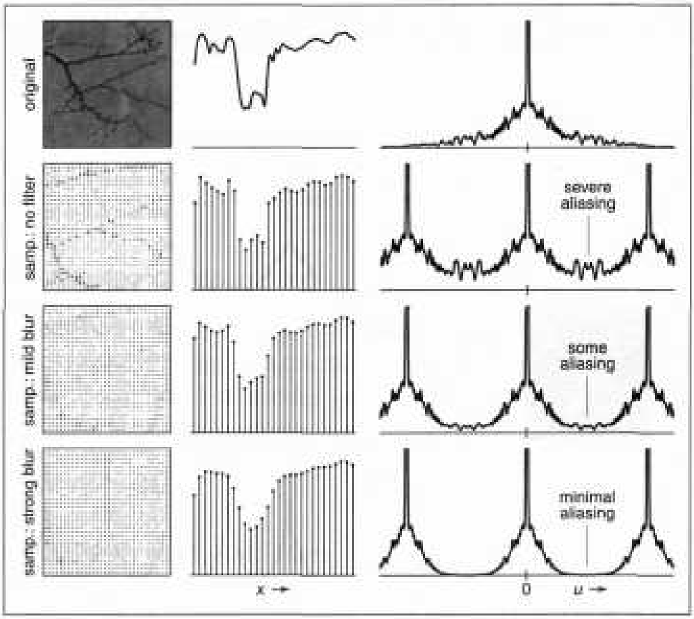


**Figure 4.51.** *How the lowpass filters from Figure 4.50 prevent aliasing during sampling.* Lowpass filtering narrows the spectrum so that the copies overlap less, and the high frequencies from the alias spectra interfere less with the base spectrum.

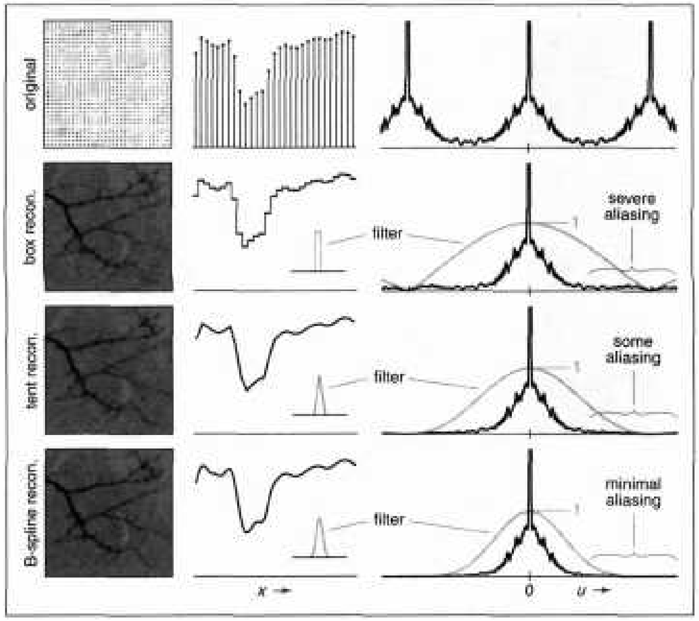


**Figure 4.52.** The effects of different reconstruction filters in the frequency domain. A good reconstruction filter attenuates the alias spectra effectively while preserving the base spectrum.

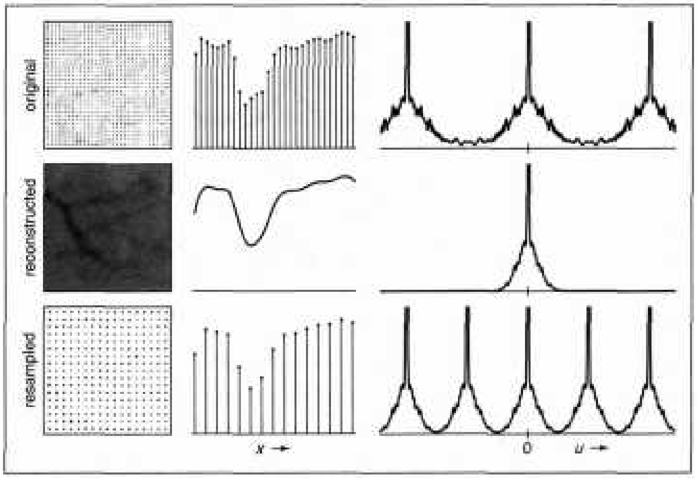


**Figure 4.53.** Resampling viewed in the frequency domain. The resampling filter both reconstructs the signal (removes the alias spectra) and bandlimits it (reduces its width) for sampling at the new rate.

representation of the filtered signal. Of course, we have lost the high frequencies, but that's better than having them get scrambled with the signal and turn into artifacts.

### Preventing Aliasing in Reconstruction

From the frequency domain perspective, the job of a reconstruction filter is to remove the alias spectra while preserving the base spectrum. In Figure 4.48, we can see that the crudest reconstruction filter, the box, does attenuate the alias spectra. Most important, it completely blocks the DC spike for all the alias spectra. This is a characteristic of all reasonable reconstruction filters: they have zeroes in frequency space at all multiples of the sample frequency. This turns out to be equivalent to the ripple-free property in the space domain.

So a good reconstruction filter needs to be a good lowpass filter, with the added requirement of completely blocking all multiples of the sample frequency. The purpose of using a reconstruction filter different from the box filter is to more completely eliminate the alias spectra, reducing the leakage of high-frequency artifacts into the reconstructed signal, while disturbing the base spectrum as little as possible. Figure 4.52 illustrates the effects of different filters when used during reconstruction. As we have seen, the box filter is quite "leaky" and results in plenty of artifacts even if the sample rate is high enough. The tent filter, resulting in linear interpolation, attenuates high frequencies more, resulting in milder artifacts, and the B-spline filter is very smooth, controlling the alias spectra very effectively. It also smooths the base spectrum some—this is the tradeoff between smoothing and aliasing that we saw earlier.

### Preventing Aliasing in Resampling

When the operations of reconstruction and sampling are combined in resampling, the same principles apply, but with one filter doing the work of both reconstruction and sampling. Figure 4.53 illustrates how a resampling filter must remove the alias spectra *and* leave the spectrum narrow enough to be sampled at the new sample rate.

## 4.5.6 Ideal Filters vs Useful Filters

Following the frequency domain analysis to its logical conclusion, a filter that is exactly a box in the frequency domain is ideal for both sampling and reconstruction. Such a filter would prevent aliasing at both stages without diminishing the frequencies below the Nyquist frequency at all.

Recall that the inverse and forward Fourier transforms are essentially identical, so the spatial domain filter that has a box as its Fourier transform is the function $\sin \pi x / \pi x = \text{sinc}\ \pi x$.

However, the sinc filter is not generally used in practice, either for sampling or for reconstruction, because it is impractical and because, even though it is optimal according to the frequency domain criteria, it doesn't produce the best results for many applications.

For sampling, the infinite extent of the sinc filter, and its relatively slow rate of decrease with distance from the center, is a liability. Also, for some kinds of sampling, the negative lobes are problematic. A Gaussian filter makes an excellent sampling filter even for difficult cases where high-frequency patterns must be removed from the input signal, because its Fourier transform falls off exponentially, with no bumps that tend to let aliases leak through. For less difficult cases, a tent filter generally suffices.

For reconstruction, the size of the sinc function again creates problems, but even more importantly, the many ripples create "ringing" artifacts in reconstructed signals.

## Exercises

1. Show that discrete convolution is commutative and associative. Do the same for continuous convolution.

2. Discrete-continuous convolution can't be commutative, because its arguments have two different types. Show that it is associative, though.

3. Prove that the B-spline is the convolution of four box functions.

4. Show that the "flipped" definition of convolution is necessary by trying to show that convolution is commutative and associative using this (incorrect) definition (see the footnote on page 80):

$$(a \star b)[i] = \sum_j a[j]b[i + j]$$

5. Prove that $\mathcal{F}\{f \star g\} = \hat{f}\hat{g}$ and $\hat{f} \star \hat{g} = \mathcal{F}\{fg\}$.

# 5

# Linear Algebra

Perhaps the most universal tools of graphics programs are the matrices that change or *transform* points and vectors. In the next chapter, we will see how a vector can be represented as a matrix with a single column, and how the vector can be represented in a different basis via multiplication with a square matrix. We will also describe how we can use such multiplications to accomplish changes in the vector such as scaling, rotation, and translation. In this chapter, we review basic linear algebra from a geometric perspective. This chapter can be skipped by readers comfortable with linear algebra. However, there may be some enlightening tidbits even for such readers, such as the development of determinants and the discussion of singular and eigenvalue decomposition.

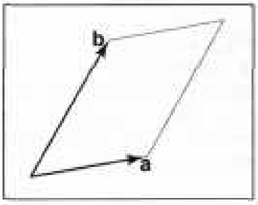


**Figure 5.1.** The signed area of the parallelogram is $|\mathbf{ab}|$, and in this case the area is positive.

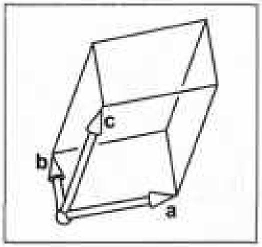


**Figure 5.2.** The signed volume of the parallelepiped shown is denoted by the determinant $|\mathbf{abc}|$, and in this case the volume is positive because the vectors form a right-handed basis.

## 5.1 Determinants

We usually think of determinants as arising in the solution of linear equations. However, for our purposes, we will think of determinants as another way to multiply vectors. For 2D vectors $\mathbf{a}$ and $\mathbf{b}$, the determinant $|\mathbf{ab}|$ is the area of the parallelogram formed by $\mathbf{a}$ and $\mathbf{b}$ (Figure 5.1). This is a signed area, and the sign is positive if $\mathbf{a}$ and $\mathbf{b}$ are right-handed and negative if they are left-handed. This means $|\mathbf{ab}| = -|\mathbf{ba}|$. In 2D we can interpret "right-handed" as meaning we would rotate the first vector counterclockwise to close the smallest angle to the second vector. In 3D the determinant must be taken with three vectors at a time. For three 3D vectors, $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$, the determinant $|\mathbf{abc}|$ is the signed volume of the parallelepiped (3D parallelogram; a sheared 3D box) formed by the three vec-

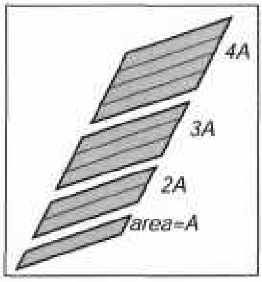


**Figure 5.3.** Scaling a parallelogram along one direction changes the area in the same proportion.

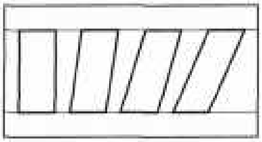


**Figure 5.4.** Shearing a parallelogram does not change its area. These four parallelograms have the same length base and thus the same area.

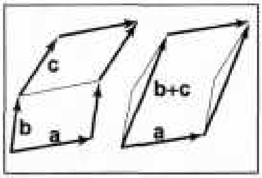


**Figure 5.5.** The geometry behind Equation 5.1. Both of the parallelograms on the left can be sheared to cover the single parallelogram on the right.

tors (Figure 5.2).To compute a 2D determinant, we first need to establish a few properties of the determinant. We note that scaling one side of a parallelogram scales its area by the same fraction (Figure 5.3):

$$|(k\mathbf{a})\mathbf{b}| = |\mathbf{a}(k\mathbf{b})| = k|\mathbf{ab}|.$$

Also, we note that "shearing" a parallelogram does not change its area (Figure 5.4):

$$|(\mathbf{a} + k\mathbf{b})\mathbf{b}| = |\mathbf{a}(\mathbf{b} + k\mathbf{a})| = |\mathbf{ab}|.$$

Finally, we see that the determinant has the following property:

$$|\mathbf{a}(\mathbf{b} + \mathbf{c})| = |\mathbf{ab}| + |\mathbf{ac}|, \tag{5.1}$$

because as shown in Figure 5.5 we can "slide" the edge between the two parallelograms over to form a single parallelogram without changing the area of either of the two original parallelograms.

Now let's assume a Cartesian representation for $\mathbf{a}$ and $\mathbf{b}$:

$$\begin{aligned}|\mathbf{ab}| &= |(x_a\mathbf{x} + y_a\mathbf{y})(x_b\mathbf{x} + y_b\mathbf{y})| \\ &= x_a x_b|\mathbf{xx}| + x_a y_b|\mathbf{xy}| + y_a x_b|\mathbf{yx}| + y_a y_b|\mathbf{yy}| \\ &= x_a x_b(0) + x_a y_b(+1) + y_a x_b(-1) + y_a y_b(0) \\ &= x_a y_b - y_a x_b.\end{aligned}$$

This simplification uses the fact that $|\mathbf{vv}| = 0$ for any vector $\mathbf{v}$, because the parallelograms would all be collinear with $\mathbf{v}$ and thus without area.

In three dimensions, the determinant of three 3D vectors $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ is denoted $|\mathbf{abc}|$. With Cartesian representations for the vectors, there are analogous rules for parallelepipeds as there are for parallelograms, and we can do an analogous expansion as we did for 2D:

$$\begin{aligned}|\mathbf{abc}| &= |(x_a\mathbf{x} + y_a\mathbf{y} + z_a\mathbf{z})(x_b\mathbf{x} + y_b\mathbf{y} + z_b\mathbf{z})(x_c\mathbf{x} + y_c\mathbf{y} + z_c\mathbf{z})| \\ &= x_a y_b z_c - x_a z_b y_c - y_a x_b z_c + y_a z_b x_c + z_a x_b y_c - z_a y_b x_c.\end{aligned}$$

As you can see, the computation of determinants in this fashion gets uglier as the dimension increases. We will discuss less error-prone ways to compute determinants in Section 5.2.3.

Determinants arise naturally when computing the expression for one vector as a linear combination of two others—for example, if we wish to express a vector $\mathbf{c}$ as a combination of vectors $\mathbf{a}$ and $\mathbf{b}$:

$$\mathbf{c} = a_c\mathbf{a} + b_c\mathbf{b}.$$

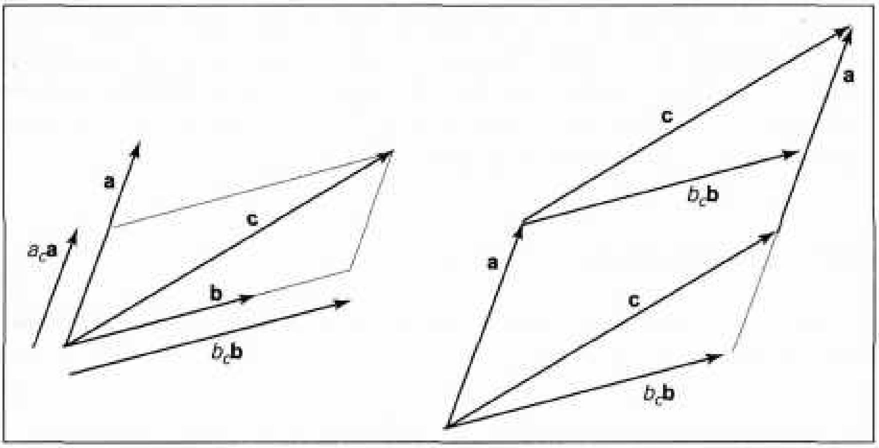


**Figure 5.6.** On the left, the vector **c** can be represented using two basis vectors as $a_c\mathbf{a} + b_c\mathbf{b}$. On the right, we see that the parallelogram formed by **a** and **c** is a sheared version of the parallelogram formed by $b_c$**b** and **a**.

We can see from Figure 5.6 that

$$|(b_c\mathbf{b})\mathbf{a}| = |\mathbf{ca}|,$$

because these parallelograms are just sheared versions of each other. Solving for $b_c$ yields

$$b_c = \frac{|\mathbf{ca}|}{|\mathbf{ba}|}.$$

An analogous argument yields

$$a_c = \frac{|\mathbf{bc}|}{|\mathbf{ba}|}.$$

This is the two-dimensional version of *Cramer's rule* which we will revisit in Section 5.2.5.

## 5.2 Matrices

A matrix is an array of numeric elements that follow certain arithmetic rules. An example of a matrix with two rows and three columns is:

$$\begin{bmatrix} 1.7 & -1.2 & 4.2 \\ 3.0 & 4.5 & -7.2 \end{bmatrix}$$

Matrices are frequently used in computer graphics for a variety of purposes including representation of spatial transforms. For our discussion, we assume the elements of a matrix are all real numbers. This chapter describes both the mechanics of matrix arithmetic and the *determinant* of "square" matrices, i.e., matrices with the same number of rows as columns.

### 5.2.1 Matrix Arithmetic

A matrix times a constant results in a matrix where each element has been multiplied by that constant, e.g.,

$$2\begin{bmatrix} 1 & -4 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} 2 & -8 \\ 6 & 4 \end{bmatrix}.$$

Matrices also add element by element, e.g.,

$$\begin{bmatrix} 1 & -4 \\ 3 & 2 \end{bmatrix} + \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 3 & -2 \\ 5 & 4 \end{bmatrix}.$$

For matrix multiplication, we "multiply" rows of the first matrix with columns of the second matrix:

$$\begin{bmatrix} a_{11} \quad \cdots \quad a_{1m} \\ \vdots \qquad\qquad \vdots \\ \boxed{a_{i1} \quad \cdots \quad a_{im}} \\ \vdots \qquad\qquad \vdots \\ a_{r1} \quad \cdots \quad a_{rm} \end{bmatrix} \left[ \begin{matrix} b_{11} \\ \vdots \\ b_{m1} \end{matrix} \quad \begin{matrix} \cdots \\ \\ \cdots \end{matrix} \quad \boxed{\begin{matrix} b_{1j} \\ \vdots \\ b_{mj} \end{matrix}} \quad \begin{matrix} \cdots \\ \\ \cdots \end{matrix} \quad \begin{matrix} b_{1c} \\ \vdots \\ b_{mc} \end{matrix} \right] = \begin{bmatrix} p_{11} & \cdots & p_{1j} & \cdots & p_{1c} \\ \vdots & & \vdots & & \vdots \\ p_{i1} & \cdots & \boxed{p_{ij}} & \cdots & p_{ic} \\ \vdots & & \vdots & & \vdots \\ p_{r1} & \cdots & p_{rj} & \cdots & p_{rc} \end{bmatrix}$$

So the element $p_{ij}$ of the resulting product is

$$p_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{im}b_{mj}.$$

Note that taking a product of two matrices is only possible if the number of columns of the left matrix is the same as the number of rows of the right matrix. For example,

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} 6 & 7 & 8 & 9 \\ 0 & 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 12 & 17 & 22 & 27 \\ 24 & 33 & 42 & 51 \end{bmatrix}.$$

Matrix multiplication is *not* commutative in most instances:

$$\mathbf{AB} \neq \mathbf{BA}. \tag{5.2}$$

Also, if $\mathbf{AB} = \mathbf{AC}$, it does not necessarily follow that $\mathbf{B} = \mathbf{C}$. Fortunately, matrix multiplication is associative and distributive:

$$\begin{aligned}(\mathbf{AB})\mathbf{C} &= \mathbf{A}(\mathbf{BC}),\\ \mathbf{A}(\mathbf{B}+\mathbf{C}) &= \mathbf{AB}+\mathbf{AC},\\ (\mathbf{A}+\mathbf{B})\mathbf{C} &= \mathbf{AC}+\mathbf{BC}.\end{aligned}$$

In graphics, we use a square matrix to transform a vector represented as a matrix. For example if you have a 2D vector $\mathbf{a} = (x_a, y_a)$ and want to "reflect" it about the $y$ axis to form vector $\mathbf{a}' = (-x_a, y_a)$, you can use a product of a two by two matrix and a two by one matrix, often called a "column vector". The operation in matrix form is:

$$\begin{bmatrix}-1 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}x_a\\ y_a\end{bmatrix} = \begin{bmatrix}-x_a\\ y_a\end{bmatrix}.$$

We can get the same result by "premultiplying" with a "row vector":

$$\begin{bmatrix}x_a & y_a\end{bmatrix}\begin{bmatrix}-1 & 0\\ 0 & 1\end{bmatrix} = \begin{bmatrix}-x_a & y_a\end{bmatrix}.$$

These days, postmultiplication using column vectors is fairly standard, but in many older books and systems you will run across row vectors and premultiplication. The only difference is that the transform matrix must be replaced with its *transpose*. The transpose $\mathbf{A}^T$ of a matrix $A$ is one whose rows are switched with its columns, e.g.,

$$\begin{bmatrix}1 & 2\\ 3 & 4\\ 5 & 6\end{bmatrix}^T = \begin{bmatrix}1 & 3 & 5\\ 2 & 4 & 6\end{bmatrix}.$$

Note that in the previous reflection example, the transpose of the matrix is the same as the matrix. Such matrices are called *symmetric*. Note that the transpose of a product of two matrices obeys:

$$(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T.$$

We would like a matrix analog of the inverse of a real number. We know the inverse of a real number $x$ is $1/x$ and that the product of $x$ and its inverse is 1. We need a matrix $\mathbf{I}$ that we can think of as a "matrix one." This exists only for square matrices and is know as the *identity matrix*; it consists of ones down the "diagonal" and zeroes elsewhere. For example, the four by four identity matrix is

$$\mathbf{I} = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}.$$

The identity matrix is a special case of a *diagonal matrix*, where all non-zero elements are along the diagonal. The diagonal are those elements whose column index equals the row index counting from the upper left. The *inverse matrix* $\mathbf{A}^{-1}$ of a matrix $\mathbf{A}$ is the matrix that ensures $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$. For example,

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^{-1} = \begin{bmatrix} -2.0 & 1.0 \\ 1.5 & -0.5 \end{bmatrix} \quad \text{because} \quad \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} -2.0 & 1.0 \\ 1.5 & -0.5 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Note that the inverse of $\mathbf{A}^{-1}$ is $\mathbf{A}$. So $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$. Also note that the inverse of a product of two matrices is

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}.$$

We will return to inverses later in the chapter.

### 5.2.2 Vector Operations in Matrix Form

We can use matrix formalism to encode vector operations for vectors when using Cartesian coordinates; if we consider the result of the dot product a one by one matrix, it can be written:

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T \mathbf{b}.$$

For example, if we take two 3D vectors we get:

$$\begin{bmatrix} x_a & y_a & z_a \end{bmatrix} \begin{bmatrix} x_b \\ y_b \\ z_b \end{bmatrix} = \begin{bmatrix} x_a x_b + y_a y_b + z_a z_b \end{bmatrix}.$$

### 5.2.3 Matrices and Determinants

Recall from Section 5.1 that the determinant takes $n$ $n$-dimensional vectors and combines them to get a signed $n$-dimensional volume of the $n$-dimensional parallelepiped defined by the vectors. For example, the determinant in 2D is the area of the parallelogram formed by the vectors. We can use matrices to handle the mechanics of computing determinants.

If we have 2D vectors $\mathbf{r}$ and $\mathbf{s}$, we denote the determinant $|\mathbf{rs}|$; this value is the signed area of the parallelogram formed by the vectors. Suppose we have two 2D vectors with Cartesian coordinates $(a, b)$ and $(A, B)$ (Figure 5.7). The determinant can be written in terms of column vectors or as a shorthand:

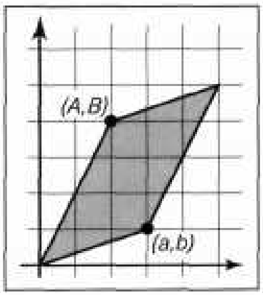


**Figure 5.7.** The 2D determinant in Equation 5.3 is the area of the parallelogram formed by the 2D vectors.

$$\left| \begin{bmatrix} a \\ b \end{bmatrix} \quad \begin{bmatrix} A \\ B \end{bmatrix} \right| \equiv \begin{vmatrix} a & A \\ b & B \end{vmatrix} = aB - Ab. \tag{5.3}$$

Note that the determinant of a matrix is the same as the determinant of its transpose:

$$\begin{vmatrix} a & A \\ b & B \end{vmatrix} = \begin{vmatrix} a & b \\ A & B \end{vmatrix} = aB - Ab.$$

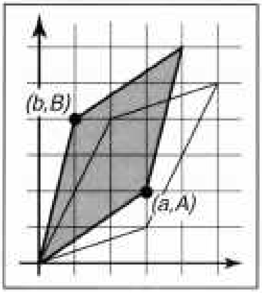


**Figure 5.8.** The sibling parallelogram has the same area as the parallelogram in Figure 5.7.

This means that for any parallelogram in 2D there is a "sibling" parallelogram that has the same area but a different shape (Figure 5.8). For example the parallelogram defined by vectors $(3, 1)$ and $(2, 4)$ has area 10, as does the parallelogram defined by vectors $(3, 2)$ and $(1, 4)$.

The geometric meaning of the 3D determinant is helpful in seeing why certain formulas make sense. For example, the equation of the plane through the points $(x_i, y_i, z_i)$ for $i = 0, 1, 2$ is

$$\begin{vmatrix} x - x_0 & x - x_1 & x - x_2 \\ y - y_0 & y - y_1 & y - y_2 \\ z - z_0 & z - z_1 & z - z_2 \end{vmatrix} = 0.$$

Each column is a vector from point $(x_i, y_i, z_i)$ to point $(x, y, z)$. The volume of the parallelepiped with those vectors as sides is zero only if $(x, y, z)$ is coplanar with the three other points. Almost all equations involving determinants have similarly simple underlying geometry.

As we saw earlier, we can compute determinants by a brute force expansion where most terms are zero, and there is a great deal of bookkeeping on plus and minus signs. The standard way to manage the algebra of computing determinants is to use a form of *Laplace's expansion*. The key part of computing the determinant this way is to find *cofactors* of various matrix elements. Each element of a square matrix has a cofactor which is the determinant of a matrix with one fewer row and column possibly multiplied by minus one. The smaller matrix is obtained by eliminating the row and column that the element in question is in. For example, for a 10 by 10 matrix, the cofactor of $a_{82}$ is the determinant of the 9 by 9 matrix with the 8th row and 2nd column eliminated. The sign of a cofactor is positive if the sum of the row and column indices is even and negative, otherwise. This can be remembered by a checkerboard pattern:

$$\begin{bmatrix} + & - & + & - & \cdots \\ - & + & - & + & \cdots \\ + & - & + & - & \cdots \\ - & + & - & + & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}.$$

So, for a four by four matrix,

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}.$$

The cofactors of the first row are

$$a^c_{11} = \begin{vmatrix} a_{22} & a_{23} & a_{24} \\ a_{32} & a_{33} & a_{34} \\ a_{42} & a_{43} & a_{44} \end{vmatrix}, \quad a^c_{12} = -\begin{vmatrix} a_{21} & a_{23} & a_{24} \\ a_{31} & a_{33} & a_{34} \\ a_{41} & a_{43} & a_{44} \end{vmatrix},$$

$$a^c_{13} = \begin{vmatrix} a_{21} & a_{22} & a_{24} \\ a_{31} & a_{32} & a_{34} \\ a_{41} & a_{42} & a_{44} \end{vmatrix}, \quad a^c_{14} = -\begin{vmatrix} a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{vmatrix}.$$

The determinant of a matrix is found by taking the sum of products of the elements of any row or column with their cofactors. For example, the determinant of the four by four matrix above taken about its second column is

$$|\mathbf{A}| = a_{12}a^c_{12} + a_{22}a^c_{22} + a_{32}a^c_{32} + a_{42}a^c_{42}.$$

We could do a similar expansion about any row or column and they would all yield the same result. Note the recursive nature of this expansion.

A concrete example for the determinant of a particular three by three matrix by expanding the cofactors of the first row is

$$\begin{aligned} \begin{vmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{vmatrix} &= 0\begin{vmatrix} 4 & 5 \\ 7 & 8 \end{vmatrix} - 1\begin{vmatrix} 3 & 5 \\ 6 & 8 \end{vmatrix} + 2\begin{vmatrix} 3 & 4 \\ 6 & 7 \end{vmatrix} \\ &= 0(32 - 35) - 1(24 - 30) + 2(21 - 24) \\ &= 0. \end{aligned}$$

We can deduce that the volume of the parallelepiped formed by the vectors defined by the columns (or rows since the determinant of the transpose is the same) is zero. This is equivalent to saying that the columns (or rows) are not linearly independent. Note that the sum of the first and third rows is twice the second row, which implies linear dependence.

### 5.2.4 Computing Inverses

Determinants give us a tool to compute the inverse of a matrix. It is a very inefficient method for large matrices, but often in graphics our matrices are small. A key to developing this method is that the determinant of a matrix with two identical rows is zero. This should be clear because the volume of the $n$-dimensional parallelepiped is zero if two of its sides are the same. Suppose we have a four by four matrix $\mathbf{A}$ and we wish to find its inverse $\mathbf{A}^{-1}$. The inverse is

$$\mathbf{A}^{-1} = \frac{1}{|\mathbf{A}|}\begin{bmatrix} a^c_{11} & a^c_{21} & a^c_{31} & a^c_{41} \\ a^c_{12} & a^c_{22} & a^c_{32} & a^c_{42} \\ a^c_{13} & a^c_{23} & a^c_{33} & a^c_{43} \\ a^c_{14} & a^c_{24} & a^c_{34} & a^c_{44} \end{bmatrix}.$$

Note that this is just the transpose of the matrix where elements of $\mathbf{A}$ are replaced by their respective cofactors multiplied by the leading constant (1 or -1). This matrix is called the *adjoint* of $\mathbf{A}$. The adjoint is the transpose of the *cofactor* matrix of $\mathbf{A}$. We can see why this is an inverse. Look at the product $\mathbf{A}\mathbf{A}^{-1}$ which we expect to be the identity. If we multiply the first row of $\mathbf{A}$ by the first column of the adjoint matrix we need to get $|\mathbf{A}|$ (remember the leading constant above divides by $|\mathbf{A}|$:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}\begin{bmatrix} a^c_{11} & \cdot & \cdot & \cdot \\ a^c_{12} & \cdot & \cdot & \cdot \\ a^c_{13} & \cdot & \cdot & \cdot \\ a^c_{14} & \cdot & \cdot & \cdot \end{bmatrix} = \begin{bmatrix} |\mathbf{A}| & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}.$$

This is true because the elements in the first row of $\mathbf{A}$ are multiplied exactly by their cofactors in the first column of the adjoint matrix which is exactly the determinant. The other values along the diagonal of the resulting matrix are $|\mathbf{A}|$ for analogous reasons. The zeros follow a similar logic:

$$\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}\begin{bmatrix} a^c_{11} & \cdot & \cdot & \cdot \\ a^c_{12} & \cdot & \cdot & \cdot \\ a^c_{13} & \cdot & \cdot & \cdot \\ a^c_{14} & \cdot & \cdot & \cdot \end{bmatrix} = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}.$$

Note that this product is a determinant of *some* matrix:

$$a_{21}a^c_{11} + a_{22}a^c_{12} + a_{23}a^c_{13} + a_{24}a^c_{14}.$$

The matrix in fact is

$$\begin{bmatrix} a_{21} & a_{22} & a_{23} & a_{24} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44}. \end{bmatrix}.$$

Because the first two rows are identical, the matrix is singular, and thus, its determinant is zero.

The argument above does not apply just to four by four matrices; using that size just simplifies typography. For any matrix, the inverse is the adjoint matrix divided by the determinant of the matrix being inverted. The adjoint is the transpose of the cofactor matrix, which is just the matrix whose elements have been replaced by their cofactors. For example, the inverse of a three by three matrix whose determinant is 6 is

$$\begin{bmatrix} 1 & 1 & 2 \\ 1 & 3 & 4 \\ 0 & 2 & 5 \end{bmatrix}^{-1} = \frac{1}{6}\begin{bmatrix} \begin{vmatrix} 3 & 4 \\ 2 & 5 \end{vmatrix} & -\begin{vmatrix} 1 & 2 \\ 2 & 5 \end{vmatrix} & \begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix} \\ -\begin{vmatrix} 1 & 4 \\ 0 & 5 \end{vmatrix} & \begin{vmatrix} 1 & 2 \\ 0 & 5 \end{vmatrix} & -\begin{vmatrix} 1 & 2 \\ 1 & 4 \end{vmatrix} \\ \begin{vmatrix} 1 & 3 \\ 0 & 2 \end{vmatrix} & -\begin{vmatrix} 1 & 1 \\ 0 & 2 \end{vmatrix} & \begin{vmatrix} 1 & 1 \\ 1 & 3 \end{vmatrix} \end{bmatrix}$$

$$= \frac{1}{6}\begin{bmatrix} 7 & -1 & -2 \\ -5 & 5 & -2 \\ 2 & -2 & 2 \end{bmatrix}.$$

You can check this yourself by multiplying the matrices and making sure you get the identity.

### 5.2.5 Linear Systems

We often encounter linear systems in graphics with "$n$ equations and $n$ unknowns," for example ($n = 3$),

$$\begin{aligned} 3x + 7y + 2z &= 4, \\ 2x - 4y - 3z &= -1, \\ 5x + 2y + z &= 1. \end{aligned}$$

Here $x$, $y$ and $z$ are the "unknowns" for which we wish to solve. We can write this in matrix form:

$$\begin{bmatrix} 3 & 7 & 2 \\ 2 & -4 & -3 \\ 5 & 2 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ -1 \\ 1 \end{bmatrix}.$$

A common shorthand for such systems is $\mathbf{Ax} = \mathbf{b}$ where it is assumed that $\mathbf{A}$ is a square matrix with known constants, $\mathbf{x}$ is an unknown column vector (with elements $x$, $y$, and $z$ in our example), and $\mathbf{b}$ is a column matrix of known constants.

There are many ways to solve such systems, but for small systems we will use *Cramer's rule* as we saw earlier in 2D from a geometric standpoint. Here, we show this algebraically. The solution to the above equation is

$$x = \frac{\begin{vmatrix} 4 & 7 & 2 \\ -1 & -4 & -3 \\ 1 & 2 & 1 \end{vmatrix}}{\begin{vmatrix} 3 & 7 & 2 \\ 2 & -4 & -3 \\ 5 & 2 & 1 \end{vmatrix}} \quad y = \frac{\begin{vmatrix} 3 & 4 & 2 \\ 2 & -1 & -3 \\ 5 & 1 & 1 \end{vmatrix}}{\begin{vmatrix} 3 & 7 & 2 \\ 2 & -4 & -3 \\ 5 & 2 & 1 \end{vmatrix}} \quad z = \frac{\begin{vmatrix} 3 & 7 & 4 \\ 2 & -4 & -1 \\ 5 & 2 & 1 \end{vmatrix}}{\begin{vmatrix} 3 & 7 & 2 \\ 2 & -4 & -3 \\ 5 & 2 & 1 \end{vmatrix}}$$

The rule here is to take a ratio of determinants, where the denominator is $|\mathbf{A}|$ and the numerator is the determinant of a matrix created by replacing a column of $\mathbf{A}$ with the column vector $\mathbf{b}$. The column replaced corresponds to the position of the unknown in vector $\mathbf{x}$. For example, $y$ is the second unknown and the second column is replaced. Note that if $|\mathbf{A}| = 0$, the division is undefined and there is no solution. This is just another version of the rule that if $\mathbf{A}$ is singular (zero determinant) then there is no unique solution to the equations.

### 5.2.6 Eigenvalues and Matrix Diagonalization

Square matrices have *eigenvalues* and *eigenvectors* associated with them. The eigenvectors are those *non-zero* vectors whose directions do not change when multiplied by the matrix. For example, suppose for a matrix $\mathbf{A}$ and vector $\mathbf{a}$, we have

$$\mathbf{Aa} = \lambda\mathbf{a}. \tag{5.4}$$

This means we have stretched or compressed $\mathbf{a}$, but its direction has not changed. The scalefactor $\lambda$ is called the eigenvalue associated with eigenvector $\mathbf{a}$. Knowing the eigenvalues and eigenvectors of matrices is helpful in a variety of practical applications. We will describe them to gain insight into geometric transformation matrices, and as a step toward singular values and vectors described in the next section.

If we assume a matrix has at least one eigenvector, then we can do a standard manipulation to find it. First, we make both sides the product of a square matrix and a vector:

$$\mathbf{Aa} = \lambda\mathbf{Ia}, \tag{5.5}$$

where $\mathbf{I}$ is an identity matrix. This can be rewritten

$$\mathbf{Aa} - \lambda\mathbf{Ia} = 0. \tag{5.6}$$

Because matrix multiplication is distributive, we can group the matrices:

$$(\mathbf{A} - \lambda\mathbf{I})\,\mathbf{a} = 0. \tag{5.7}$$

This equation can only be true if the matrix $(\mathbf{A} - \lambda\mathbf{I})$ is singular, and thus its determinant is zero. The elements in this matrix are the numbers in $\mathbf{A}$ except along the diagonal. For example, for a 2 by 2 matrix the eigenvalues obey

$$\begin{vmatrix} a_{11} - \lambda & a_{12} \\ a_{21} & a_{22} - \lambda \end{vmatrix} = \lambda^2 - (a_{11} + a_{22})\lambda + (a_{11}a_{22} - a_{12}a_{21}) = 0. \tag{5.8}$$

Because this is a quadratic equation, we know there are exactly two solutions for $\lambda$. These solutions may or may not be unique or real. A similar manipulation for an $n$ by $n$ matrix will yield an $n$th degree polynomial in $\lambda$. Because it is not possible, in general, to find exact explicit solutions of polynomial equations of degree greater than four, we can only be guaranteed to find eigenvalues of matrices 4 by 4 or smaller by analytic methods. For larger matrices, numerical methods are the only option.

An important special case is eigenvalues of symmetric matrices (where $\mathbf{A} = \mathbf{A}^T$). Here, it is known that the eigenvalues are real. If they are also distinct, their eigenvectors are mutually orthogonal. Such matrices can be put into *diagonal form*:

$$\mathbf{A} = \mathbf{RDR}^T, \tag{5.9}$$

where $\mathbf{R}$ is an orthogonal matrix and $\mathbf{D}$ is a diagonal matrix. Recall that an orthogonal matrix might be better called an orthonormal matrix; its columns are mutually orthogonal and the sum of the squares of the elements of each column are one. The columns of $\mathbf{R}$ are the eigenvectors of $\mathbf{A}$ and the non-zero elements of $\mathbf{D}$ are the eigenvalues of $\mathbf{A}$. For example, given the matrix

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix},$$

the eigenvalues of $\mathbf{A}$ are the solutions to

$$\lambda^2 - 3\lambda + 1 = 0.$$

We approximate the exact values for compactness of notation:

$$\lambda = \frac{3 \pm \sqrt{5}}{2}, \;\approx\; \begin{bmatrix} 2.618 \\ 0.382 \end{bmatrix}.$$

Now we can find the associated eigenvector. The first is the nontrivial (not $x = y = 0$) solution to the homogeneous equation,

$$\begin{bmatrix} 2 - 2.618 & 1 \\ 1 & 1 - 2.618 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

This is approximately $(x, y) = (0.8507, 0.5257)$. Note that there are infinitely many solutions parallel to that 2D vector, and we just picked the one of unit length. Similarly the eigenvector associated with $\lambda_2$ is $(x, y) = (-0.5257, 0.8507)$. This means the diagonal form of $\mathbf{A}$ is (within some precision due to our numeric approximation):

$$\begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 0.8507 & -0.5257 \\ 0.5257 & 0.8507 \end{bmatrix} \begin{bmatrix} 2.618 & 0 \\ 0 & 0.382 \end{bmatrix} \begin{bmatrix} 0.8507 & 0.5257 \\ -0.5257 & 0.8507 \end{bmatrix}.$$

We will revisit the geometry of this matrix as a transform in the next chapter.

### 5.2.7 Singular Value Decomposition

We saw in the last section that any symmetric matrix can be "diagonalized". However, most matrices we encounter in graphics are not symmetric. Fortunately, these matrices can be decomposed using *singular value decomposition* (SVD). We take the matrix $\mathbf{M}$ and represent it as

$$\mathbf{M} = \mathbf{USV},$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal and $\mathbf{S}$ is diagonal. The diagonal elements of $\mathbf{S}$ are the *singular values* of $\mathbf{M}$. There is a standard trick to computing the SVD. First we define $\mathbf{A} = \mathbf{MM}^T$. We assume that we can perform a SVD on $\mathbf{M}$:

$$\mathbf{A} = \mathbf{MM}^T = (\mathbf{USV})(\mathbf{USV})^T = \mathbf{USVV}^T\mathbf{SU}^T = \mathbf{US}^2\mathbf{U}^T.$$

The substitution is based on the fact that $(\mathbf{BC})^T = \mathbf{C}^T\mathbf{B}^T$, that the transpose of an orthogonal matrix is its inverse, and the transpose of a diagonal matrix is the matrix itself. The beauty of this new form is that $\mathbf{A}$ is symmetric and $\mathbf{US}^2\mathbf{U}^T$ is its eigenvalue diagonal decomposition, where $\mathbf{S}^2$ contains the eigenvalues. Thus, we find that the singular values of a matrix are the square roots of the eigenvalues of that matrix times its transpose. We now make this concrete with an example:

$$\mathbf{M} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad \mathbf{A} = \mathbf{MM}^T = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}.$$

We saw the eigenvalue decomposition for this matrix in the previous section. We observe immediately

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.8507 & -0.5257 \\ 0.5257 & 0.8507 \end{bmatrix} \begin{bmatrix} \sqrt{2.618} & 0 \\ 0 & \sqrt{0.382} \end{bmatrix} \mathbf{V}.$$

We can solve for $\mathbf{V}$ algebraically:

$$\mathbf{V} = \mathbf{S}^{-1}\mathbf{U}^T\mathbf{M}.$$

The inverse of $\mathbf{S}$ is a diagonal matrix with the reciprocals of the diagonal elements of $\mathbf{S}$. This yields

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} = \mathbf{U}\begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix}\mathbf{V}$$

$$= \begin{bmatrix} 0.8507 & -0.5257 \\ 0.5257 & 0.8507 \end{bmatrix}\begin{bmatrix} 1.618 & 0 \\ 0 & 0.618 \end{bmatrix}\begin{bmatrix} 0.5257 & 0.8507 \\ -0.8507 & 0.5257 \end{bmatrix}.$$

This form used the standard symbol $\sigma$ for a singular value. Note that for a symmetric matrix, the eigenvalues and the singular values are the same ($\sigma_i = \lambda_i$). Also note that the eigenvalue diagonalization and the SVD are the same. We will examine the geometry of SVD more in Section 6.1.6.

## Frequently Asked Questions

• Why is matrix multiplication defined the way it is rather than just element by element?

Element by element multiplication is a perfectly good way to define matrix multiplication, and indeed it has nice properties. However, in practice it is not very useful. Ultimately most matrices are used to transform column vectors, e.g., in 3D you might have

$$\mathbf{b} = \mathbf{M}\mathbf{a},$$

where $\mathbf{a}$ and $\mathbf{b}$ are vectors and $\mathbf{M}$ is a three by three matrix. To allow geometric operations such as rotation, combinations of all three elements of $\mathbf{a}$ must go into each element of $\mathbf{b}$. That requires us to either go row-by-row or column-by-column through $\mathbf{M}$. That choice is made based on composition of matrices having the desired property,

$$\mathbf{M}_2(\mathbf{M}_1\mathbf{a}) = (\mathbf{M}_2\mathbf{M}_1)\mathbf{a}$$

which allows us to use one composite matrix $\mathbf{C} = \mathbf{M}_2\mathbf{M}_1$ to transform our vector. This is valuable when many vectors will be transformed by the same composite matrix. So, in summary, the somewhat weird rule for matrix multiplication is engineered to have these desired properties.

**• Sometimes I hear that eigenvalues and singular values are the same thing and sometimes that one is the square of the other. Which is right?**

If a real matrix $\mathbf{M}$ is symmetric, then its eigenvalues and singular values are the same. If $\mathbf{M}$ is not symmetric, the matrix $\mathbf{A} = \mathbf{M}\mathbf{M}^T$ is symmetric and has real eignenvalues. The singular values of $\mathbf{M}$ and $\mathbf{M}^T$ are the same and are the square roots of the singular/eigenvalues of $\mathbf{A}$. Thus, when the square root statement is made, it is because two different matrices (with a very particular relationship) are being talked about: $\mathbf{A} = \mathbf{M}\mathbf{M}^T$.

## Notes

The discussion of determinants as volumes is based on *A Vector Space Approach to Geometry* (Hausner, 1998). Hausner has an excellent discussion of vector analysis and the fundamentals of geometry as well. The geometric derivation of Cramer's rule in 2D is taken from *Practical Linear Algebra: A Geometry Toolbox* (Farin & Hansford, 2004). That book also has geometric interpretations of other linear algebra operations such as Gaussian elimination. The discussion of eigenvalues and singular values is based primarily on *Linear Algebra and its Applications* (Strang, 1988). The example of SVD of the shear matrix is based on a discussion in *Computer Graphics and Geometric Modeling* (Salomon, 1999).

## Exercises

1. Write an implicit equation for the 2D line through points $(x_0, y_0)$ and $(x_1, y_1)$ using a 2D determinant.

2. Show that if the rows of a matrix are orthonormal, then so are the columns.

3. Show that the eigenvalues of a diagonal matrix are its diagonal elements.

4. Show that for a square matrix $\mathbf{A}$, $\mathbf{A}\mathbf{A}^T$ is a symmetric matrix.

5. Show that for three 3D vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}$, the following identity holds: $|\mathbf{abc}| = (\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c}$.

6. Explain why the volume of the tetrahedron with side vectors $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ (see Figure 5.2) is given by $|\mathbf{abc}|/6$.

7. Given the $(x, y)$ coordinates of the three vertices of a 2D triangle, explain why the area is given by

$$\frac{1}{2}\begin{vmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ 1 & 1 & 1 \end{vmatrix}.$$

# 6

# Transformation Matrices

In this chapter, we describe how we can use matrix multiplications to accomplish changes in a vector such as scaling, rotation, and translation. We also discuss how these transforms operate differently on locations (points), displacement vectors, and surface normal vectors.

We will show how a set of points transforms if the points are represented as offset vectors from the origin. So think of the image we shall use (a clock) as a bunch of points that are the ends of vectors whose tails are at the origin.

## 6.1 Basic 2D Transforms

We can use matrices to change the components of a 2D vector. For example:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_{11}x + a_{12}y \\ a_{21}x + a_{22}y \end{bmatrix}.$$

Such a transformation can change vectors in a variety of ways that are useful. In particular, it can be used to scale, rotate, and shear. We will introduce more general transformations later, but the basics of transformation are embodied in the simple formula above. For our purposes, consider moving along the $x$-axis a horizontal move, and along the $y$-axis, a vertical move.

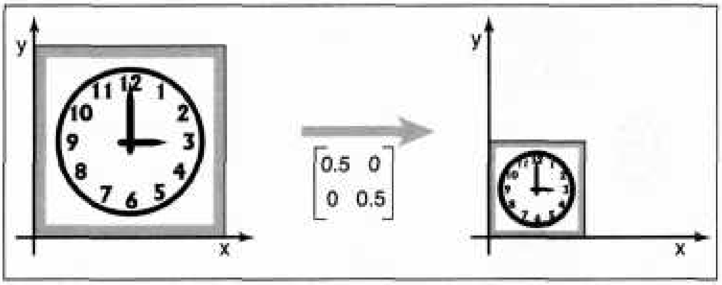


**Figure 6.1.** *Scaling uniformly by half for each axis: The scale matrix has the proportion of change in each of the diagonal elements and zeroes in the off-diagonal elements.*

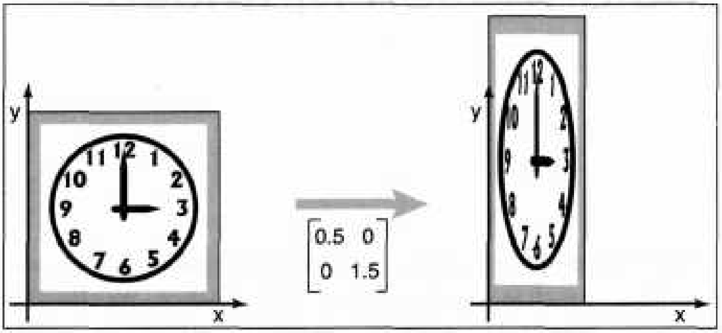


**Figure 6.2.** Scaling non-uniformly in *x* and *y*: The scaling matrix is diagonal with non-equal elements. Note that the square outline of the clock becomes a rectangle and the circular face becomes an ellipse.

### 6.1.1 Scaling

The most basic transform is a *scale*. This transform can change length and possibly direction:

$$\text{scale}(s_x, s_y) = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}.$$

Note what this matrix does to a vector with Cartesian components $(x, y)$:

$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \end{bmatrix}.$$

For example, the matrix that shrinks $x$ and $y$ uniformly by a factor of two is (Figure 6.1)

$$\text{scale}(0.5, 0.5) = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}.$$

A matrix which halves in the horizontal and increases to three-halves in the vertical is (see Figure 6.2)

$$\text{scale}(0.5, 1.5) = \begin{bmatrix} 0.5 & 0 \\ 0 & 1.5 \end{bmatrix}.$$

### 6.1.2 Shearing

A shear is something that pushes things sideways, producing something like a deck of cards across which you push your hand; the bottom card stays put and cards move more the closer they are to the top of the deck. The horizontal and vertical shear matrices are

$$\text{shear-x}(s) = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix}, \quad \text{shear-y}(s) = \begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix}.$$

For example, the transform that shears horizontally so that vertical lines become 45° lines leaning towards the right is (see Figure 6.3)

$$\text{shear-x}(1) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

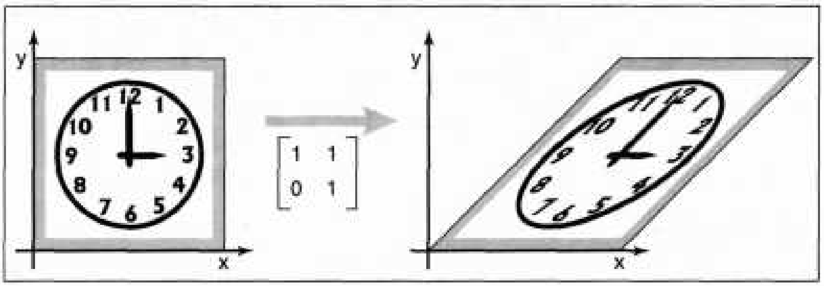


**Figure 6.3.** *An x shear matrix moves points to the right in proportion to their y-coordinate. Now the square outline of the clock becomes a parallelogram and, as with scaling, the circular face of the clock becomes an ellipse.*

An analogous transform vertically is (see Figure 6.4)

$$\text{shear-y}(1) = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

Note that the square outline of the sheared clock becomes a parallelogram. Also, note that the circular face of the sheared clock looks like it could be an ellipse. In fact it is, as will be evident when we discuss SVD.

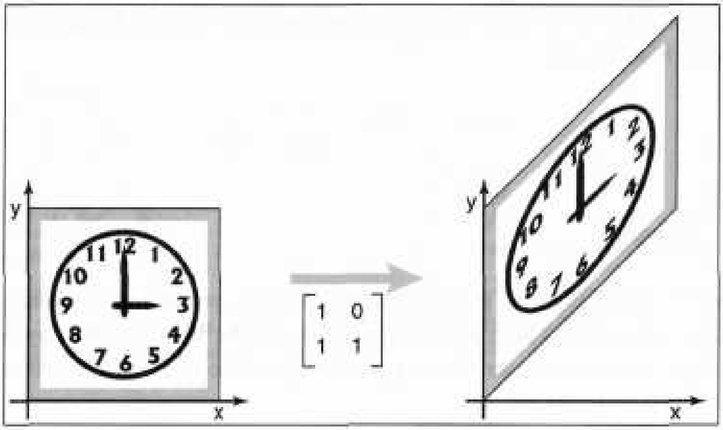


**Figure 6.4.** A *y* shear matrix moves points up in proportion to their *x*-coordinate.

Another way to think of a shear is in terms of rotation of only the vertical (or horizontal) axes. The shear transform that takes a vertical axis and tilts it clockwise by an angle $\phi$ is

$$\begin{bmatrix} 1 & \tan\phi \\ 0 & 1 \end{bmatrix}.$$

Similarly, the shear matrix which rotates the horizontal axis counterclockwise by angle $\phi$ is

$$\begin{bmatrix} 1 & 0 \\ \tan\phi & 1 \end{bmatrix}.$$

### 6.1.3 Rotation

Suppose we want to rotate a vector by an angle $\phi$ counterclockwise. First, suppose we have a vector $\mathbf{a} = (x_a, y_a)$, and we want to rotate it by an angle $\phi$ to get to vector $\mathbf{b} = (x_b, y_b)$. If the vector $\mathbf{a}$ makes an angle $\alpha$ with the $x$-axis, and its length is $r = x_a^2 + y_a^2$, then we know that by definition,

$$x_a = r\cos\alpha,$$
$$y_a = r\sin\alpha.$$

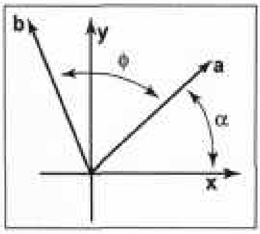


**Figure 6.5.** The geometry for Equation 6.1.

Because $\mathbf{b}$ is a rotation of $\mathbf{a}$, it also has length $r$. Because it is rotated an angle $\phi$ from $\mathbf{a}$, the angle $\mathbf{b}$ makes with the $x$-axis is $(\alpha + \phi)$ (Figure 6.5). From basic trigonometry we know that

$$\begin{aligned} x_b &= r\cos(\alpha+\phi) = r\cos\alpha\cos\phi - r\sin\alpha\sin\phi, \\ y_b &= r\sin(\alpha+\phi) = r\sin\alpha\cos\phi + r\cos\alpha\sin\phi. \end{aligned} \tag{6.1}$$

Substituting the components in $x_a = r\cos\alpha$ and $y_a = r\sin\alpha$ gives

$$\begin{aligned} x_b &= x_a\cos\phi - y_a\sin\phi, \\ y_b &= y_a\cos\phi + x_a\sin\phi. \end{aligned}$$

In matrix form, the equivalent transformation that takes **a** to **b** is

$$\text{rotate}(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix}.$$

For example, a matrix that rotates vectors by $\pi/4$ radians (45 degrees) is (see Figure 6.6)

$$\begin{bmatrix} \cos\frac{\pi}{4} & -\sin\frac{\pi}{4} \\ \sin\frac{\pi}{4} & \cos\frac{\pi}{4} \end{bmatrix} = \begin{bmatrix} 0.707 & -0.707 \\ 0.707 & 0.707 \end{bmatrix}.$$

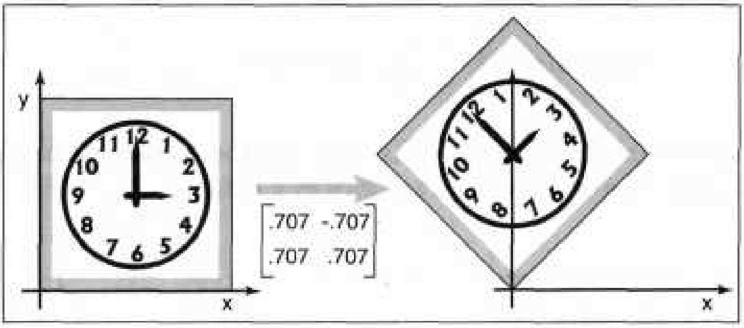


**Figure 6.6.** A rotation by 45 degrees. Note that the rotation is counterclockwise and that cos(45°) = sin(45°) ≈ .707.

A matrix which rotates by $\pi/6$ radians (30 degrees) in the *clockwise* direction is a rotation by $-\pi/6$ radians in our framework (see Figure 6.7):

$$\begin{bmatrix} \cos\frac{-\pi}{6} & -\sin\frac{-\pi}{6} \\ \sin\frac{-\pi}{6} & \cos\frac{-\pi}{6} \end{bmatrix} = \begin{bmatrix} 0.866 & 0.5 \\ -0.5 & 0.866 \end{bmatrix}.$$

Because the norm of each row of a rotation matrix is one ($\sin^2\phi + \cos^2\phi = 1$), and the rows are orthogonal ($\cos\phi(-\sin\phi) + \sin\phi\cos\phi = 0$), we see that rotation matrices are orthogonal matrices (i.e., orthogonal rows each of length one).

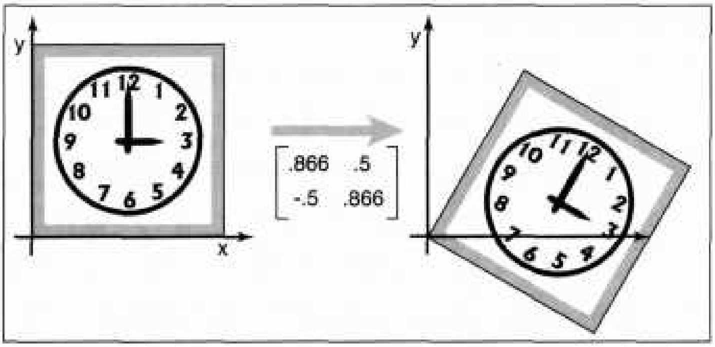


**Figure 6.7.** A rotation by minus thirty degrees. Note that the rotation is clockwise and that cos(-30°) ≈ .866 and sin(-30°) = -.5.

### 6.1.4 Reflection

We can reflect a vector around either of the coordinate axes (see Figures 6.8 and 6.9):

$$\text{reflect-y}(s) = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \text{reflect-x}(s) = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

While one might expect that the matrix with $-1$ in both elements of the diagonal is also a reflection, in fact it is just a rotation by $\pi$ radians.

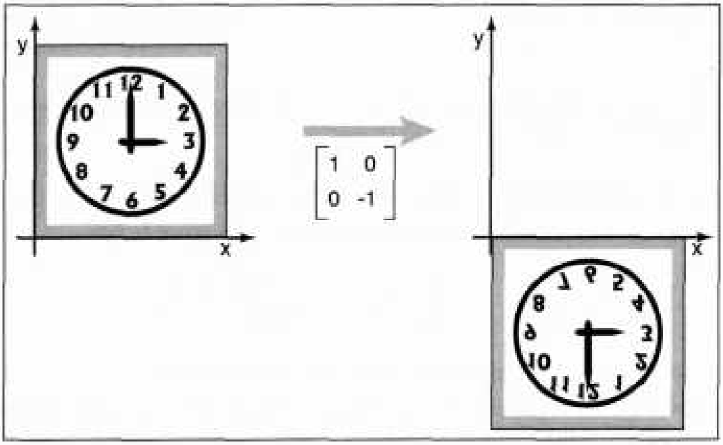


**Figure 6.8.** A reflection about the *x*-axis is achieved by multiplying all *y*-coordinates by -1.

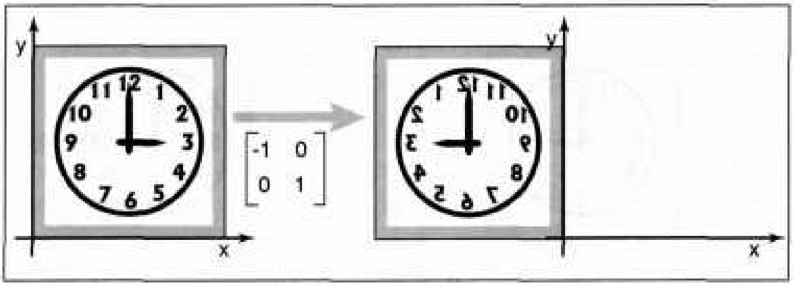


Figure 6.9. A reflection about the *y*-axis is achieved by multiplying all *x*-coordinates by -1.

### 6.1.5 Composition of 2D Transforms

It is common for graphics programs to apply more than one transformation to an object. For example, we might want to first apply a scale $\mathbf{S}$, and then a rotation $\mathbf{R}$. This would be done in two steps on a 2D vector $\mathbf{v}_1$:

$$\text{first,}\mathbf{v}_2 = \mathbf{S}\mathbf{v}_1, \text{ then,}\mathbf{v}_3 = \mathbf{R}\mathbf{v}_2.$$

Another way to write this is

$$\mathbf{v}_3 = \mathbf{R}\left(\mathbf{S}\mathbf{v}_1\right).$$

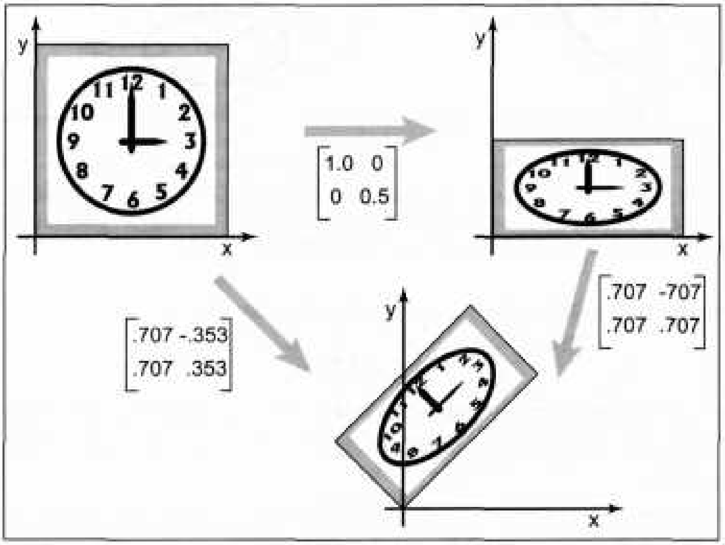


Figure 6.10. Applying the two transform matrices in sequence is the same as applying the product of those matrices once. This is a key concept that underlies most graphics hardware and software.

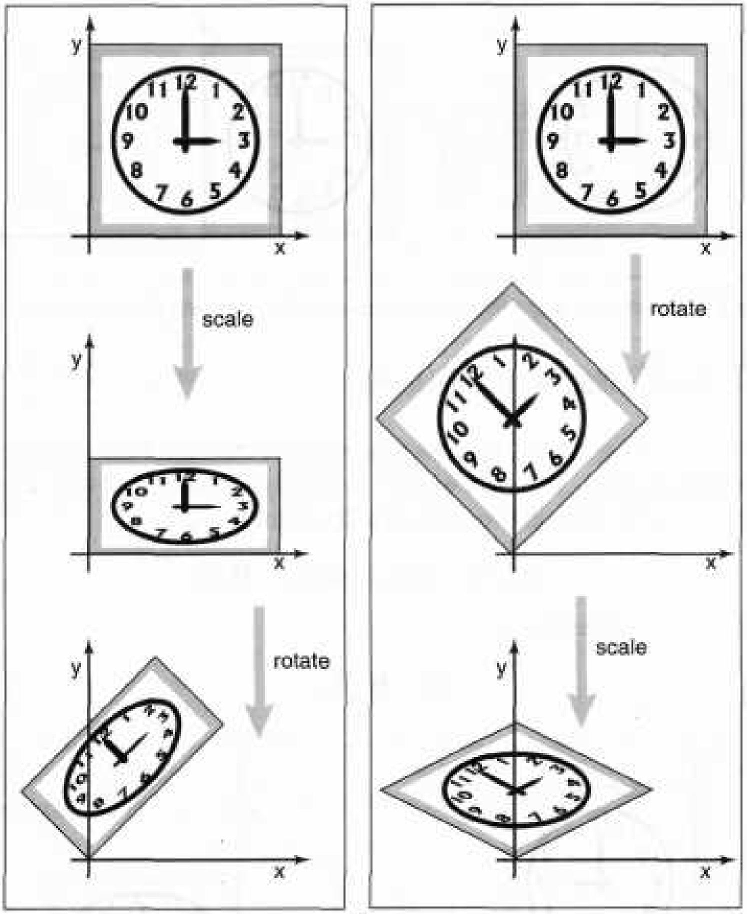


**Figure 6.11.** The order in which two transforms are applied is usually important. In this example, we do a scale by one-half in *y* and then rotate by 45°. Reversing the order in which these two transforms are applied yields a different result.

Because matrix multiplication is associative, we can also write

$$\mathbf{v}_3 = (\mathbf{RS})\,\mathbf{v}_1.$$

In other words, we can represent the effects of transforms by two matrices in a single matrix of the same size by multiplying the two matrices: $\mathbf{M} = \mathbf{RS}$ (Figure 6.10).

It is *very important* to remember that these transforms are applied from the *right side first*. So the matrix $\mathbf{M} = \mathbf{RS}$ first applies $\mathbf{S}$ and then $\mathbf{R}$.

As an example, suppose we want to scale by one-half in the vertical direction and then rotate by $\pi/4$ radians (45 degrees). The resulting matrix is

$$\begin{bmatrix} 0.707 & -0.707 \\ 0.707 & 0.707 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0.5 \end{bmatrix} = \begin{bmatrix} 0.707 & -0.353 \\ 0.707 & 0.353 \end{bmatrix}.$$

It is important to always remember that matrix multiplication is not commutative. So the order of transforms *does* matter. For example, scaling then rotating is usually different than rotating then scaling (see Figure 6.11).

### 6.1.6 Decomposition of 2D Transforms

For any given transformation matrix $\mathbf{M}$, we can decompose it into various matrix products $\mathbf{M} = \mathbf{M}_1\mathbf{M}_2$, $\mathbf{M} = \mathbf{M}_3\mathbf{M}_4\mathbf{M}_5$, and so on.

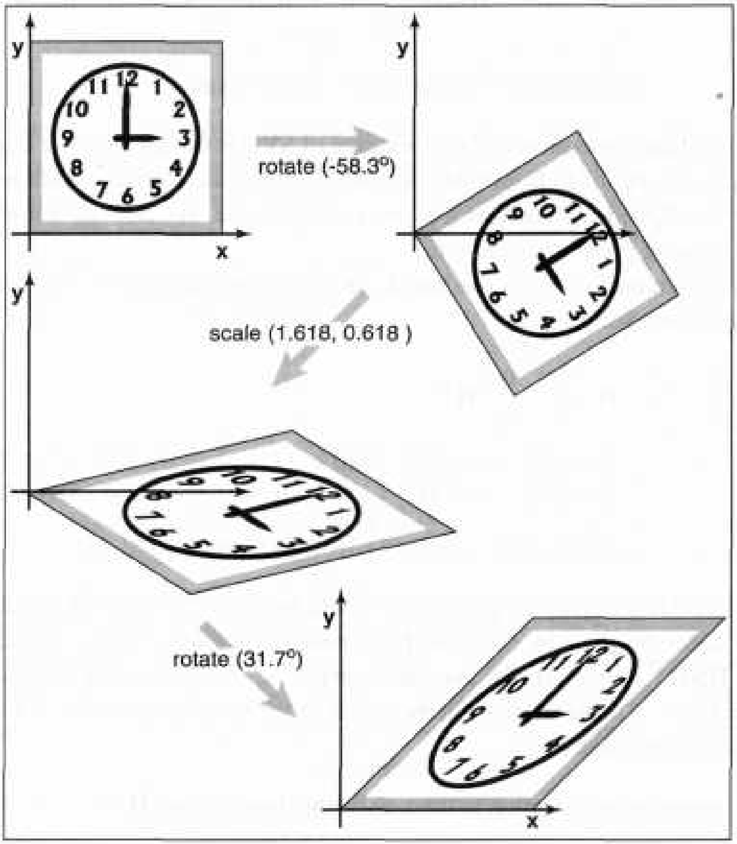


**Figure 6.12.** Singular Value Decomposition (SVD) for a shear matrix. Any 2D matrix can be decomposed into a product of rotation, scale, rotation. Note that the circular face of the clock must become an ellipse because it is just a rotated and scaled circle.

An interesting result is that any 2D transform can be decomposed into the product of a rotation, a scale, and a rotation, with the caveat that the scale may have minus signs in it (i.e., it may include a reflection). This observation follows from the existence of the *singular value decomposition* (SVD) discussed in Section 5.2.7. As shown there, a matrix is a product $\mathbf{M} = \mathbf{R}_2\mathbf{S}\mathbf{R}_1$, where $\mathbf{R}_1$ and $\mathbf{R}_2$ are rotation matrices, and $\mathbf{S}$ is a scale matrix. The example used in Section 5.2.7 is in fact a shear matrix (Figure 6.12):

$$\begin{aligned}\begin{bmatrix}1 & 1\\ 0 & 1\end{bmatrix} &= \mathbf{R}_2\begin{bmatrix}\sigma_1 & 0\\ 0 & \sigma_2\end{bmatrix}\mathbf{R}_1\\ &= \begin{bmatrix}0.8507 & -0.5257\\ 0.5257 & 0.8507\end{bmatrix}\begin{bmatrix}1.618 & 0\\ 0 & 0.618\end{bmatrix}\begin{bmatrix}0.5257 & 0.8507\\ -0.8507 & 0.5257\end{bmatrix}\\ &= \text{rotate}\,(31.7^\circ)\ \text{scale}\,(1.618, 0.618)\ \text{rotate}\,(-58.3^\circ).\end{aligned}$$

An immediate consequence of the existence of SVD is that all 2D basic transform matrices can be made from rotation matrices and scale matrices (with negative elements allowed). Shear matrices are a convenience, but they are not required for expressing potential transforms.

For a symmetric matrix, we can do an eigenvalue decomposition. Recall the example from Section 5.2.6:

$$\begin{aligned}\begin{bmatrix}2 & 1\\ 1 & 1\end{bmatrix} &= \mathbf{R}\begin{bmatrix}\lambda_1 & 0\\ 0 & \lambda_2\end{bmatrix}\mathbf{R}^T\\ &= \begin{bmatrix}0.8507 & -0.5257\\ 0.5257 & 0.8507\end{bmatrix}\begin{bmatrix}2.618 & 0\\ 0 & 0.382\end{bmatrix}\begin{bmatrix}0.8507 & 0.5257\\ -0.5257 & 0.8507\end{bmatrix}\\ &= \text{rotate}\,(31.7^\circ)\ \text{scale}\,(2.618, 0.382)\ \text{rotate}\,(-31.7^\circ).\end{aligned}$$

Note that this is exactly the product we would get if we did SVD; SVD and eigenvalue decomposition are the same for symmetric matrices. Again, the diagonal form $\mathbf{RSR}^T$ only exists for symmetric matrices, so SVD is certainly the more general tool. The geometric interpretation of the transformation by a diagonalized symmetric matrix is:

1. Rotate some direction to the $x$-axis (the transform by $\mathbf{R}^T$);

2. Scale in $x$ and $y$ by $(\lambda_1, \lambda_2)$ (the transform by $\mathbf{S}$);

3. Rotate the $y$-axis back to the original direction (the transform by $\mathbf{R}$).

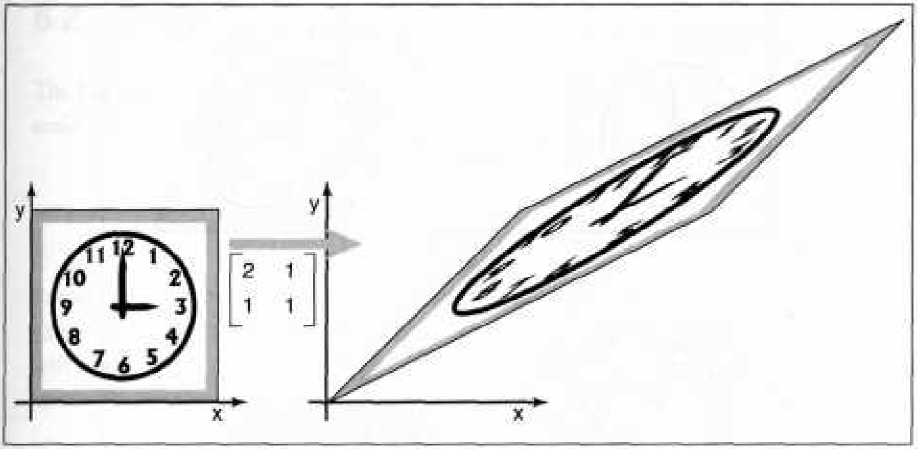


**Figure 6.13.** *A symmetric matrix is always a scale along some axis. In this case it is along the $\phi$ = 31.7° direction which means the real eigenvector for this matrix is in that direction.*

These three transforms together have the effect of a scale in an arbitrary direction (Figure 6.13). For example, the matrix above, according to its eigenvalue decomposition, scales in a direction 31.7° counterclockwise from three o'clock (the $x$-axis). This is a touch before 2 p.m. on the clockface as is confirmed by the figure. We can also reverse the diagonalization process; to scale by $(\lambda_1, \lambda_2)$ with the scaling by $\lambda_1$ in angle $\phi$ we have

$$\begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix} = \begin{bmatrix} \lambda_1 \cos^2\phi + \lambda_2 \sin^2\phi & (\lambda_2 - \lambda_1)\cos\phi\sin\phi \\ (\lambda_2 - \lambda_1)\cos\phi\sin\phi & \lambda_2\cos^2\phi + \lambda_1\sin^2\phi \end{bmatrix}.$$

We should take heart that this is a symmetric matrix as we know must be true since we constructed it assuming an eigenvalue diagonalization was possible.

In summary, every matrix can be decomposed via SVD into a rotation times a scale times another rotation. Only symmetric matrices can be decomposed via eigenvalue diagonalization into a rotation times a scale times the inverse-rotation, and such matrices are a simple scale in an arbitrary direction. The SVD of a symmetric matrix will yield the same triple product as eigenvalue decomposition via a slightly more complex algebraic manipulation.

Another decomposition uses shears to represent non-zero rotations (Paeth, 1990). The following identity allows this:

$$\begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix} = \begin{bmatrix} 1 & \frac{\cos\phi - 1}{\sin\phi} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \sin\phi & 1 \end{bmatrix} \begin{bmatrix} 1 & \frac{\cos\phi - 1}{\sin\phi} \\ 0 & 1 \end{bmatrix}.$$

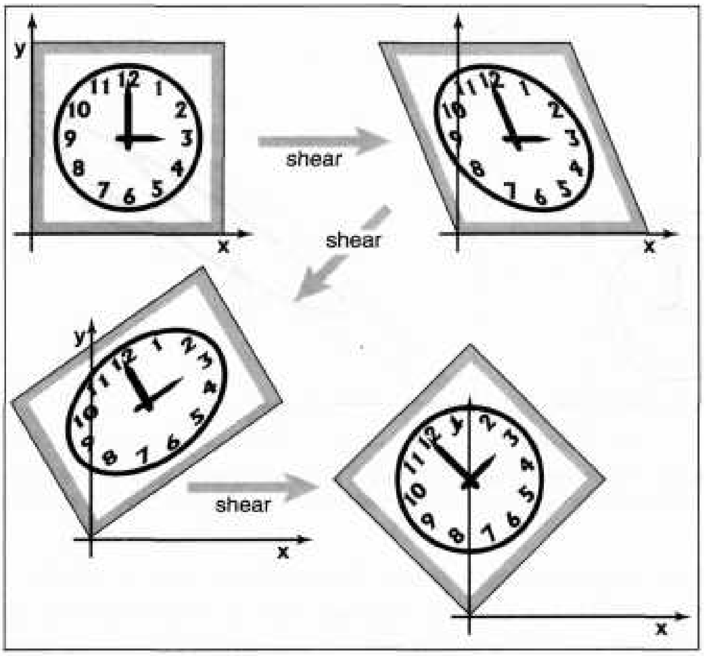


**Figure 6.14.** Any 2D rotation can be accomplished by three shears in sequence. In this case a rotation by 45° is decomposed as shown in Equation 6.2.

For example, a rotation by $\pi/4$ (45 degrees) is (see Figure 6.14)

$$\text{rotate}(\frac{\pi}{4}) = \begin{bmatrix} 1 & 1-\sqrt{2} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \frac{\sqrt{2}}{2} & 1 \end{bmatrix} \begin{bmatrix} 1 & 1-\sqrt{2} \\ 0 & 1 \end{bmatrix}. \tag{6.2}$$

This particular transform is useful for raster rotation because shearing is a very efficient raster operation for images; it introduces some jagginess, but will leave no holes. The key observation is that if we take a raster position $(i, j)$ and apply a horizontal shear to it, we get

$$\begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} i + sj \\ j \end{bmatrix}.$$

If we round $sj$ to the nearest integer, this amounts to taking each row in the image and moving it sideways by some amount—a different amount for each row. Because it is the same displacement within a row, this allows us to rotate with no gaps in the resulting image. A similar action works for a vertical shear. Thus, we can implement a simple raster rotation easily.

## 6.2 Basic 3D Transforms

The basic 3D transforms are an extension of the 2D transforms. For example, a scale along Cartesian axes is

$$\text{scale}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}. \tag{6.3}$$

Rotation is somewhat more complicated because there are now more possible axes of rotation. However, if we simply want to rotate about the $z$-axis, which will only change $x$- and $y$-coordinates, we can use the 2D rotation matrix with no operation on $z$:

$$\text{rotate-z}(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Similarly we can construct matrices to rotate about the $x$-axis and the $y$-axis:

$$\text{rotate-x}(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix},$$

$$\text{rotate-y}(\phi) = \begin{bmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{bmatrix}.$$

We will discuss rotations about arbitrary axes in the next section.

As in two dimensions, we can shear along a particular axis, for example,

$$\text{shear-x}(d_y, d_z) = \begin{bmatrix} 1 & d_y & d_z \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

As with 2D transforms, any 3D transformation matrix can be decomposed using SVD into a rotation, scale, and another rotation. Any symmetric 3D matrix has an eigenvalue decomposition into rotation, scale, and inverse-rotation. Finally, a 3D rotation can be decomposed into a product of 3D shear matrices.

### 6.2.1 Arbitrary 3D Rotations

As in 2D, 3D rotations are *orthonormal* matrices. Geometrically, this means that the three rows of the matrix are the Cartesian coordinates of three mutually-orthogonal unit vectors as discussed in Section 2.4.5. The columns are three, potentially different, mutually-orthogonal unit vectors. There are an infinite number of such rotation matrices. Let's write down such a matrix:

$$\mathbf{R}_{uvw} = \begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix}.$$

Here, $\mathbf{u} = x_u\mathbf{x} + y_u\mathbf{y} + z_u\mathbf{z}$ and so on for $\mathbf{v}$ and $\mathbf{w}$. Since the three vectors are orthonormal we know that

$$\mathbf{u}\cdot\mathbf{u} = \mathbf{v}\cdot\mathbf{v} = \mathbf{w}\cdot\mathbf{w} = 1,$$
$$\mathbf{u}\cdot\mathbf{v} = \mathbf{v}\cdot\mathbf{w} = \mathbf{w}\cdot\mathbf{u} = 0.$$

We can infer some of the behavior of the rotation matrix by applying it to the vectors $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$. For example,

$$\mathbf{R}_{uvw}\mathbf{u} = \begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix} \begin{bmatrix} x_u \\ y_u \\ z_u \end{bmatrix} = \begin{bmatrix} x_ux_u + y_uy_u + z_uz_u \\ x_vx_u + y_vy_u + z_vz_u \\ x_wx_u + y_wy_u + z_wz_u \end{bmatrix}.$$

Note that those three rows of $\mathbf{R}_{uvw}\mathbf{u}$ are all dot products:

$$\mathbf{R}_{uvw}\mathbf{u} = \begin{bmatrix} \mathbf{u}\cdot\mathbf{u} \\ \mathbf{v}\cdot\mathbf{u} \\ \mathbf{w}\cdot\mathbf{u} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \mathbf{x}.$$

Similarly, $\mathbf{R}_{uvw}\mathbf{v} = \mathbf{y}$, and $\mathbf{R}_{uvw}\mathbf{w} = \mathbf{z}$. So $\mathbf{R}_{uvw}$ takes the basis $\mathbf{uvw}$ to the corresponding Cartesian axes via rotation.

If $\mathbf{R}_{uvw}$ is a rotation matrix with orthonormal rows, then $\mathbf{R}_{uvw}^T$ is also a rotation matrix with orthonormal columns, and in fact is the inverse of $\mathbf{R}_{uvw}$ (the inverse of an orthogonal matrix is always its transpose). An important point is that for transformation matrices, the algebraic inverse is also the geometric inverse. So if $\mathbf{R}_{uvw}$ takes $\mathbf{u}$ to $\mathbf{x}$, then $\mathbf{R}_{uvw}^T$ takes $\mathbf{x}$ to $\mathbf{u}$. The same should be true of $\mathbf{v}$ and $\mathbf{y}$ as we can confirm:

$$\mathbf{R}_{uvw}^T\mathbf{y} = \begin{bmatrix} x_u & x_v & x_w \\ y_u & y_v & y_w \\ z_u & z_v & z_w \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} x_v \\ y_v \\ z_v \end{bmatrix} = \mathbf{v}.$$

So we can always create rotation matrices from orthonormal bases.

If we wish to rotate about an arbitrary vector $\mathbf{a}$, we can form an orthonormal basis with $\mathbf{w} = \mathbf{a}$, rotate that basis to the canonical basis $\mathbf{xyz}$, rotate about the $z$-axis, and then rotate the canonical basis back to the $\mathbf{uvw}$ basis. In matrix form, to rotate about the $w$-axis by an angle $\phi$:

$$\begin{bmatrix} x_u & x_v & x_w \\ y_u & y_v & y_w \\ z_u & z_v & z_w \end{bmatrix} \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix}.$$

Here we have $\mathbf{w}$ a unit vector in the direction of $\mathbf{a}$ (i.e. $\mathbf{a}$ divided by its own length). But what are $\mathbf{u}$ and $\mathbf{v}$? A method to find reasonable $\mathbf{u}$ and $\mathbf{v}$ is given in Section 2.4.6.

Note that if we have a rotation matrix and we wish to know in which direction it is rotating, we can compute the one real eigenvalue (which will be $\lambda = 1$), and the corresponding eigenvector is the "pole" of the rotation. This is the one axis that is not changed by the rotation.

### 6.2.2 Transforming Normal Vectors

While most 3D vectors we use represent positions (offset vectors from the origin) or directions, such as where light comes from, some vectors represent *surface normals*. Surface normal vectors are perpendicular to the tangent plane of a surface. These normals do not transform the way we would like when the underlying surface is transformed. For example, if the points of a surface are transformed by a matrix $\mathbf{M}$, a vector $\mathbf{t}$ that is tangent to the surface and is multiplied by $\mathbf{M}$ will be tangent to the transformed surface. However, a surface normal vector $\mathbf{n}$ that is transformed by $\mathbf{M}$ may not be normal to the transformed surface (Figure 6.15).

We can derive a transform matrix $\mathbf{N}$ which does take $\mathbf{n}$ to a vector perpendicular to the transformed surface. One way to attack this issue is to note that a surface normal vector and a tangent vector are perpendicular, so their dot product is zero, which is expressed in matrix form as

$$\mathbf{n}^T\mathbf{t} = \mathbf{0}. \tag{6.4}$$

If we denote the desired transformed vectors as $\mathbf{t}_M = \mathbf{Mt}$ and $\mathbf{n}_N = \mathbf{Nn}$, our goal is to find $\mathbf{N}$ such that $\mathbf{n}_N^T\mathbf{t}_M = \mathbf{0}$. We can find $\mathbf{N}$ by some algebraic

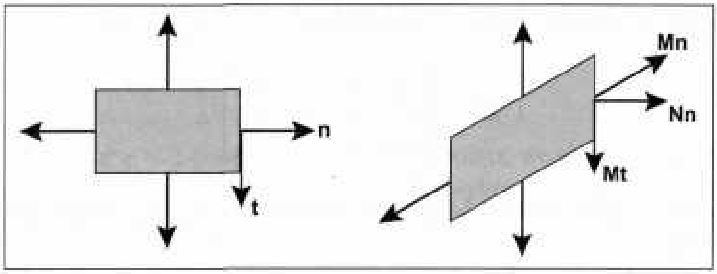


Figure 6.15. When a normal vector is translated using the same matrix that transforms the points on an object, the resulting vector may not be perpendicular to the surface as is shown here for the sheared rectangle. The tangent vector, however, does transform to a vector tangent to the transformed surface.

tricks. First, we can sneak an identity matrix into the dot product, and then take advantage of $\mathbf{M}^{-1}\mathbf{M} = \mathbf{I}$:

$$\mathbf{n}^T\mathbf{t} = \mathbf{n}^T\mathbf{I}\mathbf{t} = \mathbf{n}^T\mathbf{M}^{-1}\mathbf{M}\mathbf{t} = \mathbf{0}.$$

Although the manipulations above don't obviously get us anywhere, note that we can add parentheses that make the above expression more obviously a dot product:

$$\left(\mathbf{n}^T\mathbf{M}^{-1}\right)(\mathbf{M}\mathbf{t}) = \left(\mathbf{n}^T\mathbf{M}^{-1}\right)\mathbf{t}_M = \mathbf{0}.$$

This means that the row vector that is perpendicular to $\mathbf{t}_M$ is the left part of the expression above. This expression holds for any of the tangent vectors in the tangent plane. Since there is only one direction in 3D (and its opposite) that is perpendicular to all such tangent vectors, we know that the left part of the expression above must be the row vector expression for $\mathbf{n}_N$, i.e., it is $\mathbf{n}_N^T$, so this allows us to infer $\mathbf{N}$:

$$\mathbf{n}_N^T = \mathbf{n}^T\mathbf{M}^{-1},$$

so we can take the transpose of that to get

$$\mathbf{n}_N = \left(\mathbf{n}^T\mathbf{M}^{-1}\right)^T = \left(\mathbf{M}^{-1}\right)^T\mathbf{n}. \qquad (6.5)$$

Therefore, we can see that the matrix which correctly transforms normal vectors so they remain normal is $\mathbf{N} = (\mathbf{M}^{-1})^T$, i.e., the transpose of the inverse matrix. Since this matrix may change the length of $\mathbf{n}$, we can multiply it by an arbitrary scalar and it will still produce $\mathbf{n}_N$ with the right direction. Recall from Section 5.2.3 that the inverse of a matrix is the transpose of the cofactor matrix divided by the determinant. Because we don't care about the length of a normal vector, we can skip the division and find that for a 3 by 3 matrix,

$$\mathbf{N} = \begin{bmatrix} m_{11}^c & m_{12}^c & m_{13}^c \\ m_{21}^c & m_{22}^c & m_{23}^c \\ m_{31}^c & m_{32}^c & m_{33}^c \end{bmatrix}.$$

This assumes the element of $\mathbf{M}$ in row $i$ and column $j$ is $m_{ij}$. So the full expression for $\mathbf{N}$ is

$$\mathbf{N} = \begin{bmatrix} m_{22}m_{33} - m_{23}m_{32} & m_{23}m_{31} - m_{21}m_{33} & m_{21}m_{32} - m_{22}m_{31} \\ m_{13}m_{32} - m_{12}m_{33} & m_{11}m_{33} - m_{13}m_{31} & m_{12}m_{31} - m_{11}m_{32} \\ m_{12}m_{23} - m_{13}m_{22} & m_{13}m_{21} - m_{11}m_{23} & m_{11}m_{22} - m_{12}m_{21} \end{bmatrix}.$$

## 6.3 Translation

We have been looking at methods to change vectors using a matrix $\mathbf{M}$. In two dimensions, these transforms have the form,

$$\begin{aligned} x' &= m_{11}x + m_{12}y, \\ y' &= m_{21}x + m_{22}y. \end{aligned}$$

We cannot use such transforms to *move* locations we have represented as offset vectors from the origin. Recall that for directions and offset vectors without an origin, it does not make sense to talk about moving them; we are only talking about locations here. To move a location, we need a transform of the form,

$$\begin{aligned} x' &= x + x_t, \\ y' &= y + y_t. \end{aligned}$$

There is just no way to do that by multiplying $(x, y)$ by a two by two matrix. It would be feasible to just keep track of scales and rotations as a matrix and keep track of translations (moves) separately, but doing that would involve fairly painful bookkeeping. Instead, we can use a technique to move the computation into a higher dimension. This technique has become standard in almost every graphics program and especially in every graphics hardware chip.

The key observation is that when we do a 3D shear based on the $z$-coordinate we get this transform:

$$\begin{bmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x + x_t z \\ y + y_t z \\ z \end{bmatrix}.$$

Note that this almost has the form we want in $x$ and $y$ for a 2D translation, but has a $z$ hanging around that doesn't have a meaning in 2D. Now comes the key decision: we will add a coordinate $z = 1$ to all 2D locations. This gives us:

$$\begin{bmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + x_t \\ y + y_t \\ 1 \end{bmatrix}.$$

By associating a $z = 1$-coordinate with all 2D points, we now can encode translations into matrix form. For example, to first translate in 2D by $(t_x, t_y)$ and then rotate by angle $\phi$ we would use the matrix

$$\mathbf{M} = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{bmatrix}.$$

Note that the 2D rotation matrix is now three by three with zeros in the "translation slots." With this type of formalism, which uses shears along $z = 1$ to encode translations, we can represent any number of 2D shears, 2D rotations, and 2D translations as one composite 3D matrix. Interestingly, the bottom row of that matrix will always be $(0, 0, 1)$, so we even don't really have to store it. We just need to remember it is there when we multiply two matrices together.

A problem with this new formalism is that we do not want direction or arbitrary offset vectors to move when we apply a translation. Fortunately, we can do this by making their third coordinate zero. This gives

$$\begin{bmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}.$$

This is exactly the behavior we want for vectors. So the third coordinate in 2D will be either 1 or 0 depending on whether we are encoding a position or a direction. This coordinate is usually called the *homogeneous* coordinate (Roberts, 1965; Riesenfeld, 1981; Penna & Patterson, 1986). We actually do need to store the homogeneous coordinate so we can distinguish between locations and other vectors. For example,

$$\begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix} \text{ is a location} \quad \text{and} \quad \begin{bmatrix} 3 \\ 2 \\ 0 \end{bmatrix} \text{ is a displacement or direction.}$$

Later, when we do perspective viewing, we will see that it is useful to allow the homogeneous coordinate to be some value other than one or zero.

In 3D, the same technique works: we can add a fourth coordinate, a homogeneous coordinate, and then we have translations:

$$\begin{bmatrix} 1 & 0 & 0 & x_t \\ 0 & 1 & 0 & y_t \\ 0 & 0 & 1 & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + x_t \\ y + y_t \\ z + z_t \\ 1 \end{bmatrix}.$$

Again, for a vector, the fourth coordinate is zero and the vector is thus unaffected by translations.

It is interesting to note that if we multiply an arbitrary matrix composed of shears and rotations with a simple translation, (translation comes second) we get

$$\begin{bmatrix} 1 & 0 & 0 & x_t \\ 0 & 1 & 0 & y_t \\ 0 & 0 & 1 & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & x_t \\ a_{21} & a_{22} & a_{23} & y_t \\ a_{31} & a_{32} & a_{33} & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Thus we can look at any matrix and think of it as a scaling/rotation part and a translation part because the components are nicely separated from each other.

An important class of transforms are *rigid-body* transforms. These are composed only of translations and rotations, so they have no stretching or shrinking of the objects. Such transforms will have a pure rotation for the $a_{ij}$ above.

### 6.3.1 Windowing Transforms

Often in graphics we need to create a transform matrix that takes points in the rectangle $[a, A] \times [b, B]$ to the rectangle $[c, C] \times [d, D]$. This can be accomplished with a single scale and translate in sequence. However, it is more intuitive to create the transform from a sequence of three operations (Figure 6.16):

1. Move the point $(a, b)$ to the origin.

2. Scale the rectangle to be the same size as the target rectangle.

3. Move the origin to point $(c, d)$.

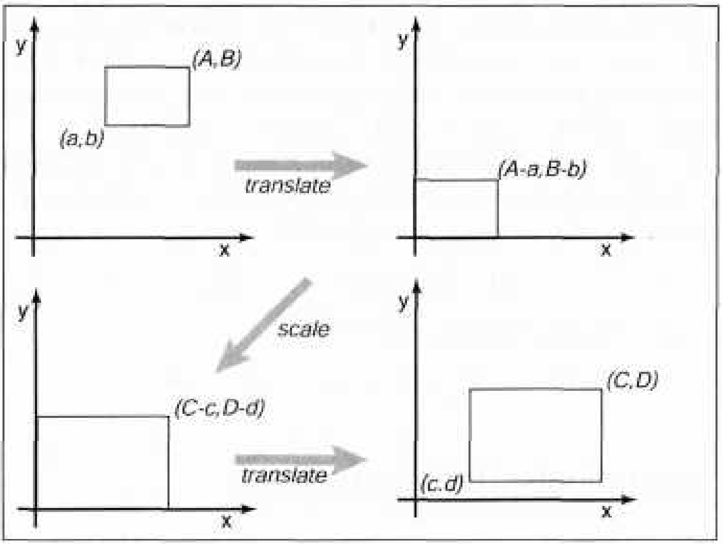


**Figure 6.16.** To take one rectangle (window) to the other, we first shift the lower-left corner to the origin, then scale it to the new size, and then move the origin to the lower-left corner of the target rectangle.

Remembering that the right-hand matrix is applied first, we can write

$$\begin{aligned}\text{window} &= \text{translate}\ (c,d)\ \ \text{scale}\ \left(\frac{C-c}{A-a}, \frac{D-d}{B-b}\right)\ \ \text{translate}\ (-a,-b)\\ &= \begin{bmatrix} 1 & 0 & c \\ 0 & 1 & d \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{C-c}{A-a} & 0 & 0 \\ 0 & \frac{D-d}{B-b} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{bmatrix} \qquad (6.6)\\ &= \begin{bmatrix} \frac{C-c}{A-a} & 0 & \frac{cA-Ca}{A-a} \\ 0 & \frac{D-d}{B-b} & \frac{dB-Db}{B-b} \\ 0 & 0 & 1 \end{bmatrix}.\end{aligned}$$

It is perhaps not surprising to some readers that the resulting matrix has the form it does, but the constructive process with the three matrices leaves no doubt as to the correctness of the result.

## 6.4 Inverses of Transformation Matrices

While we can always invert a matrix algebraically, we can use geometry if we know what the transform does. For example, the inverse of scale$(s_x, s_y, s_z)$ is scale$(1/s_x, 1/s_y, 1/s_z)$. The inverse of a rotation is its transpose. The inverse of a translation is a translation in the opposite direction. If we have a series of matrices $\mathbf{M} = \mathbf{M}_1\mathbf{M}_2 \cdots \mathbf{M}_n$ then $\mathbf{M}^{-1} = \mathbf{M}_n^{-1} \cdots \mathbf{M}_2^{-1}\mathbf{M}_1^{-1}$.

Interestingly, we can use SVD to invert a matrix as well. Since we know that any matrix can be decomposed into a rotation times a scale times a rotation, inversion is straightforward. For example in 3D we have

$$\mathbf{M} = \mathbf{R}_1 \text{scale}(\sigma_1, \sigma_2, \sigma_3)\mathbf{R}_2,$$

and from the rules above it follows easily that

$$\mathbf{M}^{-1} = \mathbf{R}_2^T \text{scale}(1/\sigma_1, 1/\sigma_2, 1/\sigma_3)\mathbf{R}_1^T.$$

## 6.5 Coordinate Transformations

All of the previous discussion has been in terms of using transformation matrices to move points around. We can also think of them as simply changing the coordinate system in which the point is represented. For example, in Figure 6.17, we

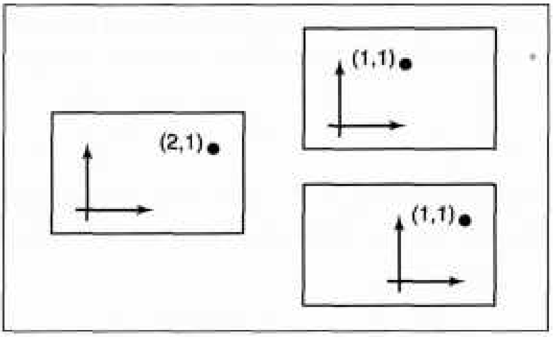


**Figure 6.17.** The point (2,1) has a transform "translate by (-1,0)" applied to it. On the top right is our mental image if we view this transformation as a physical movement, and on the bottom right is our mental image if we view it as a change of coordinates (a movement of the origin in this case). The artificial boundary is just an artifice, and the relative position of the axes and the point are the same in either case.

see two ways to visualize a movement. In different contexts, either interpretation may be more suitable. For example, suppose we have the model of a city and a car. We can provide an illusion of motion as long as the relative coordinates of the car and city change in an appropriate manner. This can be accomplished by changing either car or city coordinates. Intuitively, we should probably think in terms of moving the car; however, we could think in terms of changing the city coordinates as well. It is not really important how one chooses to think about these things, provided the thinking is consistent and documented.

Often we need to manage multiple Cartesian-style coordinate systems, each having its own basis vectors and origins. Typically there is a "global" or "canonical" coordinate system. In 2D the usual convention is is to use the point $\mathbf{o}$ for the origin, and $\mathbf{x}$ and $\mathbf{y}$ for the right-handed orthonormal basis vectors $\mathbf{x}$ and $\mathbf{y}$ (Figure 6.18). Another coordinate system might have an origin $\mathbf{e}$ and right-handed orthonormal basis vectors $\mathbf{u}$ and $\mathbf{v}$. Note that typically the canonical data $\mathbf{o}$, $\mathbf{x}$, and $\mathbf{y}$ are never stored explicitly. They are the frame-of-reference for all other

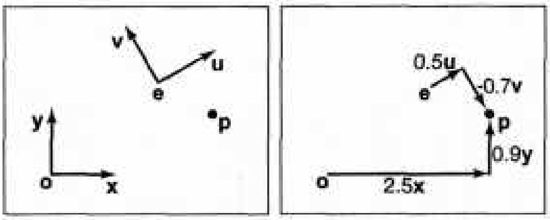


**Figure 6.18.** The point **p** can be represented in terms of either coordinate system.

coordinate systems. In that coordinate system, we often write down the location of $\mathbf{p}$ as an ordered pair, which is shorthand for a full vector expression:

$$\mathbf{p} = (x_p, y_p) \equiv \mathbf{o} + x_p\mathbf{x} + y_p\mathbf{y}.$$

For example, in Figure 6.18, $(x_p, y_p) = (2.5, 0.9)$. Note that the pair $(x_p, y_p)$ implicitly assumes the origin $\mathbf{o}$. Similarly, we can express $\mathbf{p}$ in terms of another equation:

$$\mathbf{p} = (u_p, v_p) \equiv \mathbf{e} + u_p\mathbf{u} + v_p\mathbf{v}. \tag{6.7}$$

In Figure 6.18, this has $(u_p, v_p) = (0.5, -0.7)$. Again, the origin $\mathbf{e}$ is left as an implicit part of the coordinate system associated with $\mathbf{u}$ and $\mathbf{v}$. We can use the simple matrix machinery of this chapter to move back and forth between coordinate systems:

$$\begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_e \\ 0 & 1 & y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & x_v & 0 \\ y_u & y_v & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix}.$$

Note that this assumes we have the point $\mathbf{e}$ and vectors $\mathbf{u}$ and $\mathbf{v}$ stored in canonical coordinates; the $xy$ coordinate system is the first among equals. To go in the other direction we have

$$\begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & y_u & 0 \\ x_v & y_v & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_e \\ 0 & 1 & -y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix}.$$

Analogously, in 3D we have

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & x_e \\ 0 & 1 & 0 & y_e \\ 0 & 0 & 1 & z_e \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & x_v & x_w & 0 \\ y_u & y_v & y_w & 0 \\ z_u & z_v & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ w_p \\ 1 \end{bmatrix}, \tag{6.8}$$

and

$$\begin{bmatrix} u_p \\ v_p \\ w_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix}. \tag{6.9}$$

## Frequently Asked Questions

- Can't I just hardcode transforms rather than use the matrix formalisms?

Yes, but in practice it is harder to derive, harder to debug, and not any more efficient. Also, all current graphics APIs use this matrix formalism so it must be understood even to use graphics libraries.

- The bottom row of the matrix is always (0,0,0,1). Do I have to store it?

You do not have to store it unless you include perspective transforms (Chapter 7).

## Notes

The derivation of the transformation properties of normals is based on *Properties of Surface Normal Transformations* (Turkowski, 1990). In many treatments through the mid-1990s, vectors were represented as row vectors and premultiplied, e.g., $\mathbf{b} = \mathbf{a}\mathbf{M}$. In our notation this would be $\mathbf{b}^T = \mathbf{a}^T\mathbf{M}^T$. If you want to find a rotation matrix $\mathbf{R}$ that takes one vector $\mathbf{a}$ to a vector $\mathbf{b}$ of the same length: $\mathbf{b} = \mathbf{R}\mathbf{a}$ you could use two rotations constructed from orthonormal bases. A more efficient method is given in *Efficiently Building a Matrix to Rotate One Vector to Another* (Möller & Hughes, 1999).

## Exercises

1. Show that the inverse of a rotation matrix is its transpose.
2. Write down the 4 by 4 3D matrix to move by $(x_m, y_m, z_m)$.
3. Write down the 4 by 4 3D matrix to rotate by an angle $\theta$ about the $y$-axis.
4. Write down the 4 by 4 3D matrix to scale an object by 50% in all directions.
5. Write the 2D rotation matrix that rotates by 90 degrees clockwise.
6. Write the matrix from Problem 5 as a product of three shear matrices.
7. Describe in words what this 2D transform matrix does:

$$\begin{bmatrix} 0 & -1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

8. Write down the 3 by 3 matrix that rotates a 2D point by angle $\theta$ about a point $\mathbf{p} = (x_p, y_p)$.

9. Write down the 4 by 4 rotation matrix that takes the orthonormal 3D vectors $\mathbf{u} = (x_u, y_u, z_u)$, $\mathbf{v} = (x_v, y_v, z_v)$ and $\mathbf{w} = (x_w, y_w, z_w)$, to orthonormal 3D vectors $\mathbf{a} = (x_a, y_a, z_a)$, $\mathbf{b} = (x_b, y_b, z_b)$ and $\mathbf{c} = (x_c, y_c, z_c)$. So $M\mathbf{u} = \mathbf{a}$, $M\mathbf{v} = \mathbf{b}$, and $M\mathbf{w} = \mathbf{c}$.

10. What is the inverse matrix for the answer to the previous problem?

# 7

# Viewing

The transform tools developed in the last chapter will make it straightforward for us to create images of 3D line segments. In this chapter, we develop the methods to produce 3D orthographic and perspective views of line segments in space with no "hidden-line" removal (Figure 7.1). Note that in the orthographic projection, the parallel lines in 3D are parallel in the image, while in the perspective projection they may not be parallel in the image. For the entire chapter, we assume that 3D line segments are specified by two end points, each of the form $(x, y, z)$. In the next chapter we use BSP trees and z-buffers to allow opaque objects with hidden-line and hidden-surface removal, and we will use triangle faces rather than triangle edges.

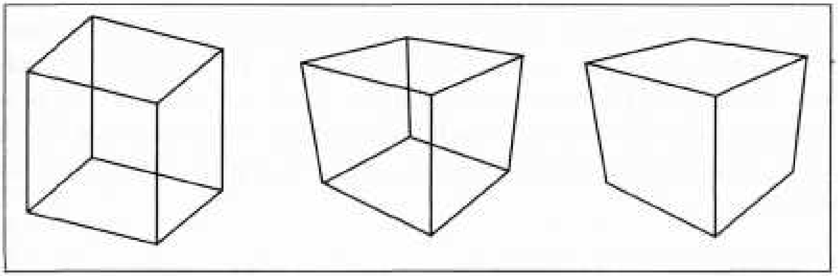


**Figure 7.1.** Left: orthographic projection. Middle: perspective projection. Right: perspective projection with hidden lines removed.

## 7.1 Drawing the Canonical View Volume

We begin with a problem whose solution will be reused for any viewing condition. We want to map lines to the screen along the $z$-axis in the positive direction. We will limit this projection to objects within the *canonical*[1] *view volume*. This is the volume defined by all 3D points whose Cartesian coordinates are between $-1$ and $+1$. This cube can be specified as $(x, y, z) \in [-1, 1]^3$ (Figure 7.2).

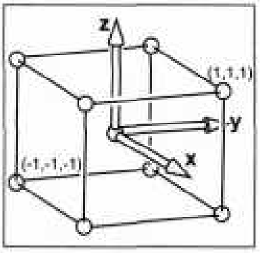


**Figure 7.2.** The canonical view volume is a cube with side of length two centered at the origin.

If we have a screen made up of $n_x$ by $n_y$ pixels, we project $x = -1$ to the left side of the screen, $x = +1$ to the right half of the screen, $y = -1$ to the bottom of the screen, and $y = +1$ to the top of the screen. Note that this is mapping the square $[-1, +1]^2$ to a potentially non-square rectangle. That is not a problem; $x$ and $y$ will just have different scaling parameters when they are converted to pixel coordinates. For now we will assume that all line segments to be drawn are completely inside the canonical view volume. Later we will relax that assumption when we discuss *clipping*.

Recall from Chapter 3 that pixels have a finite square extent, the coordinates have a 0.5 unit overshoot from the pixel centers, and the smallest pixel center coordinates are $(0, 0)$; an $n_x$ by $n_y$ screen has integer centers and boundaries defined by $[-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5]$. This is called a windowing transform and is given by Equation 6.6:

$$\begin{bmatrix} x_{\text{pixel}} \\ y_{\text{pixel}} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{n_x}{2} & 0 & \frac{n_x - 1}{2} \\ 0 & \frac{n_y}{2} & \frac{n_y - 1}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{\text{canonical}} \\ y_{\text{canonical}} \\ 1 \end{bmatrix}. \tag{7.1}$$

Note that there is no allowance for $z$ values here. In practice $z$ values will be kept or the correct ordering of surfaces will not be kept. However, we will ignore $z$ for now to make the matrices a little smaller.

Also, recall that in some APIs the $y$-axis points *downward*. We'll examine this case because it makes a nice exercise. It creates the somewhat odd transform where the $y$ values need to be flipped. The transform to convert points in the canonical view volume to such pixel coordinates is a variant of the windowing transform (Equation 6.6), but because of the flip, we will derive it from scratch.

[1] The word "canonical" crops up in many contexts and usually refers to some customary variable or shape that is in some way "nice." For example, if we were to define a "canonical circle," it would likely be of unit radius and centered at the origin.

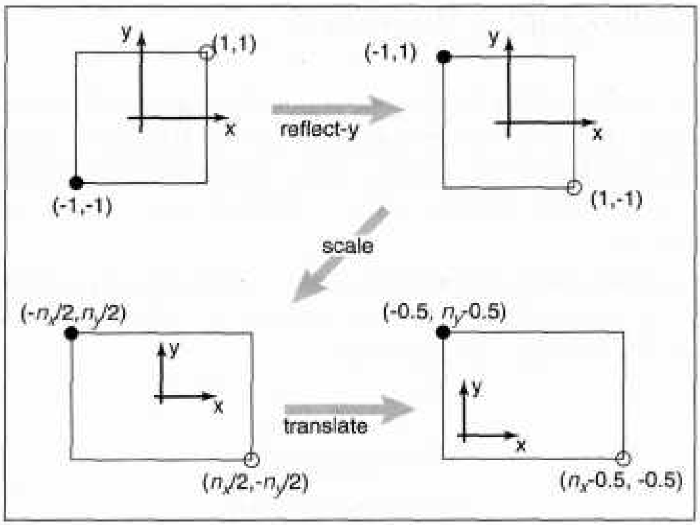


**Figure 7.3.** When the *y* screen coordinates increase from the top of the screen to the bottom, the transform from the *xy*-components of the canonical view volume to the screen coordinates involve a flip.

It can be accomplished as shown in Figure 7.3, which yields

$$\begin{aligned}\begin{bmatrix} x_{\text{pixel}} \\ y_{\text{pixel}} \\ 1 \end{bmatrix} &= \begin{bmatrix} 1 & 0 & \frac{n_x-1}{2} \\ 0 & 1 & \frac{n_y-1}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{n_x}{2} & 0 & 0 \\ 0 & \frac{n_y}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{\text{canonical}} \\ y_{\text{canonical}} \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} \frac{n_x}{2} & 0 & \frac{n_x-1}{2} \\ 0 & -\frac{n_y}{2} & \frac{n_y-1}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{\text{canonical}} \\ y_{\text{canonical}} \\ 1 \end{bmatrix}. \end{aligned} \tag{7.2}$$

We save a lot of pain by doing transforms using more than one matrix as we derive things. For example, there is no reason not to implement Equation 7.1 as a product of three simple matrices. If you implement a matrix library with reflection, scale, and translate initialization routines for matrices, these can be re-used in many situations. Also, the resulting code is easier to debug; note how much easier it is to relate the expanded form of Equation 7.2 to Figure 7.3. If we expand the full-blown algebra by hand, we might make mistakes, and the equations would be less informative; we can see by looking at the matrix product that we scale/flip and then move. That would be much harder to see with the expanded product. So let the matrix multiplication handle the algebra in your implementation. No efficiency is lost except for a small preprocessing time, and your code is more modifiable and more likely to be correct.

## 7.2 Orthographic Projection

We usually want to render lines in some region of space other than the canonical view volume. The basic step is to take each line with 3D endpoints $\mathbf{a}$ and $\mathbf{b}$ and to use a matrix $\mathbf{M}$ to take these points to $\mathbf{Ma}$ and $\mathbf{Mb}$ which can be drawn in the canonical view volume. The matrix $\mathbf{M}$ encodes all the nasty geometry of viewing and projection.

The simplest case occurs when this volume is the axis-aligned box $[l, r] \times [b, t] \times [n, f]$ shown in Figure 7.4. We call this box the *orthographic view volume* and refer to the bounding planes as follows:

$$
\begin{aligned}
x = l &\equiv \text{left plane},\\
x = r &\equiv \text{right plane},\\
y = b &\equiv \text{bottom plane},\\
y = t &\equiv \text{top plane},\\
z = n &\equiv \text{near plane},\\
z = f &\equiv \text{far plane}.
\end{aligned}
$$

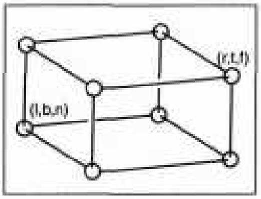


**Figure 7.4.** The orthographic view volume.

That vocabulary assumes a viewer who is looking along the *minus* $z$-axis with his head pointing in the $y$-direction.[2] This implies that $n > f$ which may be unintuitive, but if you assume the entire orthographic view volume has negative $z$ values then the $z = n$ "near" plane is closer to the viewer if and only if $n > f$; here $f$ is a smaller number than $n$, i.e., a negative number of larger absolute value than $n$.

This concept is shown in Figure 7.5. The transform from orthographic view volume to the canonical view volume is really just a 3D version of the 2D windowing transform presented in Section 6.3.1.

This transformation that takes $y = b$ to $y = -1$, $y = t$ to $y = +1$, $x = l$ to $x = -1$, $z = n$ to $z = 1$, and $z = f$ to $z = -1$ can be encoded as a scale and then a move, or a move and then a scale. Note that with this choice for $[n, f]$, $f$ is a more negative number than $n$ in $z$. Here $n$ stands for "near plane" and $f$ stands for "far plane". Many programs flip them at this point as we will discuss

[2] Most programmers find it intuitive to have the $x$-axis pointing right and the $y$-axis pointing up. In a right-handed coordinate system, this implies that we are looking in the $-z$-direction. Some systems use a left-handed coordinate system for viewing so that the gaze direction is along the $+z$-direction. Which is the best compromise is a matter of taste, and this text assumes a right-handed coordinate system. A reference that argues for the left-handed system instead is given in the notes at the end of the chapter.

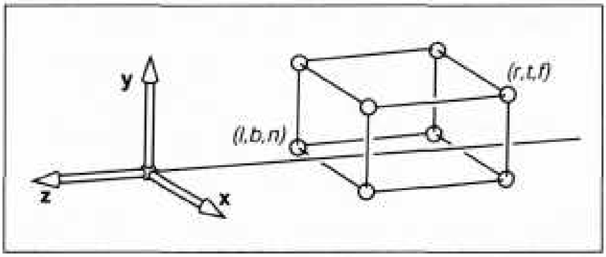


**Figure 7.5.** The orthographic view volume is along the negative *z*-axis, so *f* is a more negative number than *n*, thus *n* > *f*.

later. Moving to the origin first is somewhat more intuitive for most people:

$$\begin{bmatrix} x_{\text{canonical}} \\ y_{\text{canonical}} \\ z_{\text{canonical}} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{b+t}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \qquad (7.3)$$

Note that we have gone to 4 by 4 transformation matrices because $z$ is also being manipulated here, and because $n - f$ is a positive number.

To draw 3D line segments in the orthographic view volume, we project them into screen $xy$-coordinates and ignore $z$-coordinates. We do this by combining Equations 7.1 and 7.3. Because Equation 7.1 is 2D, we add a "do nothing" operation on $z$ to get

$$\mathbf{M}_o = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{b+t}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}. \qquad (7.4)$$

Note that in a program we multiply the three square matrices together to form one matrix $\mathbf{M}_o$, and then manipulate points as follows:

$$\begin{bmatrix} x_{\text{pixel}} \\ y_{\text{pixel}} \\ z_{\text{canonical}} \\ 1 \end{bmatrix} = \mathbf{M}_o \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

The $z$-coordinate will now be in $[-1, 1]$. We don't take advantage of this now, but it will be useful when we examine z-buffer algorithms.

Note that we could also derive Equation 7.4 as just a 3D windowing transform. It is straightforward to generalize Equation 6.6 to 3D; to take the 3D box $[a, A] \times$

$[b, B] \times [c, C]$ to the 3D box $[d, D] \times [e, E] \times [f, F]$ we have

$$\text{window3D} = \begin{bmatrix} \frac{D-d}{A-a} & 0 & 0 & \frac{dA-Da}{A-a} \\ 0 & \frac{E-e}{B-b} & 0 & \frac{eB-Eb}{B-b} \\ 0 & 0 & \frac{F-f}{C-c} & \frac{fC-Fc}{C-c} \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{7.5}$$

Equation 7.4 takes $[l, r] \times [t, b] \times [n, f]$ to $[-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5] \times [-1, 1]$, which can be plugged into Equation 7.5 to get the matrix we want.

The code to draw many 3D lines with endpoints $\mathbf{a}_i$ and $\mathbf{b}_i$ thus becomes both simple and efficient:

```
compute $\mathbf{M}_o$
for each line segment $(\mathbf{a}_i, \mathbf{b}_i)$ do
    $\mathbf{p} = \mathbf{M}_o\mathbf{a}_i$
    $\mathbf{q} = \mathbf{M}_o\mathbf{b}_i$
    drawline($x_p, y_p, x_q, y_q$)
```

This is a first example of how matrix transformation machinery makes graphics programs clean and efficient.

### 7.2.1 Arbitrary View Positions

We'd like to able to change the viewpoint in 3D and look in any direction. There are a multitude of conventions for specifying viewer position and orientation. We will use the following one (see Figure 7.6):

- the eye position $\mathbf{e}$,
- the gaze direction $\mathbf{g}$,
- the view-up vector $\mathbf{t}$.

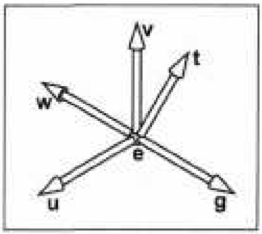


**Figure 7.6.** The user specifies viewing as an eye position **e**, a gaze direction **g**, and an up vector **t**. We construct a right-handed basis with **w** pointing opposite to the gaze and **v** being in the same plane as **g** and **t**.

The eye position is a location that the eye "sees from." If you think of graphics as a photographic process, it is the center of the lens. The gaze direction is any vector in the direction that the viewer is looking. The view-up vector is any vector in the plane that both bisects the viewer's head into right and left halves and points "to the sky" for a person standing on the ground. These vectors provide us with enough information to set up a coordinate system with origin $\mathbf{e}$ and a *uvw* basis:

$$\mathbf{w} = -\frac{\mathbf{g}}{||\mathbf{g}||},$$

$$\mathbf{u} = \frac{\mathbf{t} \times \mathbf{w}}{||\mathbf{t} \times \mathbf{w}||},$$

$$\mathbf{v} = \mathbf{w} \times \mathbf{u}.$$

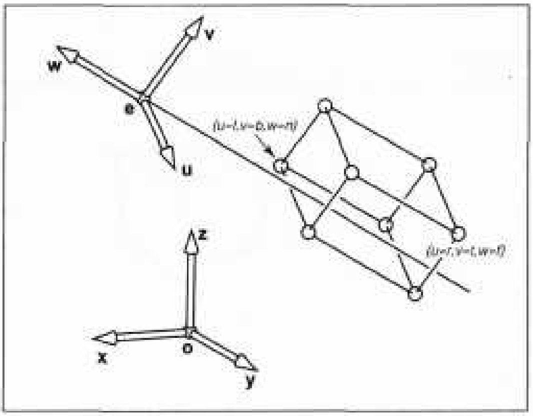


**Figure 7.7.** For arbitrary viewing, we need to change the points to be stored in the "appropriate" coordinate system. In this case it has origin **e** and offset coordinates in terms of **uvw**.

Note that our job would be done if all points we wished to transform were stored in coordinates with origin $\mathbf{e}$ and $uvw$-axes, but as shown in Figure 7.7, they are stored with the canonical origin $\mathbf{o}$ and $xyz$-axes. To use the machinery we have already developed, we just need to convert the coordinates of the line segment endpoints we wish to draw into $(u, v, w)$-coordinates offset from origin $\mathbf{e}$. Alternatively (the math is the same), we can think of the transform as moving $\mathbf{e}$ to the origin and aligning $\mathbf{uvw}$ to $\mathbf{xyz}$. This view transformation is

$$\mathbf{M}_v = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

If we multiply a point $\mathbf{p}$ by $\mathbf{M}_v$, we will have "aligned" it to the coordinate axes. This allows us to make a very minor change to the $z$-axis viewing algorithm we saw earlier:

```
compute $\mathbf{M}_v$
compute $\mathbf{M}_o$
$\mathbf{M} = \mathbf{M}_o\mathbf{M}_v$
for each line segment $(\mathbf{a}_i, \mathbf{b}_i)$ do
    $\mathbf{p} = \mathbf{M}\mathbf{a}_i$
    $\mathbf{q} = \mathbf{M}\mathbf{b}_i$
    drawline$(x_p, y_p, x_q, y_q)$
```

This illustrates the remarkable power we gain when using transformation matrices to implement algorithms. Almost no code is needed once the matrix infrastructure is in place.

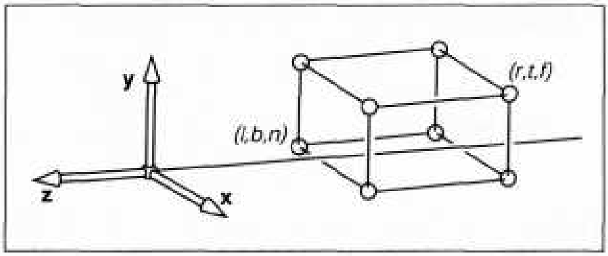


**Figure 7.5.** The orthographic view volume is along the negative $z$-axis, so $f$ is a more negative number than $n$, thus $n > f$.

later. Moving to the origin first is somewhat more intuitive for most people:

$$\begin{bmatrix} x_{\text{canonical}} \\ y_{\text{canonical}} \\ z_{\text{canonical}} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{b+t}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \tag{7.3}$$

Note that we have gone to 4 by 4 transformation matrices because $z$ is also being manipulated here, and because $n - f$ is a positive number.

To draw 3D line segments in the orthographic view volume, we project them into screen $xy$-coordinates and ignore $z$-coordinates. We do this by combining Equations 7.1 and 7.3. Because Equation 7.1 is 2D, we add a "do nothing" operation on $z$ to get

$$\mathbf{M}_o = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{b+t}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{7.4}$$

Note that in a program we multiply the three square matrices together to form one matrix $\mathbf{M}_o$, and then manipulate points as follows:

$$\begin{bmatrix} x_{\text{pixel}} \\ y_{\text{pixel}} \\ z_{\text{canonical}} \\ 1 \end{bmatrix} = \mathbf{M}_o \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

The $z$-coordinate will now be in $[-1, 1]$. We don't take advantage of this now, but it will be useful when we examine z-buffer algorithms.

Note that we could also derive Equation 7.4 as just a 3D windowing transform. It is straightforward to generalize Equation 6.6 to 3D; to take the 3D box $[a, A] \times$

$[b, B] \times [c, C]$ to the 3D box $[d, D] \times [e, E] \times [f, F]$ we have

$$\text{window3D} = \begin{bmatrix} \frac{D-d}{A-a} & 0 & 0 & \frac{dA-Da}{A-a} \\ 0 & \frac{E-e}{B-b} & 0 & \frac{eB-Eb}{B-b} \\ 0 & 0 & \frac{F-f}{C-c} & \frac{fC-Fc}{C-c} \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{7.5}$$

Equation 7.4 takes $[l, r] \times [t, b] \times [n, f]$ to $[-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5] \times [-1, 1]$, which can be plugged into Equation 7.5 to get the matrix we want.

The code to draw many 3D lines with endpoints $\mathbf{a}_i$ and $\mathbf{b}_i$ thus becomes both simple and efficient:

```
compute $\mathbf{M}_o$
for each line segment $(\mathbf{a}_i, \mathbf{b}_i)$ do
    $\mathbf{p} = \mathbf{M}_o\mathbf{a}_i$
    $\mathbf{q} = \mathbf{M}_o\mathbf{b}_i$
    drawline($x_p, y_p, x_q, y_q$)
```

This is a first example of how matrix transformation machinery makes graphics programs clean and efficient.

### 7.2.1 Arbitrary View Positions

We'd like to able to change the viewpoint in 3D and look in any direction. There are a multitude of conventions for specifying viewer position and orientation. We will use the following one (see Figure 7.6):

- the eye position $\mathbf{e}$,
- the gaze direction $\mathbf{g}$,
- the view-up vector $\mathbf{t}$.

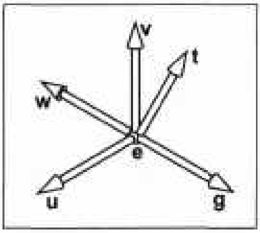


**Figure 7.6.** The user specifies viewing as an eye position **e**, a gaze direction **g**, and an up vector **t**. We construct a right-handed basis with **w** pointing opposite to the gaze and **v** being in the same plane as **g** and **t**.

The eye position is a location that the eye "sees from." If you think of graphics as a photographic process, it is the center of the lens. The gaze direction is any vector in the direction that the viewer is looking. The view-up vector is any vector in the plane that both bisects the viewer's head into right and left halves and points "to the sky" for a person standing on the ground. These vectors provide us with enough information to set up a coordinate system with origin $\mathbf{e}$ and a *uvw* basis:

$$\mathbf{w} = -\frac{\mathbf{g}}{\|\mathbf{g}\|},$$

$$\mathbf{u} = \frac{\mathbf{t} \times \mathbf{w}}{\|\mathbf{t} \times \mathbf{w}\|},$$

$$\mathbf{v} = \mathbf{w} \times \mathbf{u}.$$

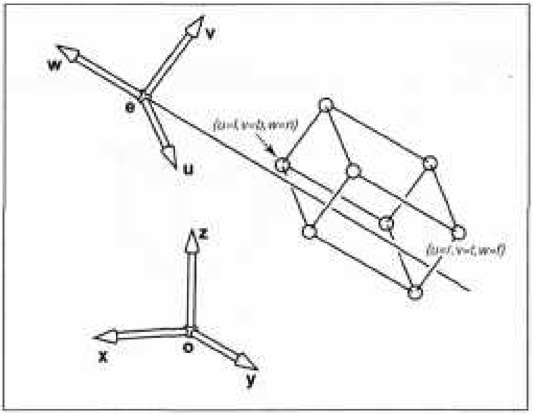


**Figure 7.7.** For arbitrary viewing, we need to change the points to be stored in the "appropriate" coordinate system. In this case it has origin **e** and offset coordinates in terms of **uvw**.

Note that our job would be done if all points we wished to transform were stored in coordinates with origin $\mathbf{e}$ and *uvw*-axes, but as shown in Figure 7.7, they are stored with the canonical origin $\mathbf{o}$ and *xyz*-axes. To use the machinery we have already developed, we just need to convert the coordinates of the line segment endpoints we wish to draw into $(u, v, w)$-coordinates offset from origin $\mathbf{e}$. Alternatively (the math is the same), we can think of the transform as moving $\mathbf{e}$ to the origin and aligning $\mathbf{uvw}$ to $\mathbf{xyz}$. This view transformation is

$$\mathbf{M}_v = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

If we multiply a point $\mathbf{p}$ by $\mathbf{M}_v$, we will have "aligned" it to the coordinate axes. This allows us to make a very minor change to the $z$-axis viewing algorithm we saw earlier:

```
compute $\mathbf{M}_v$
compute $\mathbf{M}_o$
$\mathbf{M} = \mathbf{M}_o\mathbf{M}_v$
for each line segment $(\mathbf{a}_i, \mathbf{b}_i)$ do
    $\mathbf{p} = \mathbf{M}\mathbf{a}_i$
    $\mathbf{q} = \mathbf{M}\mathbf{b}_i$
    drawline$(x_p, y_p, x_q, y_q)$
```

This illustrates the remarkable power we gain when using transformation matrices to implement algorithms. Almost no code is needed once the matrix infrastructure is in place.

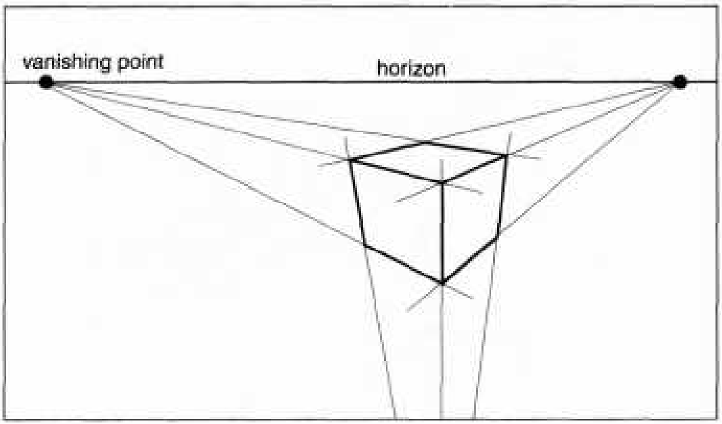


**Figure 7.8.** In three-point perspective, an artist picks "vanishing points" where parallel lines meet. Parallel horizontal lines will meet at a point on the horizon. Every set of parallel lines has its own vanishing points. These rules are followed automatically if we implement perspective based on the correct geometric principles.

## 7.3 Perspective Projection

To capture the effects of *perspective*, we need to draw line segments that are farther from the viewer smaller than similar line segments that are closer to the viewer. While one might expect to automate the artistic conventions of *three-point perspective* (Figure 7.8), in fact all such rules will be followed automatically if we follow the simple mathematical rule underlying perspective: objects are projected directly toward the eye, and they are drawn where they meet a view plane in front of the eye.

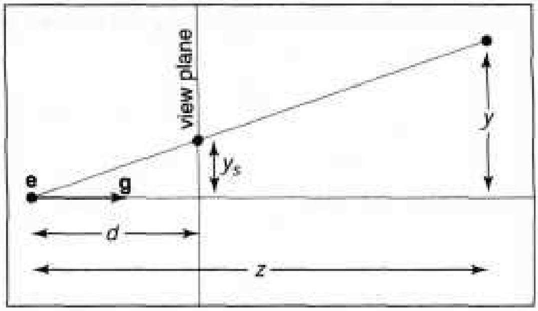


**Figure 7.9.** The geometry for Equation 7.6. The viewer's eye is at **e** and the gaze direction **g** (the minus *z*-axis). The view plane is a distance *d* from the eye. A point is projected toward **e** and where it intersects the view plane is where it is drawn.

In fact, the size of an object on the screen is proportional to $1/z$ for an eye at the origin looking up the negative z-axis. This can be expressed more precisely in an equation for the geometry in Figure 7.9:

$$y_s = \frac{d}{z}y, \tag{7.6}$$

where $y$ is the distance of the point along the $y$-axis, and $y_s$ is where the point should be drawn on the screen.

We would really like to use the matrix machinery we developed for orthographic projection to draw perspective images; we could then just multiply another matrix into our composite matrix and use the algorithm we already have. The geometric operation that allows us to do that is the transformation of the points along a line through the eye to a line parallel to the $z$-axis. This is shown in Figure 7.10. Note that it does not matter *where* the points are transformed in $z$, because the orthographic projection will ignore $z$ anyway. If we want to have the transform work seamlessly with the orthographic projection, then we want to set the view plane to be at $z = n$, and then the portion of the view plane with $(u, v) \in [l, r] \times [b, t]$ will be displayed. Such a transform is shown in Figure 7.11. One property of that transform is that lines through the eye are made parallel in such a way that their intersections with the $z = n$ plane are unchanged (Figure 7.12).

We would like it if we could write down a matrix that would accomplish the above transformation, but it cannot be done; the divide by $z$ in Equation 7.6 is not an operation that the matrix machinery we have seen so far can accomplish. However, there is a beautiful and simple extension to this technology that makes such a "perspective $z$ divide" possible. The key is to use the fourth coordinates of the points, which have been one so far, and allow them to take values other than one.

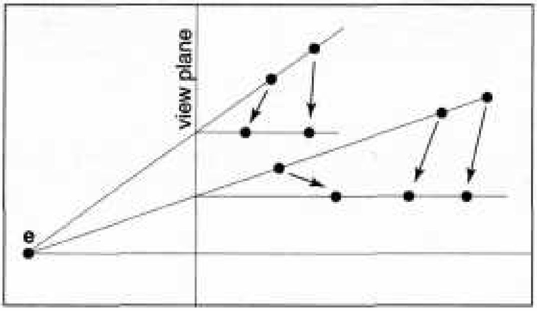


**Figure 7.10.** The perspective projection can be obtained by an orthographic projection provided lines through the eye are all parallel to the gaze direction. The spacing of the resulting lines should intersect the view plane wherever the lines through the eye do.

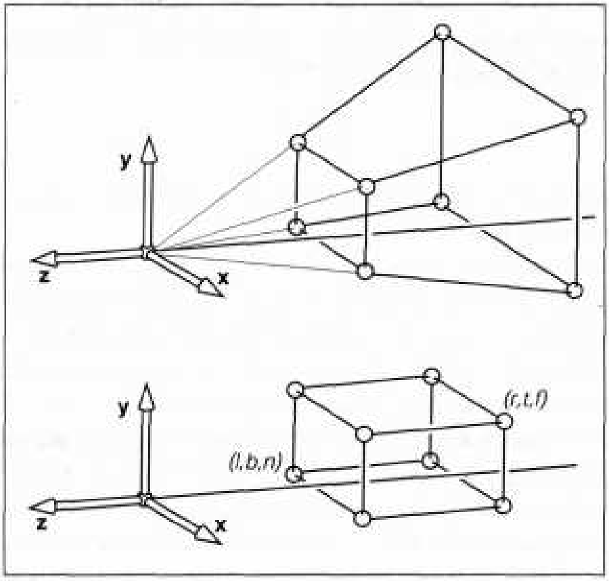


**Figure 7.11.** The perspective projection leaves points on the $z = n$ plane unchanged and maps the large $z = f$ rectangle at the back of the perspective volume to the small $z = f$ rectangle at the back of the orthographic volume.

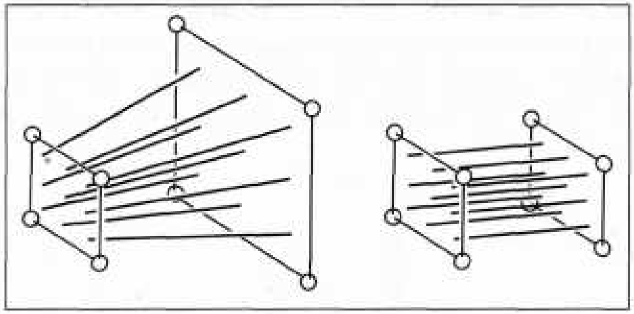


**Figure 7.12.** The perspective projection maps any line through the origin/eye to a line parallel to the $z$-axis and without moving the point on the line at $z = n$.

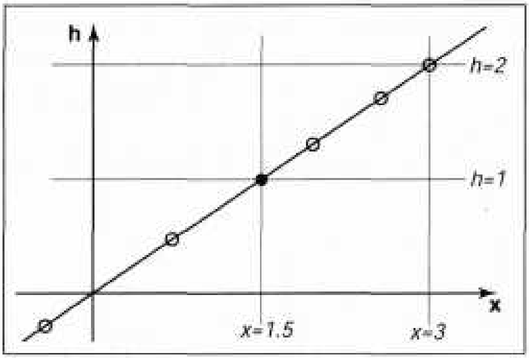


**Figure 7.13.** The homogeneous value $x = 1.5$ is represented by any point on the line $x = 1.5h$, such as points at the hollow circles. However, before we interpret $x$ as a conventional Cartesian coordinate, we first divide by $h$ to get $(x,h) = (1.5,1)$ as shown by the black point.

We refer to this fourth coordinate (for 3D locations) as the *homogeneous* coordinate $h$. This coordinate really encodes how much the other three coordinates have been scaled. Since in the end we only care about $(x, y, z)$, we define the following somewhat strange equivalence:

$$\begin{bmatrix} hx \\ hy \\ hz \\ h \end{bmatrix} \equiv \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

Thus, we can compute with points whose fourth (homogeneous) coordinate $h$ is not one, but we must divide the other three coordinates by $h$ before interpreting them as traditional Cartesian coordinates. For example, the $x$-coordinate is the same for all points on the homogeneous 4D "line" $x = 1.5h$ as shown in Figure 7.13. For example,

$$\begin{bmatrix} 6 \\ -2 \\ 8 \\ 2 \end{bmatrix} \xrightarrow{\text{homogenize}} \begin{bmatrix} 3 \\ -1 \\ 4 \\ 1 \end{bmatrix}.$$

With this *homogenization* procedure, where we get the "real" location by dividing by the fourth coordinate $h$, we can construct the *perspective* matrix

$$\mathbf{M}_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{n+f}{n} & -f \\ 0 & 0 & \frac{1}{n} & 0 \end{bmatrix}.$$

There are many matrices which can function as perspective matrices, and all of them non-linearly distort the $z$-coordinate. This specific matrix has the nice properties shown in Figures 7.11 and 7.12; it leaves points on the $z = n$ plane entirely alone, and it leaves points on the $z = f$ plane at $z = f$ while "squishing" them in $x$ and $y$ by the appropriate amount. The effect of the matrix on a point $(x, y, z)$ is:

$$\mathbf{M}_p \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z\frac{n+f}{n} - f \\ \frac{z}{n} \end{bmatrix} \xrightarrow{\text{homogenize}} \begin{bmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ n + f - \frac{fn}{z} \\ 1 \end{bmatrix}.$$

As you can see, $x$ and $y$ are scaled and, more importantly, divided by $z$. Because both $n$ and $z$ (inside the view volume) are negative, there are no "flips" in $x$ and $y$. Although it is not obvious (see the exercise at the end of the chapter), the transform also preserves the relative order of $z$ values between $z = n$ and $z = f$, allowing us to do depth ordering after this matrix is applied. This will be important later when we do hidden surface elimination.

For homogeneous points, $\mathbf{p} = h\mathbf{p}$, or more explicitly $(x, y, z, 1) = (hx, hy, hz, h)$; thus, we can take any transformation matrix $\mathbf{M}$ and multiply it by an arbitrary constant because $\mathbf{M}(h\mathbf{p}) = (h\mathbf{M})\mathbf{p} = \mathbf{M}\mathbf{p}$. For this reason, we can multiply the perspective matrix by $n$ to make it a little prettier:

$$\mathbf{M}_p = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Sometimes we will want to take the inverse of $\mathbf{M}_p$, for example to bring a screen coordinate plus $z$ back to the original space, as we might want to do for picking. The inverse is

$$\mathbf{M}_p^{-1} = \begin{bmatrix} \frac{1}{n} & 0 & 0 & 0 \\ 0 & \frac{1}{n} & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -\frac{1}{fn} & \frac{n+f}{fn} \end{bmatrix}.$$

As mentioned earlier, the effect of a homogeneous transformation matrix does not change if we multiply it by a constant. The same applies to an inverse matrix; a matrix and its inverse need only produce any diagonal matrix with identical values along the diagonal. So the inverse matrix above is the inverse of any of our perspective matrices. An alternative inverse matrix is found by multiplying by $nf$:

$$\mathbf{M}_p^{-1} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 0 & fn \\ 0 & 0 & -1 & n+f \end{bmatrix}.$$

The beauty of the perspective matrix is, that once we apply it, we can use an orthographic transform to get to the canonical view volume. Thus, all of the orthographic machinery applies, and all that we have added is one matrix and one divide operation. It is also heartening that we are not "wasting" the bottom row of our four by four matrices!

One issue, however, is how are *l,r,b,t* determined for perspective? They identify the "window" through which we look. Since the perspective matrix does not change the values of $x$ and $y$ on the $z = n$ plane, we can specify $(l, r, b, t)$ on that plane.

We would like to integrate the perspective matrix into our orthographic infrastructure. A key point is that the perspective matrix assumes we are looking up the $-z$-axis, so it cannot be applied until after the viewing matrix $\mathbf{M}_v$ has been applied. So the full set of matrices for perspective viewing is

$$\mathbf{M} = \mathbf{M}_o \mathbf{M}_p \mathbf{M}_v.$$

The resulting algorithm is:

```
compute $\mathbf{M}_o$
compute $\mathbf{M}_v$
compute $\mathbf{M}_p$
$\mathbf{M} = \mathbf{M}_o\mathbf{M}_p\mathbf{M}_v$
for each line segment $(\mathbf{a}_i, \mathbf{b}_i)$ do
  $\mathbf{p} = \mathbf{M}\mathbf{a}_i$
  $\mathbf{q} = \mathbf{M}\mathbf{b}_i$
  drawline($x_p/h_p, y_p/h_p, x_q/h_q, y_q/h_q$)
```

Note that the only change other than the additional matrix is the divide by the homogeneous coordinate $h$.

You might have the misgiving that the perspective matrix changes the value of the homogeneous coordinate, so the move and scale might no longer work properly. However, note that for a move on a homogeneous point we have

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} hx \\ hy \\ hz \\ h \end{bmatrix} = \begin{bmatrix} hx + ht_x \\ hy + ht_y \\ hz + ht_z \\ h \end{bmatrix} \xrightarrow{\text{homogenize}} \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix}.$$

Similar effects are true for other transforms (see the exercise at the end of the chapter).

The matrix $\mathbf{M}_{\text{projection}}$, commonly called the *projection* matrix, is the product of the matrix in Equation 7.3 and $\mathbf{M}_p$:

$$\mathbf{M}_{\text{projection}} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{l-r} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{b-t} & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

This or similar matrices often appear in documentation, and they are less mysterious when one realizes that they are usually the product of a few simple matrices.

Many APIs such as *OpenGL* (Shreiner, Neider, Woo, & Davis, 2004) use the same canonical view volume as presented here. They also usually have the user specify the absolute values of $n$ and $f$. The projection matrix for *OpenGL* is

$$\mathbf{M}_{\text{OpenGL}} = \begin{bmatrix} \frac{2|n|}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2|n|}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{|n|+|f|}{|n|-|f|} & \frac{2|f||n|}{|n|-|f|} \\ 0 & 0 & -1 & 0 \end{bmatrix}.$$

Other APIs set $n$ and $f$ to 0 and 1, respectively. Blinn (J. Blinn, 1996) recommends making the canonical view volume $[0,1]^3$ for efficiency. All such decisions will change the the projection matrix slightly.

## 7.4 Some Properties of the Perspective Transform

An important property of the perspective transform is that it takes lines to lines and planes to planes. In addition, it takes line segments in the view volume to line segments in the canonical volume. To see this, consider the line segment

$$\mathbf{q} + t(\mathbf{Q} - \mathbf{q}).$$

When transformed by a 4 by 4 matrix $\mathbf{M}$, it is a point with possibly varying homogeneous coordinate:

$$\mathbf{M}\mathbf{q} + t(\mathbf{M}\mathbf{Q} - \mathbf{M}\mathbf{q}) \equiv \mathbf{r} + t(\mathbf{R} - \mathbf{r}).$$

The homogenized 3D line segment is

$$\frac{\mathbf{r} + t(\mathbf{R} - \mathbf{r})}{h_r + t(h_R - h_r)}. \tag{7.7}$$

If Equation 7.7 can be rewritten in a form

$$\frac{\mathbf{r}}{h_r} + f(t)\left(\frac{\mathbf{R}}{h_R} - \frac{\mathbf{r}}{h_r}\right), \tag{7.8}$$

then all the homogenized points lie on a 3D line. Brute force manipulation of Equation 7.7 yields such a form with

$$f(t) = \frac{h_R t}{h_r + t(h_R - h_r)}. \tag{7.9}$$

It also turns out that the line segments do map to line segments preserving the ordering of the points (Exercise 8), i.e., they do not get reordered or "torn."

A byproduct of the transform taking line segments to line segments is that it takes the edges and vertices of a triangle to the edges and vertices of another triangle. Thus, it takes triangles to triangles and planes to planes.

## 7.5 Field-of-View

While we can specify any window using the $(l, r, b, t)$ and $n$ values, sometimes we would like to have a simpler system where we look through the center of the window. This implies the constraint that

$$\begin{aligned} l &= -r \\ b &= -t. \end{aligned}$$

If we also add the constraint that the pixels are square, i.e., there is no distortion of shape in the image, then the ratio of $r$ to $t$ must be the same as the ratio of the number of horizontal pixels to the number of vertical pixels:

$$\frac{n_x}{n_y} = \frac{r}{t}.$$

Once $n_x$ and $n_y$ are specified, this leaves only one degree of freedom. That is often set using the *field-of-view* shown as $\theta$ in Figure 7.14. This is sometimes called the vertical field-of-view to distinguish it from the angle between left and right sides or from the angle between diagonal corners. From the figure we can see that

$$\tan\frac{\theta}{2} = \frac{t}{|n|}.$$

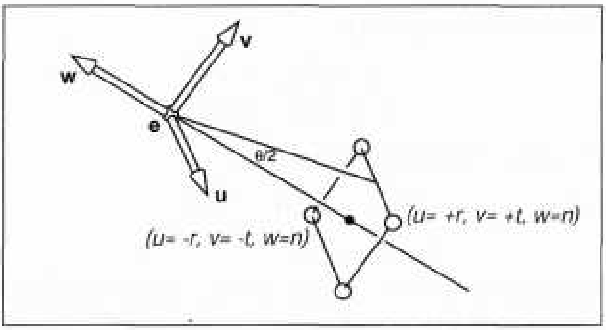


**Figure 7.14.** The field-of-view $\theta$ is the angle from the bottom of the screen to the top of the screen as measured from the eye.

If $n$ and $\theta$ are specified, then we can derive $t$ and use code for the more general viewing system. In some systems, the value of $n$ is hard-coded to some reasonable value, and thus we have one fewer degree of freedom.

## Frequently Asked Questions

- Is orthographic projection ever useful in practice?

It is useful in applications where relative length judgements are important. It can also yield simplifications where perspective would be too expensive as occurs in some medical visualization applications.

- The tessellated spheres I draw in perspective look like ovals. Is this a bug?

No. It is correct behavior. If you place your eye in the same relative position to the screen as the virtual viewer has with respect to the viewport, then these ovals will look like circles because they themselves are viewed at an angle.

- Does the perspective matrix take negative $z$ values to positive z-values with a reversed ordering? Doesn't that cause trouble?

Yes. The equation for transformed $z$ is

$$z' = n + f - \frac{fn}{z}.$$

So $z = +\epsilon$ is transformed to $z' = -\infty$ and $z = -\epsilon$ is transformed to $z = \infty$. So any line segments that span $z = 0$ will be "torn" although all points will be projected to an appropriate screen location. This tearing is not relevant when all objects are contained in the viewing volume. This is usually assured by *clipping* to the view volume. However, clipping itself is made more complicated by the tearing phenomenon as is discussed in Chapter 12.

## Notes

Most of the discussion of viewing matrices is based on information in *Real-Time Rendering* (Möller & Haines, 1999), the *OpenGL Programming Guide* (Shreiner et al., 2004), *Computer Graphics* (Hearn & Baker, 1986), and *3D Game Engine Design* (Eberly, 2000).

## Exercises

1. Show algebraically that the perspective matrix preserves order of $z$ values within the view volume.
2. For a four by four matrix whose top three rows are arbitrary and whose bottom row is $(0, 0, 0, 1)$, show that the points $(x, y, z, 1)$ and $(hx, hy, hz, h)$ transform to the same point after homogenization.
3. Verify that the form of $\mathbf{M}_p^{-1}$ given in the text is correct.
4. Verify that the full perspective to canonical matrix $\mathbf{M}_{\text{projection}}$ takes $(r, t, n)$ to $(1, 1, 1)$.
5. Write down a perspective matrix for $n = 1$, $f = 2$.
6. For the point $\mathbf{p} = (x, y, z, 1)$, what are the homogenized and unhomogenized result for that point transformed by the perspective matrix in Problem 3?
7. For the eye position $\mathbf{e} = (0, 1, 0)$, a gaze vector $\mathbf{g} = (0, -1, 0)$, and a view-up vector $\mathbf{t} = (1, 1, 0)$, what is the resulting orthonormal $\mathbf{uvw}$ basis used for coordinate rotations?
8. Show, that for a perspective transform, line segments that start in the view volume do map to line segments in the canonical volume after homogenization. Further, show that the relative ordering of points on the two segments is the same. *Hint: show that the $f(t)$ in Equation 7.9 has the properties $f(0) = 0$, $f(1) = 1$, the derivative of $f$ is positive for all $t \in [0, 1]$, and the homogeneous coordinate does not change sign.*

# 8

# Hidden Surface Elimination

While we know how to get a single triangle onto the screen by projecting its vertices from 3D to the canonical view volume, we will achieve more realism if we also do *hidden surface elimination*, where only the closest surface is visible to the viewer. This can be achieved through numerous methods; we only cover the two most commonly used ones here: BSP trees (Fuchs, Kedem, & Naylor, 1980) and z-buffering (Catmull, 1975). Ray tracing can also be thought of as a hidden surface algorithm, but it will be discussed in its own chapter since it does not integrate well into the standard project-and-rasterize process. There are many other hidden surface algorithms (Sutherland, Sproull, & Schumacker, 1974), but few besides those three are used in practice.

## 8.1 BSP Tree

If we are making many images of the same geometry from different viewpoints, as is often the case for applications such as games, we can use a *binary space partitioning* (BSP) tree algorithm to order the surfaces from front to back. The key aspect of the BSP tree is that it uses a preprocess to create a data structure that is useful for any viewpoint. So, as the viewpoint changes, the same data structure is used without change.

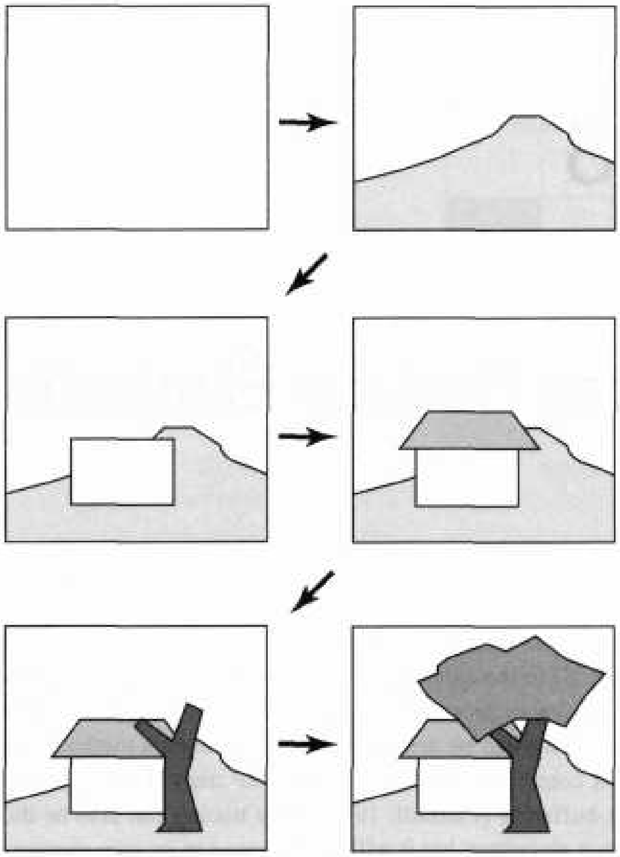


Figure 8.1. A painter's algorithm starts with a blank image and then draws the scene one object at a time from back-to-front, overdrawing whatever is already there. This automatically eliminates hidden surfaces.

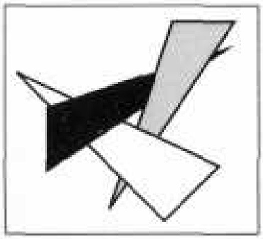


Figure 8.2. A cycle occurs if a global back-to-front ordering is not possible for a particular eye position.

### 8.1.1 Overview of BSP Tree Algorithm

The BSP tree algorithm is an example of a *painter's algorithm*. A painter's algorithm draws every object from back-to-front, with each new polygon potentially overdrawing previous polygons, as is shown in Figure 8.1. It can be implemented as follows:

```
sort objects back to front relative to viewpoint
for each object do
    draw object on screen
```

The problem with the first step (the sort) is that the relative order of multiple objects is not always well defined, even if the order of every pair of objects is. This problem is illustrated in Figure 8.2 where the three triangles form a *cycle*.

The BSP tree algorithm works on any scene composed of polygons where no polygon crosses the plane defined by any other polygon. This restriction is then relaxed by a preprocessing step. For the rest of this discussion, triangles are assumed to be the only primitive, but the ideas extend to arbitrary polygons.

The basic idea of the BSP tree can be illustrated with two triangles, $T_1$ and $T_2$. We first recall (see Section 2.7) the implicit plane equation of the plane containing $T_1$: $f_1(\mathbf{p}) = 0$. The key property of implicit planes that we wish to take advantage of is that for all points $\mathbf{p}^+$ on one side of the plane, $f_1(\mathbf{p}^+) > 0$; and for all points $\mathbf{p}^-$ on the other side of the plane, $f_1(\mathbf{p}^-) < 0$. Using this property, we can find out on which side of the plane $T_2$ lies. Again, this assumes all three vertices of $T_2$ are on the same side of the plane. For discussion, assume that $T_2$ is on the $f_1(\mathbf{p}) < 0$ side of the plane. Then, we can draw $T_1$ and $T_2$ in the right order for any eyepoint $\mathbf{e}$:

```
if ($f_1(\mathbf{e}) < 0$) then
  draw $T_1$
  draw $T_2$
else
  draw $T_2$
  draw $T_1$
```

The reason this works is that if $T_2$ and $\mathbf{e}$ are on the same side of the plane containing $T_1$, there is no way for $T_2$ to be fully or partially blocked by $T_1$ as seen from $\mathbf{e}$, so it is safe to draw $T_1$ first. If $\mathbf{e}$ and $T_2$ are on opposite sides of the plane containing $T_1$, then $T_2$ cannot fully or partially block $T_1$, and the opposite drawing order is safe (Figure 8.3).

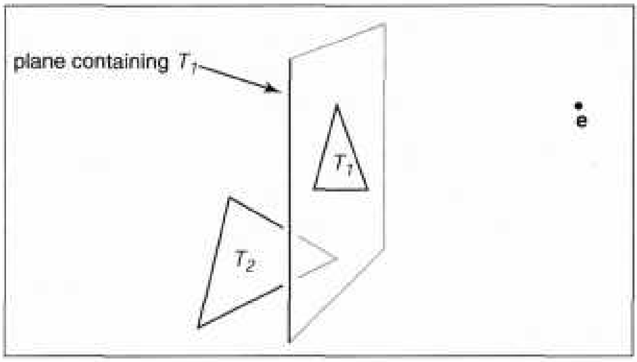


**Figure 8.3.** When **e** and $T_2$ are on opposite sides of the plane containing $T_1$, then it is safe to draw $T_2$ first and $T_1$ second. If **e** and $T_2$ are on the same side of the plane, then $T_1$ should be drawn before $T_2$. This is the core idea of the BSP tree algorithm.

This observation can be generalized to many objects provided none of them span the plane defined by $T_1$. If we use a binary tree data structure with $T_1$ as root, the *negative* branch of the tree contains all the triangles whose vertices have $f_i(\mathbf{p}) < 0$, and the *positive* branch of the tree contains all the triangles whose vertices have $f_i(\mathbf{p}) > 0$. We can draw in proper order as follows:

```
function draw(bsptree tree, point $\mathbf{e}$)
if (tree.empty) then
    return
if ($f_{\text{tree.root}}(\mathbf{e}) < 0$) then
    draw(tree.plus, $\mathbf{e}$)
    rasterize tree.triangle
    draw(tree.minus, $\mathbf{e}$)
else
    draw(tree.minus, $\mathbf{e}$)
    rasterize tree.triangle
    draw(tree.plus, $\mathbf{e}$)
```

The nice thing about that code is that it will work for any viewpoint $\mathbf{e}$, so the tree can be precomputed. Note that, if each subtree is itself a tree, where the root triangle divides the other triangles into two groups relative to the plane containing it, the code will work as is. It can be made slightly more efficient by terminating the recursive calls one level higher, but the code will still be simple. A tree illustrating this code is shown in Figure 8.4. As discussed in Section 2.7.2, the implicit equation for a point $\mathbf{p}$ on a plane containing three non-colinear points $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ is

$$f(\mathbf{p}) = ((\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})) \cdot (\mathbf{p} - \mathbf{a}) = 0. \tag{8.1}$$

It can be faster to store the $(A, B, C, D)$ of the implicit equation of the form

$$f(x, y, z) = Ax + By + Cz + D = 0. \tag{8.2}$$

Equations 8.1 and 8.2 are equivalent, as is clear when you recall that the gradient of the implicit equation is the normal to the triangle. The gradient of Equation 8.2 is $\mathbf{n} = (A, B, C)$ which is just the normal vector

$$\mathbf{n} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a}).$$

We can solve for $D$ by plugging in any point on the plane, e.g., $\mathbf{a}$:

$$\begin{aligned} D &= -Ax_a - By_a - Cz_a \\ &= -\mathbf{n} \cdot \mathbf{a}. \end{aligned}$$

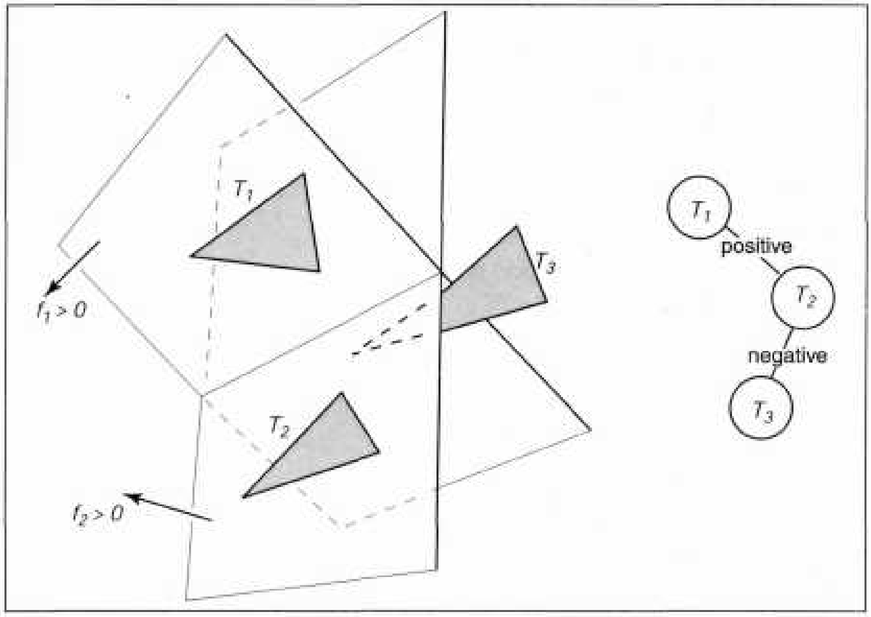


**Figure 8.4.** Three triangles and a BSP tree that is valid for them. The "positive" and "negative" are encoded by right and left subtree position, respectively.

This suggests the form:

$$\begin{aligned} f(\mathbf{p}) &= \mathbf{n} \cdot \mathbf{p} - \mathbf{n} \cdot \mathbf{a} \\ &= \mathbf{n} \cdot (\mathbf{p} - \mathbf{a}) \\ &= 0, \end{aligned}$$

which is the same as Equation 8.1 once you recall that $\mathbf{n}$ is computed using the cross product. Which form of the plane equation you use and whether you store only the vertices, $\mathbf{n}$ and the vertices, or $\mathbf{n}$, $D$, and the vertices, is probably a matter of taste—a classic time-storage tradeoff that will be settled best by profiling. For debugging, using Equation 8.1 is probably the best.

The only issue that prevents the code above from working in general is that one cannot guarantee that a triangle can be uniquely classified on one side of a plane or the other. It can have two vertices on one side of the plane and the third on the other. Or it can have vertices on the plane. This is handled by splitting the triangle into smaller triangles using the plane to "cut" them.

### 8.1.2 Building the Tree

If none of the triangles in the dataset cross each other's planes, so that all triangles are on one side of all other triangles, a BSP tree that can be traversed using the code above can be built using the following algorithm:

```
tree-root = node($T_1$)
for $i \in \{2, \ldots, N\}$ do
  tree-root.add($T_i$)

function add ( triangle $T$ )
if ($f(\mathbf{a}) < 0$ and $f(\mathbf{b}) < 0$ and $f(\mathbf{c}) < 0$) then
  if (negative subtree is empty) then
    negative-subtree = node($T$)
  else
    negative-subtree.add ($T$)
else if ($f(\mathbf{a}) > 0$ and $f(\mathbf{b}) > 0$ and $f(\mathbf{c}) > 0$) then
  if positive subtree is empty then
    positive-subtree = node($T$)
  else
    positive-subtree.add ($T$)
else
  we have assumed this case is impossible
```

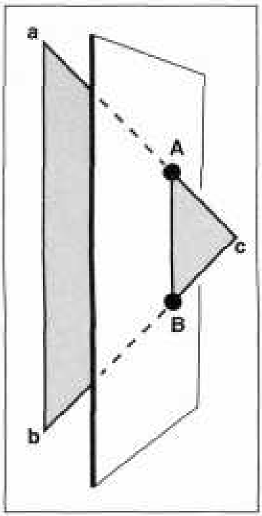


**Figure 8.5.** When a triangle spans a plane, there will be one vertex on one side and two on the other.

The only thing we need to fix is the case where the triangle crosses the dividing plane, as shown in Figure 8.5. Assume, for simplicity, that the triangle has vertices $\mathbf{a}$ and $\mathbf{b}$ on one side of the plane, and vertex $\mathbf{c}$ is on the other side. In this case, we can find the intersection points $\mathbf{A}$ and $\mathbf{B}$ and cut the triangle into three new triangles with vertices

$$T_1 = (\mathbf{a}, \mathbf{b}, \mathbf{A}),$$
$$T_2 = (\mathbf{b}, \mathbf{B}, \mathbf{A}),$$
$$T_3 = (\mathbf{A}, \mathbf{B}, \mathbf{c}),$$

as shown in Figure 8.6. This order of vertices is important so that the direction of the normal remains the same as for the original triangle. If we assume that $f(\mathbf{c}) < 0$, the following code could add these three triangles to the tree assuming the positive and negative subtrees are not empty:

```
positive-subtree = node ($T_1$)
positive-subtree = node ($T_2$)
negative-subtree = node ($T_3$)
```

A precision problem that will plague a naive implementation occurs when a vertex is very near the splitting plane. For example, if we have two vertices on one side of the splitting plane and the other vertex is only an extremely small distance on the other side, we will create a new triangle almost the same as the old one, a triangle that is a sliver, and a triangle of almost zero size. It would be better to detect this as a special case and not split into three new triangles. One might expect this case to be rare, but because many models have tessellated planes and triangles with shared vertices, it occurs frequently, and thus must be handled carefully. Some simple manipulations that accomplish this are:

```
function add( triangle $T$ )
fa = $f(\mathbf{a})$
fb = $f(\mathbf{b})$
fc = $f(\mathbf{c})$
if $(abs(fa) < \epsilon)$ then
  fa = 0
if $(abs(fb) < \epsilon)$ then
  fb = 0
if $(abs(fc) < \epsilon)$ then
  fc = 0
if ($fa \leq 0$ and fb $\leq 0$ and fc $\leq 0$) then
  if (negative subtree is empty) then
    negative-subtree = node($T$)
  else
    negative-subtree.add($T$)
else if ($fa \geq 0$ and fb $\geq 0$ and fc $\geq 0$) then
  if (positive subtree is empty) then
    positive-subtree = node($T$)
  else
    positive-subtree.add($T$)
else
  cut triangle into three triangles and add to each side
```

This takes any vertex whose $f$ value is within $\epsilon$ of the plane and counts it as positive or negative. The constant $\epsilon$ is a small positive real chosen by the user. The technique above is a rare instance where testing for floating point equality is useful and works because the zero value is set rather than being computed. Comparing for equality with a computed floating point value is almost never advisable, but we are not doing that.

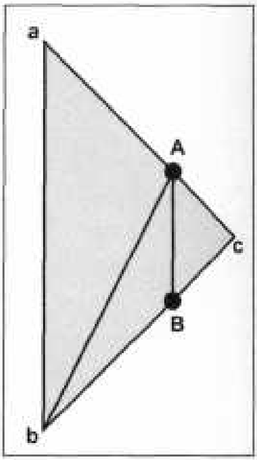


**Figure 8.6.** When a triangle is cut, we break it into three triangles, none of which span the cutting plane.

### 8.1.3 Cutting Triangles

Filling out the details of the last case "cut triangle into three triangles and add to each side" is straightforward, but tedious. We should take advantage of the BSP tree construction as a preprocess where highest efficiency is not key. Instead, we should attempt to have a clean compact code. A nice trick is to force many of the cases into one by ensuring that **c** is on one side of the plane and the other two vertices are on the other. This is easily done with swaps. Filling out the details in the final else statement (assuming the subtrees are non-empty for simplicity) gives:

```
if ($fa * fc \geq 0$) then
    swap($fb, fc$)
    swap($\mathbf{b}, \mathbf{c}$)
    swap($fa, fb$)
    swap($\mathbf{a}, \mathbf{b}$)

else if ($fb * fc \geq 0$) then
    swap($fa, fc$)
    swap($\mathbf{a}, \mathbf{c}$)
    swap($fa, fb$)
    swap($\mathbf{a}, \mathbf{b}$)
compute $\mathbf{A}$
compute $\mathbf{B}$
$T_1 = (\mathbf{a}, \mathbf{b}, \mathbf{A})$
$T_2 = (\mathbf{b}, \mathbf{B}, \mathbf{A})$
$T_3 = (\mathbf{A}, \mathbf{B}, \mathbf{c})$
if ($fc \geq 0$) then
    negative-subtree.add($T_1$)
    negative-subtree.add($T_2$)
    positive-subtree.add($T_3$)
else
    positive-subtree.add($T_1$)
    positive-subtree.add($T_2$)
    negative-subtree.add($T_3$)
```

This code takes advantage of the fact that the product of $a$ and $b$ are positive if they have the same sign—thus, the first if statement. If vertices are swapped, we must do two swaps to keep the vertices ordered counterclockwise. Note that exactly one of the vertices may lie exactly on the plane, in which case the code above will work, but one of the generated triangles will have zero area. This can be handled by ignoring the possibility, which is not that risky, because the rasterization code must handle zero area triangles in screen space (i.e., edge-on triangles). You can also add a check that does not add zero-area triangles to the tree. Finally, you can put in a special case for when exactly one of fa, fb, and fc is zero which cuts the triangle into two triangles.

To compute $\mathbf{A}$ and $\mathbf{B}$, a line segment and implicit plane intersection is needed. For example, the parametric line connecting $\mathbf{a}$ and $\mathbf{c}$ is

$$\mathbf{p}(t) = \mathbf{a} + t(\mathbf{c} - \mathbf{a}).$$

The point of intersection with the plane $\mathbf{n} \cdot \mathbf{p} + D = 0$ is found by plugging $\mathbf{p}(t)$ into the plane equation:

$$\mathbf{n} \cdot (\mathbf{a} + t(\mathbf{c} - \mathbf{a})) + D = 0,$$

and solving for $t$:

$$t = -\frac{\mathbf{n} \cdot \mathbf{a} + D}{\mathbf{n} \cdot (\mathbf{c} - \mathbf{a})}.$$

Calling this solution $t_A$, we can write the expression for $\mathbf{A}$:

$$\mathbf{A} = \mathbf{a} + t_A(\mathbf{c} - \mathbf{a}).$$

A similar computation will give $\mathbf{B}$.

### 8.1.4 Optimizing the Tree

The efficiency of tree creation is much less of a concern than tree traversal because it is a preprocess. The traversal of the BSP tree takes time proportional to the number of nodes in the tree. (How well balanced the tree is does not matter.) There will be one node for each triangle, including the triangles that are created as a result of splitting. This number can depend on the order in which triangles are added to the tree. For example, in Figure 8.7, if $T_1$ is the root, there will be two nodes in the tree, but if $T_2$ is the root, there will be more nodes, because $T_1$ will be split.

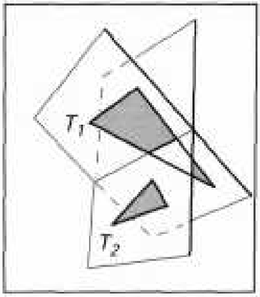


**Figure 8.7.** Using $T_1$ as the root of a BSP tree will result in a tree with two nodes. Using $T_2$ as the root will require a cut and thus make a larger tree.

It is difficult to find the "best" order of triangles to add to the tree. For $N$ triangles, there are $N!$ orderings that are possible. So trying all orderings is not usually feasible. Alternatively, some predetermined number of orderings can be tried from a random collection of permutations, and the best one can be kept for the final tree.

The splitting algorithm described above splits one triangle into three triangles. It could be more efficient to split a triangle into a triangle and a convex quadrilateral. This is probably not worth it if all input models have only triangles, but would be easy to support for implementations that accommodate arbitrary polygons.

## 8.2 Z-Buffer

The z-buffer algorithm can be found in hardware on almost every video game and graphics PC. However, it is also a useful software algorithm (Cook, Carpenter, & Catmull, 1987). The z-buffer takes advantage of the fact that our real problem is to find the closest polygon to the center of each pixel; this can be an easier problem than finding a true depth order in continuous screen space.

### 8.2.1 Z-Buffer Algorithm

The z-buffer algorithm is remarkably simple. At each pixel, we store a real $z$ value that is the distance to the closest triangle rasterized so far. When we rasterize a triangle, as discussed in Section 3.6, we can use the barycentric coordinates to interpolate the depth values of the vertices to each pixel. We only write the rgb and $z$ values into the raster if the $z$ value is closer to the viewer than what is already in that pixel. The z-buffer is first initialized to hold the farthest value that can be represented. For this discussion, we will assume $z$ is positive. If your implementation uses negative $z$, as in the last chapter, then you should change the less-than test below to a greater than test. We use the following operation in place of a straight write of the rgb value:

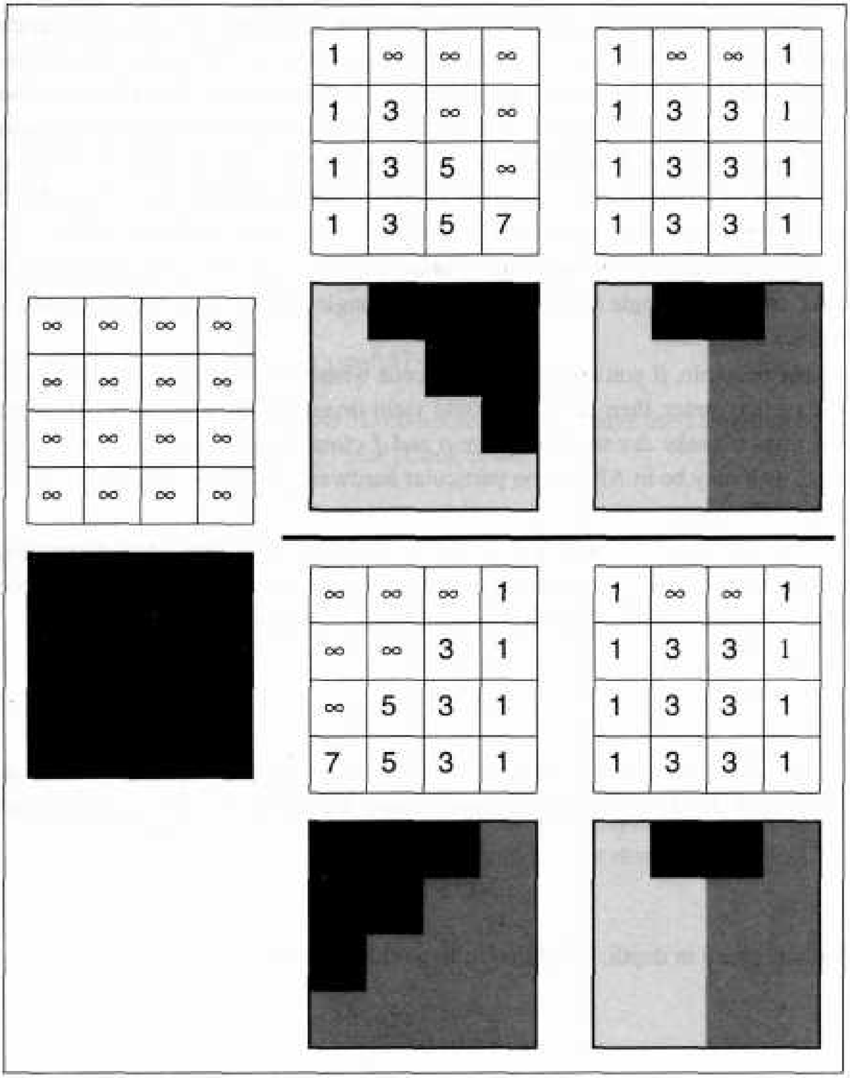
```
function setpixel(int i, int j, rgb c, real z )
if (z < z-buffer($i, j$)) then
  z-buffer($i, j$) = z
  screen($i, j$) = c
```

The simplicity of the algorithm suggests why the z-buffer is well suited to hardware implementation. The computation is straightforward, provided we have fast memory to use for the "z-buffer."

The final result of the z-buffer does not depend on triangle rasterization order, as is shown in Figure 8.8. An exceptional case, however, is when two triangles tie in z-depth for a given pixel. In that case, any of a number of conventions can hold, the most common of which is to assume the last triangle drawn has priority. It can be a better idea to leave the resolution of ties undefined so that optimization can reorder rasterization.

### 8.2.2 Integer Z-Buffer

In practice, the $z$ values stored in the buffer are non-negative integers. This is preferable to true floats because the fast memory needed for the z-buffer is somewhat expensive and is worth keeping to a minimum.



|   |   |   |   |
|---|---|---|---|
| ∞ | ∞ | ∞ | ∞ |
| ∞ | ∞ | ∞ | ∞ |
| ∞ | ∞ | ∞ | ∞ |
| ∞ | ∞ | ∞ | ∞ |

|   |   |   |   |
|---|---|---|---|
| 1 | ∞ | ∞ | ∞ |
| 1 | 3 | ∞ | ∞ |
| 1 | 3 | 5 | ∞ |
| 1 | 3 | 5 | 7 |

|   |   |   |   |
|---|---|---|---|
| 1 | ∞ | ∞ | 1 |
| 1 | 3 | 3 | 1 |
| 1 | 3 | 3 | 1 |
| 1 | 3 | 3 | 1 |

|   |   |   |   |
|---|---|---|---|
| ∞ | ∞ | ∞ | 1 |
| ∞ | ∞ | 3 | 1 |
| ∞ | 5 | 3 | 1 |
| 7 | 5 | 3 | 1 |

|   |   |   |   |
|---|---|---|---|
| 1 | ∞ | ∞ | 1 |
| 1 | 3 | 3 | 1 |
| 1 | 3 | 3 | 1 |
| 1 | 3 | 3 | 1 |

**Figure 8.8.** A z-buffer rasterizing two triangles in each of two possible orders. The first triangle is fully rasterized. The second triangle has every pixel computed, but for three of the pixels the depth-contest is lost, and those pixels are not drawn. The final image is the same regardless.

The use of integers can cause some precision problems. If we use an integer range having $B$ values $\{0, 1, \ldots, B-1\}$, we can map 0 to the near clipping plane $z = n$ and $B-1$ to the far clipping plane $z = f$. Note, that for this discussion, we assume $z$, $n$, and $f$ are positive. This will result in the same results as the negative case, but the details of the argument are easier to follow. We send each $z$ value to a "bucket" with depth $\Delta z = (f-n)/B$. We would not use the integer z-buffer if memory were not a premium, so it is useful to make $B$ as small as possible.

If we allocate $b$ bits to store the z-value, then $B = 2^b$. We need enough bits to make sure any triangle in front of another triangle will have its depth mapped to distinct depth bins.

For example, if you are rendering a scene where triangles have a separation of at least one meter, then $\Delta z < 1$ should yield images without artifacts. There are two ways to make $\Delta z$ smaller: move $n$ and $f$ closer together or increase $b$. If $b$ is fixed, as it may be in APIs or on particular hardware platforms, adjusting $n$ and $f$ is the only option.

The precision of $z$-buffers must be handled with great care when perspective images are created. The value $\Delta z$ above is used *after* the perspective divide. Recall from Section 7.3 that the result of the perspective divide is

$$z = n + f - \frac{fn}{z_w}.$$

The actual bin depth is related to $z_w$, the world depth, rather than $z$, the post-perspective divide depth. We can approximate the bin size by differentiating both sides:

$$\Delta z \approx \frac{fn\Delta z_w}{z_w^2}.$$

Bin sizes vary in depth. The bin size in world space is

$$\Delta z_w \approx \frac{z_w^2 \Delta z}{fn}.$$

Note that the quantity $\Delta z$ is as discussed before. The biggest bin will be for $z' = f$, where

$$\Delta z_w^{\max} \approx \frac{f\Delta z}{n}.$$

Note that choosing $n = 0$, a natural choice if we don't want to lose objects right in front of the eye, will result in an infinitely large bin—a very bad condition. To make $\Delta z_w^{\max}$ as small as possible, we want to minimize $f$ and maximize $n$. Thus, it is always important to choose $n$ and $f$ carefully.

## Frequently Asked Questions

• *Is a uniform distance z-buffer better than the standard one that includes perspective matrix non-linearities?*

It depends. One "feature" of the non-linearities is that the z-buffer has more resolution near the eye and less in the distance. If a level-of-detail system is used, then geometry in the distance is coarser and the "unfairness" of the z-buffer can be a good thing.

• *Is a software z-buffer ever useful?*

Yes. Most of the movies that use 3D computer graphics have used a variant of the software z-buffer developed by Pixar (Cook et al., 1987) .

## Exercises

1. Given $N$ triangles, what is the minimum number of triangles that could be added to a resulting BSP tree? What is the maximum number?
2. Suppose you are designing an integer z-buffer for flight simulation where all of the objects are at least one meter thick, are never closer to the viewer than 4 meters, and may be as far away as 100 km. How many bits are needed in the z-buffer to ensure there are no visibility errors? Suppose that visibility errors only matter near the viewer, i.e., for distances less than 100 meters. How many bits are needed in that case?

# 9

# Surface Shading

To make objects appear to have more volume, it can help to use *shading*, i.e., the surface is "painted" with light. This chapter presents the most common heuristic shading methods. The first two, diffuse and Phong shading, were developed in the 1970s and are available in most graphics libraries. The last, artistic shading, uses artistic conventions to assign color to objects. This creates images reminiscent of technical drawings, which is desirable in many applications.

## 9.1 Diffuse Shading

Many objects in the world have a surface appearance loosely described as "matte," indicating that the object is not at all shiny. Examples include paper, unfinished wood, and dry unpolished stones. To a large degree, such objects do not have a color change with a change in viewpoint. For example, if you stare at a particular point on a piece of paper and move while keeping your gaze fixed on that point, the color at that point will stay relatively constant. Such matte objects can be considered as behaving as *Lambertian* objects. This section discusses how to implement the shading of such objects. A key point is that all formulas in this chapter should be evaluated in world coordinates and not in the warped coordinates after the perspective transform is applied. Otherwise, the angles between normals are changed and the shading will be inaccurate.

### 9.1.1 Lambertian Shading Model

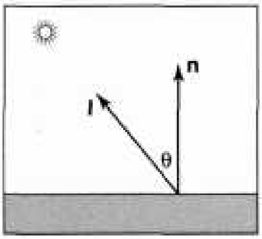


Figure 9.1. The geometry for Lambert's Law. Both **n** and **l** are unit vectors.

A Lambertian object obeys *Lambert's cosine law*, which states that the color $c$ of a surface is proportional to the cosine of the angle between the surface normal and the direction to the light source (Gouraud, 1971):

$$c \propto \cos\theta,$$

or in vector form,

$$c \propto \mathbf{n} \cdot \mathbf{l},$$

where $\mathbf{n}$ and $\mathbf{l}$ are shown in Figure 9.1. Thus, the color on the surface will vary according to the cosine of the angle between the surface normal and the light direction. Note that the vector $\mathbf{l}$ is typically assumed not to depend on the location of the object. That assumption is equivalent to assuming the light is "distant" relative to object size. Such a "distant" light is often called a *directional light*, because its position is specified only by a direction.

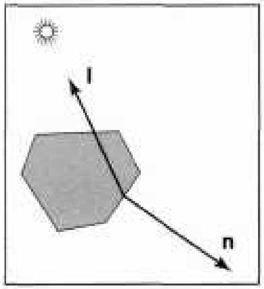


Figure 9.2. When a surface points away from the light, it should receive no light. This case can be verified by checking whether the dot product of **l** and **n** is negative.

A surface can be made lighter or darker by changing the intensity of the light source or the reflectance of the surface. The diffuse reflectance $c_r$ is the fraction of light reflected by the surface. This fraction will be different for different color components. For example, a surface is red if it reflects a higher fraction of red incident light than blue incident light. If we assume surface color is proportional to the light reflected from a surface, then the diffuse reflectance $c_r$—an RGB color—must also be included:

$$c \propto c_r \mathbf{n} \cdot \mathbf{l}. \tag{9.1}$$

The right-hand side of Equation 9.1 is an RGB color with all RGB components in the range $[0, 1]$. We would like to add the effects of light intensity while keeping the RGB components in the range $[0, 1]$. This suggests adding an RGB intensity term $c_l$ which itself has components in the range $[0, 1]$:

$$c = c_r c_l \mathbf{n} \cdot \mathbf{l}. \tag{9.2}$$

This is a very convenient form, but it can produce RGB components for $c$ that are outside the range $[0, 1]$, because the dot product can be negative. The dot product is negative when the surface is pointing away from the light as shown in Figure 9.2.

The "max" function can be added to Equation 9.2 to test for that case:

$$c = c_r c_l \max(0, \mathbf{n} \cdot \mathbf{l}). \tag{9.3}$$

Another way to deal with the "negative" light is to use an absolute value:

$$c = c_r c_l |\mathbf{n} \cdot \mathbf{l}|. \tag{9.4}$$

While Equation 9.4 may seem physically implausible, it actually corresponds to Equation 9.3 with two lights in opposite directions. For this reason it is often called *two-sided* lighting (Figure 9.3).

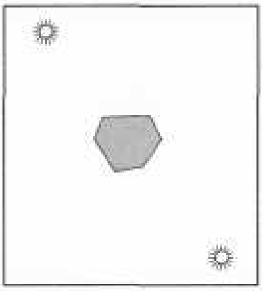


**Figure 9.3.** Using Equation 9.4, the two-sided lighting formula, is equivalent to assuming two opposing light sources of the same color.

### 9.1.2 Ambient Shading

One problem with the diffuse shading of Equation 9.3 is that any point whose normal faces away from the light will be black. In real life, light is reflected all over, and some light is incident from every direction. In addition, there is often skylight giving "ambient" lighting. One way to handle this is to use several light sources. A common trick is to always put a dim source at the eye so that all visible points will receive some light. Another way is to use two-sided lighting as described by Equation 9.4. A more common approach is to add an ambient term (Gouraud, 1971). This is just a constant color term added to Equation 9.3:

$$c = c_r \left( c_a + c_l \max\left(0, \mathbf{n} \cdot \mathbf{l}\right)\right).$$

Intuitively, you can think of the ambient color $c_a$ as the average color of all surfaces in the scene. If you want to ensure that the computed RGB color stays in the range $[0, 1]^3$, then $c_a + c_l \leq (1, 1, 1)$. Otherwise your code should "clamp" RGB values above one to have the value one.

### 9.1.3 Vertex-Based Diffuse Shading

If we apply Equation 9.1 to an object made up of triangles, it will typically have a faceted appearance. Often, the triangles are an approximation to a smooth surface. To avoid the faceted appearance, we can place surface normal vectors at the vertices of the triangles (Phong, 1975), and apply Equation 9.3 at each of the vertices using the normal vectors at the vertices (see Figure 9.4). This will give a color at each triangle vertex, and this color can be interpolated using the barycentric interpolation described in Section 3.6.

One problem with shading at triangle vertices is that we need to get the normals from somewhere. Many models will come with normals supplied. If you tessellate your own smooth model, you can create normals when you create the triangles. If you are presented with a polygonal model that does not have normals at vertices and you want to shade it smoothly, you can compute normals by a variety of heuristic methods. The simplest is to just average the normals of the triangles that share each vertex and use this average normal at the vertex. This

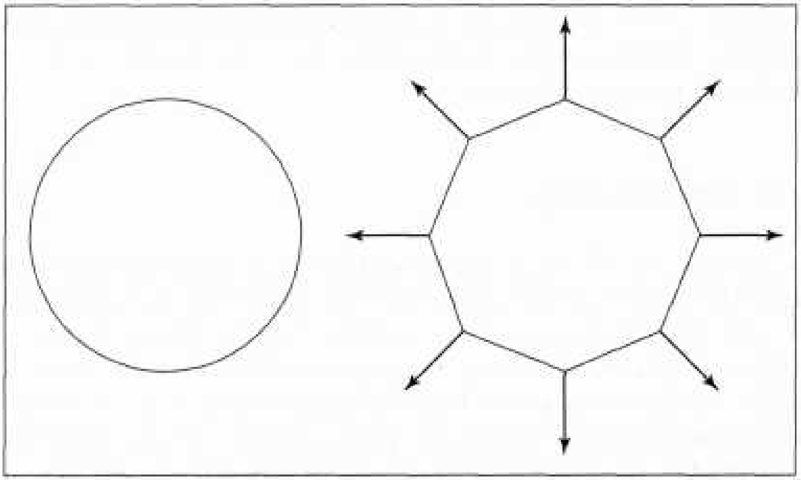


Figure 9.4. A circle (left) is approximated by an octagon (right). Vertex normals record the surface normal of the original curve.

average normal will not automatically be of unit length, so you should convert it to a unit vector before using it for shading.

## 9.2 Phong Shading

Some surfaces are essentially like matte surfaces, but they have *highlights*. Examples of such surfaces include polished tile floors, gloss paint, and whiteboards. Highlights move across a surface as the viewpoint moves. This means that we must add a unit vector **e** toward the eye into our equations. If you look carefully at highlights, you will see that they are really reflections of the light; sometimes these reflections are blurred. The color of these highlights is the color of the light—the surface color seems to have little effect. This is because the reflection occurs at the object's surface, and the light that penetrates the surface and picks up the object's color is scattered diffusely.

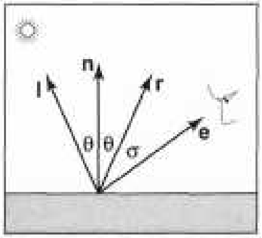


Figure 9.5. The geometry for the Phong illumination model. The eye should see a highlight if $\sigma$ is small.

### 9.2.1 Phong Lighting Model

We want to add a fuzzy "spot" the same color as the light source in the right place. The center of the dot should be drawn where the direction **e** to the eye "lines" up with the natural direction of reflection **r** as shown in Figure 9.5. Here "lines up" is mathematically equivalent to "where $\sigma$ is zero". We would like to have the highlight have some non-zero area, so that the eye sees some highlight wherever $\sigma$ is small.

Given $\mathbf{r}$, we'd like a heuristic function that is bright when $\mathbf{e} = \mathbf{r}$ and falls off gradually when $\mathbf{e}$ moves away from $\mathbf{r}$. An obvious candidate is the cosine of the angle between them:

$$c = c_l(\mathbf{e} \cdot \mathbf{r}),$$

There are two problems with using this equation. The first is that the dot product can be negative. This can be solved computationally with an "if" statement that sets the color to zero when the dot product is negative. The more serious problem is that the highlight produced by this equation is much wider than that seen in real

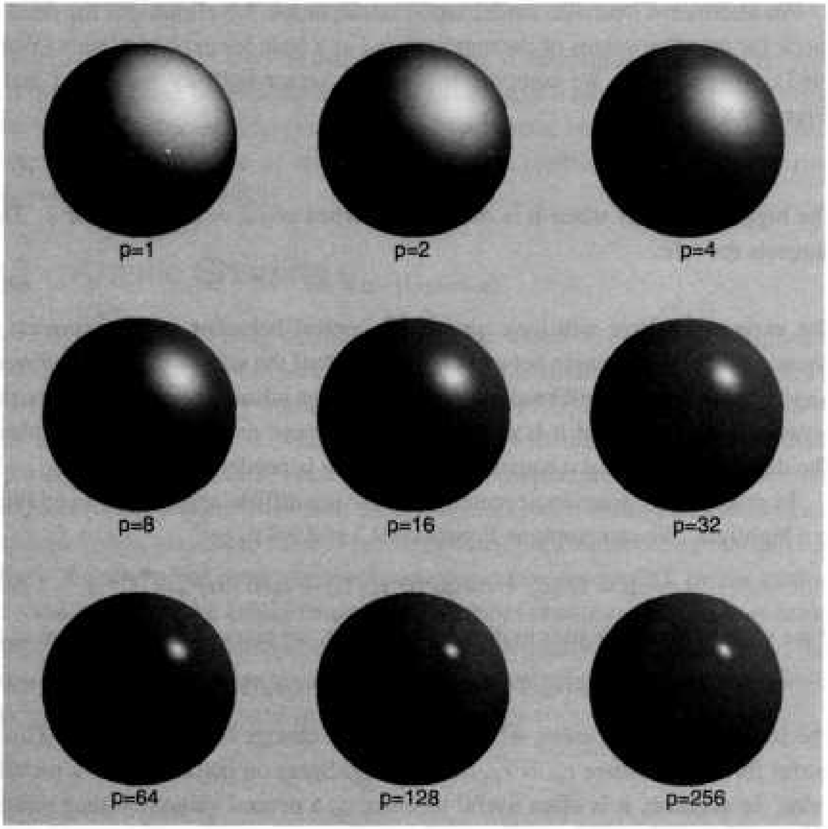


**Figure 9.6.** The effect of the Phong exponent on highlight characteristics. This uses Equation 9.5 for the highlight. There is also a diffuse component, giving the objects a shiny but non-metallic appearance. *Image courtesy of Nate Robins.* (See also Plate IV.)

life. The maximum is in the right place and it is the right color, but it is just too big. We can narrow it without reducing its maximum color by raising to a power:

$$c = c_l \max(0, \mathbf{e} \cdot \mathbf{r})^p. \tag{9.5}$$

Here $p$ is called the *Phong exponent*; it is a positive real number (Phong, 1975). The effect that changing the Phong exponent has on the highlight can be seen in Figure 9.6.

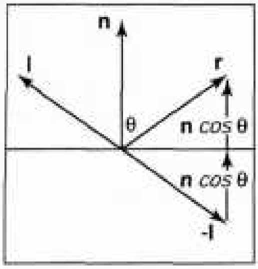


**Figure 9.7.** The geometry for calculating the vector **r**.

To implement Equation 9.5, we first need to compute the unit vector $\mathbf{r}$. Given unit vectors $\mathbf{l}$ and $\mathbf{n}$, $\mathbf{r}$ is the vector $\mathbf{l}$ reflected about $\mathbf{n}$. Figure 9.7 shows that this vector can be computed as

$$\mathbf{r} = -\mathbf{l} + 2(\mathbf{l} \cdot \mathbf{n})\mathbf{n}, \tag{9.6}$$

where the dot product is used to compute $\cos\theta$.

An alternative heuristic model based on Equation 9.5 eliminates the need to check for negative values of the number used as a base for exponentiation (Warn, 1983). Instead of $\mathbf{r}$, we compute $\mathbf{h}$, the unit vector halfway between $\mathbf{l}$ and $\mathbf{e}$ (Figure 9.8):

$$\mathbf{h} = \frac{\mathbf{e} + \mathbf{l}}{\|\mathbf{e} + \mathbf{l}\|}.$$

The highlight occurs when $\mathbf{h}$ is near $\mathbf{n}$, i.e., when $\cos\omega = \mathbf{h} \cdot \mathbf{n}$ is near 1. This suggests the rule:

$$c = c_l (\mathbf{h} \cdot \mathbf{n})^p. \tag{9.7}$$

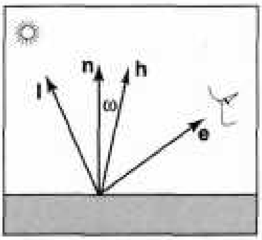


**Figure 9.8.** The unit vector **h** is halfway between **l** and **e**.

The exponent $p$ here will have analogous control behavior to the exponent in Equation 9.5, but the angle between $\mathbf{h}$ and $\mathbf{n}$ is half the size of the angle between $\mathbf{e}$ and $\mathbf{r}$, so the details will be slightly different. The advantage of using the cosine between $\mathbf{n}$ and $\mathbf{h}$ is that it is always positive for eye and light above the plane. The disadvantage is that a square root and divide is needed to compute $\mathbf{h}$.

In practice, we want most materials to have a diffuse appearance in addition to a highlight. We can combine Equations 9.3 and 9.7 to get

$$c = c_r \left(c_a + c_l \max\left(0, \mathbf{n} \cdot \mathbf{l}\right)\right) + c_l (\mathbf{h} \cdot \mathbf{n})^p. \tag{9.8}$$

If we want to allow the user to dim the highlight, we can add a control term $c_p$:

$$c = c_r \left(c_a + c_l \max\left(0, \mathbf{n} \cdot \mathbf{l}\right)\right) + c_l c_p (\mathbf{h} \cdot \mathbf{n})^p. \tag{9.9}$$

The term $c_p$ is a RGB color, which allows us to change highlight colors. This is useful for metals where $c_p = c_r$, because highlights on metal take on a metallic color. In addition, it is often useful to make $c_p$ a neutral value less than one, so that colors stay below one. For example, setting $c_p = 1 - M$ where $M$ is the maximum component of $c_r$ will keep colors below one for one light source and no ambient term.

### 9.2.2 Surface Normal Vector Interpolation

Smooth surfaces with highlights tend to change color quickly compared to Lambertian surfaces with the same geometry. Thus, shading at the normal vectors can generate disturbing artifacts.

These problems can be reduced by interpolating the normal vectors across the polygon and then applying Phong shading at each pixel. This allows you to get good images without making the size of the triangles extremely small. Recall from Chapter 3, that when rasterizing a triangle, we compute barycentric coordinates $(\alpha, \beta, \gamma)$ to interpolate the vertex colors $c_0, c_1, c_2$:

$$c = \alpha c_0 + \beta c_1 + \gamma c_2. \tag{9.10}$$

We can use the same equation to interpolate surface normals $\mathbf{n}_0$, $\mathbf{n}_1$, and $\mathbf{n}_2$:

$$\mathbf{n} = \alpha \mathbf{n}_0 + \beta \mathbf{n}_1 + \gamma \mathbf{n}_2. \tag{9.11}$$

And Equation 9.9 can then be evaluated for the $\mathbf{n}$ computed at each pixel. Note that the $\mathbf{n}$ resulting from Equation 9.11 is usually not a unit normal. Better visual results will be achieved if it is converted to a unit vector before it is used in shading computations. This type of normal interpolation is often called *Phong normal interpolation* (Phong, 1975).

## 9.3 Artistic Shading

The Lambertian and Phong shading methods are based on heuristics designed to imitate the appearance of objects in the real world. Artistic shading is designed to mimic drawings made by human artists (Yessios, 1979; Dooley & Cohen, 1990; Saito & Takahashi, 1990; L. Williams, 1991). Such shading seems to have advantages in many applications. For example, auto manufacturers hire artists to draw diagrams for car owners' manuals. This is more expensive than using much more "realistic" photographs, so there is probably some intrinsic advantage to the techniques of artists when certain types of communication are needed. In this section, we show how to make subtly shaded line drawings reminiscent of human-drawn images. Creating such images is often called *non-photorealistic rendering*, but we will avoid that term because many non-photorealistic techniques are used for efficiency that are not related to any artistic practice.

### 9.3.1 Line Drawing

The most obvious thing we see in human drawings that we don't see in real life is *silhouettes*. When we have a set of triangles with shared edges, we should draw an edge as a silhouette when one of the two triangles sharing an edge faces toward the viewer, and the other triangle faces away from the viewer. This condition can be tested for two normals $\mathbf{n}_0$ and $\mathbf{n}_1$ by

$$\text{draw silhouette if } (\mathbf{e} \cdot \mathbf{n}_0)(\mathbf{e} \cdot \mathbf{n}_1) \leq 0.$$

Here $\mathbf{e}$ is a vector from the edge to the eye. This can be any point on the edge or either of the triangles. Alternatively, if $f_i(\mathbf{p}) = 0$ are the implicit plane equations for the two triangles, the test can be written

$$\text{draw silhouette if } \; f_0(\mathbf{e}) f_1(\mathbf{e}) \leq 0.$$

We would also like to draw visible edges of a polygonal model. To do this, we can use either of the hidden surface methods of Chapter 8 for drawing in the background color, and then draw the outlines of each triangle in black. This, in fact, will also capture the silhouettes. Unfortunately, if the polygons represent a smooth surface, we really don't want to draw most of those edges. However, we might want to draw all *creases* where there really is a corner in the geometry. We can test for creases by using a heuristic threshold:

$$\text{draw crease if } (\mathbf{n}_0 \cdot \mathbf{n}_1) \leq \text{threshold}.$$

This combined with the silhouette test will give nice-looking line drawings.

### 9.3.2 Cool-to-Warm Shading

When artists shade line drawings, they often use low intensity shading to give some impression of curve to the surface and to give colors to objects (Gooch, Gooch, Shirley, & Cohen, 1998). Surfaces facing in one direction are shaded with a cool color, such as a blue, and surfaces facing in the opposite direction are shaded with a warm color, such as orange. Typically these colors are not very saturated and are also not dark. That way, black silhouettes show up nicely. Overall this gives a cartoon-like effect. This can be achieved by setting up a direction to a "warm" light $\mathbf{l}$ and using the cosine to modulate color, where the warmth constant $k_w$ is defined on $[0, 1]$:

$$k_w = \frac{1 + \mathbf{n} \cdot \mathbf{l}}{2}.$$

The color $c$ is then just a linear blend of the cool color $c_c$ and the warm color $c_w$:

$$c = k_w c_w + (1 - k_w) c_c.$$

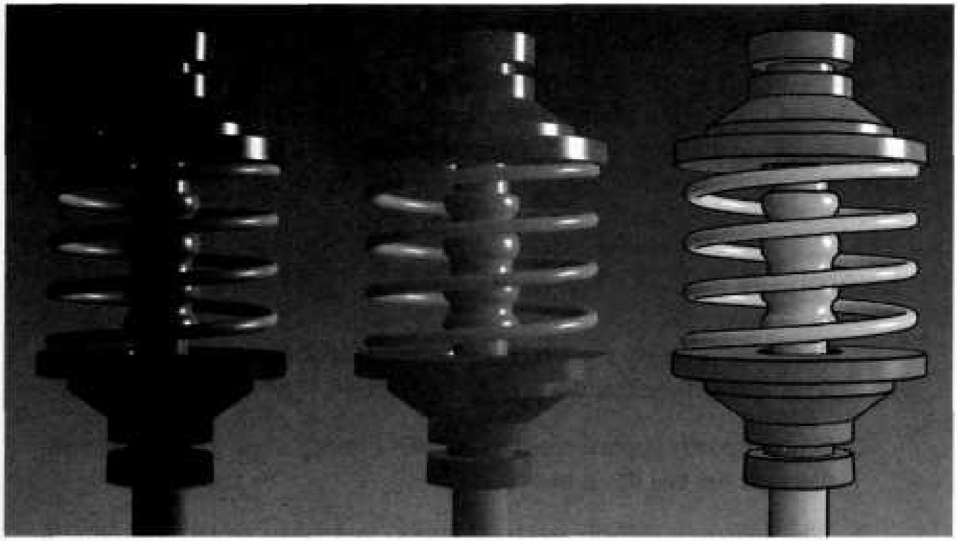


**Figure 9.9.** Left: a Phong-illuminated image. Middle: cool-to-warm shading is not useful without silhouettes. Right: cool-to-warm shading plus silhouettes. *Image courtesy Amy Gooch.* (See also Plate V.)

There are many possible $c_w$ and $c_b$ that will produce reasonable looking results. A good starting place for a guess is

$$
\begin{aligned}
c_c &= (0.4, 0.4, 0.7), \\
c_c &= (0.8, 0.6, 0.6).
\end{aligned}
$$

Figure 9.9 shows a comparison between traditional Phong lighting and this type of artistic shading.

## Frequently Asked Questions

- All of the shading in this chapter seems like enormous hacks. Is that true?

Yes. However, they are carefully designed hacks that have proven useful in practice. In the long run, we will probably have better-motivated algorithms that include physics, psychology, and tone-mapping. However, the improvements in image quality will probably be incremental.

- **I hate calling pow(). Is there a way to avoid it when doing Phong lighting?**

A simple way is to only have exponents that are themselves a power of two, i.e., 2, 4, 8, 16, .... In practice, this is not a problematic restriction for most applications. A look-up table is also possible, but will often not give a large speed-up.

## Exercises

1. The moon is poorly approximated by diffuse or Phong shading. What observations tell you that this is true?
2. Velvet is poorly approximated by diffuse or Phong shading. What observations tell you that this is true?
3. Why do most highlights on plastic objects look white, while those on gold metal look gold?

# 10

# Ray Tracing

*Ray tracing* is a method to produce realistic images; it determines visible surfaces in an image at the pixel level (Appel, 1968; Kay & Greenberg, 1979; Whitted, 1980). Unlike the z-buffer and BSP tree, ray tracing operates pixel-by-pixel rather than primitive-by-primitive. This tends to make ray tracing relatively slow for scenes with large objects in screen space. However, it has a variety of nice features which often make it the right choice for batch rendering and even for some interactive applications.

Ray tracing's primary benefit is that it is relatively straightforward to compute shadows and reflections. In addition, ray tracing is well suited to "walkthroughs" of extremely large models due to advanced ray tracing's low asymptotic time complexity which makes up for the required preprocessing of the model (Snyder & Barr, 1987; Muuss, 1995; Parker, Martin, et al., 1999; Wald, Slusallek, Benthin, & Wagner, 2001).

In an interactive 3D program implemented in a conventional z-buffer environment, it is often useful to be able to select an object using a mouse. The mouse is clicked in pixel $(i, j)$ and the "picked" object is whatever object is "seen" through that pixel. If the rasterization process includes an object identification buffer, this is just a matter of looking up the value in pixel $(i, j)$ of that buffer. However, if that buffer is not available, we can solve the problem of which object is visible via brute force geometrical computation using a "ray intersection test." In this way, ray tracing is useful also to programmers who use only standard graphics APIs.

This chapter also discusses *distribution ray tracing* (Cook, Porter, & Carpenter, 1984), where multiple random rays are sent through each pixel in an image to simultaneously solve the antialiasing, soft shadow, fuzzy reflection, and depth-of-field problems.

## 10.1 The Basic Ray-Tracing Algorithm

The simplest use of ray tracing is to produce images similar to those produced by the z-buffer and BSP-tree algorithms. Fundamentally, those methods make sure the appropriate object is "seen" through each pixel,and that the pixel color is shaded based on that object's material properties, the surface normal seen through that pixel, and the light geometry.

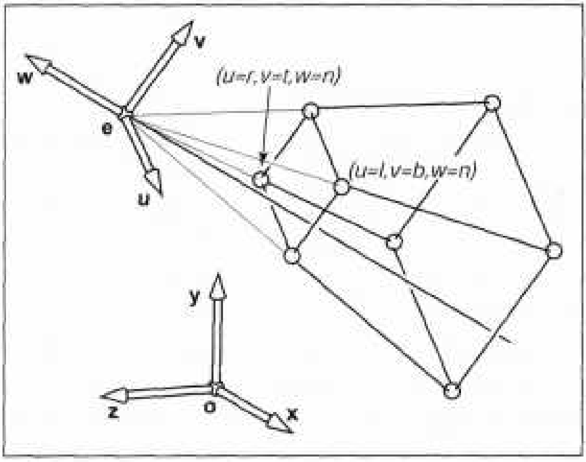


**Figure 10.1.** The 3D window we look through is the same as in Chapter 7. The borders of the window have simple coordinates in the *uvw*-coordinate system with respect to origin **e**.

Figure 10.1 shows the basic viewing geometry for ray tracing, which is the same as we saw earlier in Chapter 7. The geometry is aligned to a *uvw* coordinate system with the origin at the eye location **e**. The key idea in ray tracing is to identify locations on the view plane at $w = n$ that correspond to pixel centers, as shown in Figure 10.2. A "ray," really just a directed 3D line, is then sent from **e** to that point. We then "gaze" in the direction of the ray to see the first object seen in that direction. This is shown in Figure 10.3, where the ray intersects two triangles, but only the first triangle hit, $T_2$, is returned.

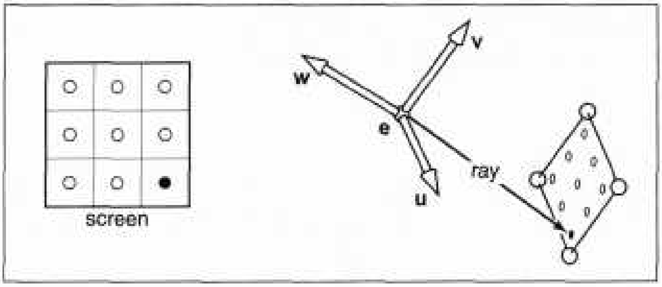


**Figure 10.2.** The sample points on the screen are mapped to a similar array on the 3D window. A viewing ray is sent to each of these locations.

The structure of the basic ray tracing program is:

```
Compute $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ basis vectors
for each pixel do
    compute viewing ray
    find first object hit by ray and its surface normal $\mathbf{n}$
    set pixel color to value based on material, light, and $\mathbf{n}$
```

The pixel color can be computed using the shading equations of the last chapter.

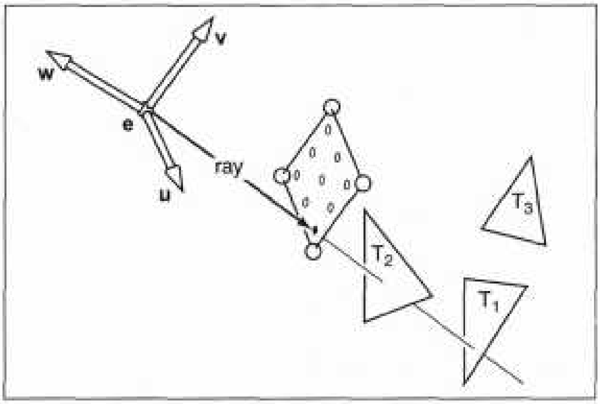


**Figure 10.3.** The ray is "traced" into the scene and the first object hit is the one seen through the pixel. In this case, the triangle $T_2$ is returned.

## 10.2 Computing Viewing Rays

First we need to determine a mathematical representation for a ray. A ray is really just an origin point and a propagation direction; a 3D parametric line is ideal for

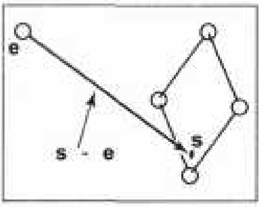


**Figure 10.4.** The ray from the eye to a point on the screen.

this. As discussed in Section 2.8.1, the 3D parametric line from the eye $\mathbf{e}$ to a point $\mathbf{s}$ on the screen (see Figure 10.4) is given by

$$\mathbf{p}(t) = \mathbf{e} + t(\mathbf{s} - \mathbf{e}).$$

This should be interpreted as, "we advance from $\mathbf{e}$ along the vector $(\mathbf{s} - \mathbf{e})$ a fractional distance $t$ to find the point $\mathbf{p}$." So given $t$, we can determine a point $\mathbf{p}$. Note that $\mathbf{p}(0) = \mathbf{e}$, and $\mathbf{p}(1) = \mathbf{s}$. Also note that for positive $t$, if $t_1 < t_2$, then $\mathbf{p}(t_1)$ is closer to the eye than $\mathbf{p}(t_2)$. Also, if $t < 0$, then $\mathbf{p}(t)$ is "behind" the eye. These facts will be useful when we search for the closest object hit by the ray that is not behind the eye. Note that we are overloading the variable $t$ here which is also used for the top of the screen's $v$-coordinate.

To compute a viewing ray, we need to know $\mathbf{e}$ (which is given) and $\mathbf{s}$. Finding $\mathbf{s}$ may look somewhat difficult. In fact, it is relatively straightforward using the same transform machinery we used for viewing in the context of projecting lines and triangles.

First, we find the coordinates of $\mathbf{s}$ in the $uvw$-coordinate system with origin $\mathbf{e}$. For all points on the screen, $w_s = n$ as shown in Figure 10.2. The $uv$-coordinates are found by the windowing transform that takes $[-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5]$ to $[l, r] \times [b, t]$:

$$\begin{aligned} u_s &= l + (r - l)\frac{i + 0.5}{n_x}, \\ v_s &= b + (t - b)\frac{j + 0.5}{n_y}, \end{aligned}$$

where $(i, j)$ are the pixel indices. This gives us $\mathbf{s}$ in $uvw$-coordinates. By definition, we can convert to canonical coordinates:

$$\mathbf{s} = \mathbf{e} + u_s\mathbf{u} + v_s\mathbf{v} + w_s\mathbf{w}. \tag{10.1}$$

Alternatively, we could use the matrix form (Equation 6.8):

$$\begin{bmatrix} x_s \\ y_s \\ z_s \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & x_e \\ 0 & 1 & 0 & y_e \\ 0 & 0 & 1 & z_e \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & x_v & x_w & 0 \\ y_u & y_v & y_w & 0 \\ z_u & z_v & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_s \\ v_s \\ w_s \\ 1 \end{bmatrix}, \tag{10.2}$$

which is just the matrix form of Equation 10.1. We can compose this with the windowing transform in matrix form if we wished, but this is probably not worth doing unless you like the matrix form of equations better.

## 10.3 Ray-Object Intersection

Given a ray $\mathbf{e} + t\mathbf{d}$, we want to find the first intersection with any object where $t > 0$. It will later prove useful to solve a slightly more general problem of finding the first intersection in the interval $[t_0, t_1]$, and using $[0, \infty)$ for viewing rays. We solve this for both spheres and triangles in this section. In the next section, multiple objects are discussed.

### 10.3.1 Ray-Sphere Intersection

Given a ray $\mathbf{p}(t) = \mathbf{e} + t\mathbf{d}$ and an implicit surface $f(\mathbf{p}) = 0$, we'd like to know where they intersect. The intersection points occur when points on the ray satisfy the implicit equation

$$f(\mathbf{p}(t)) = 0.$$

This is just

$$f(\mathbf{e} + t\mathbf{d}) = 0.$$

A sphere with center $\mathbf{c} = (x_c, y_c, z_c)$ and radius $R$ can be represented by the implicit equation

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 - R^2 = 0.$$

We can write this same equation in vector form:

$$(\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) - R^2 = 0.$$

Any point $\mathbf{p}$ that satisfies this equation is on the sphere. If we plug points on the ray $\mathbf{p}(t) = \mathbf{e} + t\mathbf{d}$ into this equation, we can solve for the values of $t$ on the ray that yield points on the sphere:

$$(\mathbf{e} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{e} + t\mathbf{d} - \mathbf{c}) - R^2 = 0.$$

Rearranging terms yields

$$(\mathbf{d} \cdot \mathbf{d})t^2 + 2\mathbf{d} \cdot (\mathbf{e} - \mathbf{c})t + (\mathbf{e} - \mathbf{c}) \cdot (\mathbf{e} - \mathbf{c}) - R^2 = 0.$$

Here, everything is known except the parameter $t$, so this is a classic quadratic equation in $t$, meaning it has the form

$$At^2 + Bt + C = 0.$$

The solution to this equation is discussed in Section 2.2. The term under the square root sign in the quadratic solution, $B^2 - 4AC$, is called the *discriminant* and tells us how many real solutions there are. If the discriminant is negative, its square root is imaginary and there are no intersections between the sphere and the line. If the discriminant is positive, there are two solutions: one solution where the ray enters the sphere and one where it leaves. If the discriminant is zero, the ray grazes the sphere touching it at exactly one point. Plugging in the actual terms for the sphere and eliminating the common factors of two, we get

$$t = \frac{-\mathbf{d}\cdot(\mathbf{e}-\mathbf{c}) \pm \sqrt{(\mathbf{d}\cdot(\mathbf{e}-\mathbf{c}))^2 - (\mathbf{d}\cdot\mathbf{d})\left((\mathbf{e}-\mathbf{c})\cdot(\mathbf{e}-\mathbf{c}) - R^2\right)}}{(\mathbf{d}\cdot\mathbf{d})}.$$

In an actual implementation, you should first check the value of the discriminant before computing other terms. If the sphere is used only as a bounding object for more complex objects, then we need only determine whether we hit it; checking the discriminant suffices.

As discussed in Section 2.7.1, the normal vector at point $\mathbf{p}$ is given by the gradient $\mathbf{n} = 2(\mathbf{p}-\mathbf{c})$. The unit normal is $(\mathbf{p}-\mathbf{c})/R$.

### 10.3.2 Ray-Triangle Intersection

There are many algorithms for computing ray-triangle intersections. We will use the form that uses barycentric coordinates for the parametric plane containing the triangle, because it requires no long-term storage other than the vertices of the triangle (Snyder & Barr, 1987).

To intersect a ray with a parametric surface, we set up a system of equations where the Cartesian coordinates all match:

$$\begin{aligned} x_e + tx_d &= f(u,v), \\ y_e + ty_d &= g(u,v), \\ z_e + tz_d &= h(u,v). \end{aligned}$$

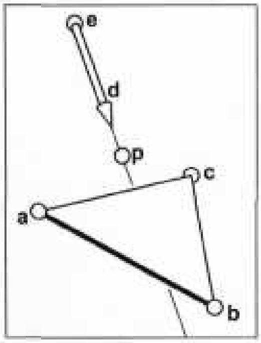


**Figure 10.5.** The ray hits the plane containing the triangle at point **p**.

Here, we have three equations and three unknowns ($t$, $u$, and $v$), so we can solve numerically for the unknowns. If we are lucky, we can solve for them analytically.

In the case where the parametric surface is a parametric plane, the parametric equation can be written in vector form as discussed in Section 2.11.2. If the vertices of the triangle are $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$, then the intersection will occur when

$$\mathbf{e} + t\mathbf{d} = \mathbf{a} + \beta(\mathbf{b}-\mathbf{a}) + \gamma(\mathbf{c}-\mathbf{a}). \tag{10.3}$$

The hitpoint $\mathbf{p}$ will be at $\mathbf{e} + t\mathbf{d}$ as shown in Figure 10.5. Again, from Section 2.11.2, we know the hitpoint is in the triangle if and only if $\beta > 0$, $\gamma > 0$, and $\beta + \gamma < 1$. Otherwise, it hits the plane outside the triangle. If there are no solutions, either the triangle is degenerate or the ray is parallel to the plane containing the triangle.

To solve for $t$, $\beta$, and $\gamma$ in Equation 10.3, we expand it from its vector form into the three equations for the three coordinates:

$$\begin{aligned}
x_e + t x_d &= x_a + \beta(x_b - x_a) + \gamma(x_c - x_a),\\
y_e + t y_d &= y_a + \beta(y_b - y_a) + \gamma(y_c - y_a),\\
z_e + t z_d &= z_a + \beta(z_b - z_a) + \gamma(z_c - z_a).
\end{aligned}$$

This can be rewritten as a standard linear equation:

$$\begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_e \\ y_a - y_e \\ z_a - z_e \end{bmatrix}.$$

The fastest classic method to solve this $3 \times 3$ linear system is *Cramer's rule*. This gives us the solutions

$$\beta = \frac{\begin{vmatrix} x_a - x_e & x_a - x_c & x_d \\ y_a - y_e & y_a - y_c & y_d \\ z_a - z_e & z_a - z_c & z_d \end{vmatrix}}{|\mathbf{A}|},$$

$$\gamma = \frac{\begin{vmatrix} x_a - x_b & x_a - x_e & x_d \\ y_a - y_b & y_a - y_e & y_d \\ z_a - z_b & z_a - z_e & z_d \end{vmatrix}}{|\mathbf{A}|},$$

$$t = \frac{\begin{vmatrix} x_a - x_b & x_a - x_c & x_a - x_e \\ y_a - y_b & y_a - y_c & y_a - y_e \\ z_a - z_b & z_a - z_c & z_a - z_e \end{vmatrix}}{|\mathbf{A}|},$$

where the matrix $\mathbf{A}$ is

$$\mathbf{A} = \begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix},$$

and $|\mathbf{A}|$ denotes the determinant of $\mathbf{A}$. The $3 \times 3$ determinants have common subterms that can be exploited. Looking at the linear systems with dummy variables

$$\begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} j \\ k \\ l \end{bmatrix},$$

Cramer's rule gives us

$$\beta = \frac{j(ei - hf) + k(gf - di) + l(dh - eg)}{M},$$

$$\gamma = \frac{i(ak - jb) + h(jc - al) + g(bl - kc)}{M},$$

$$t = -\frac{f(ak - jb) + e(jc - al) + d(bl - kc)}{M},$$

where

$$M = a(ei - hf) + b(gf - di) + c(dh - eg).$$

We can reduce the number of operations by reusing numbers such as "*ei-minus-hf*."

The algorithm for the ray-triangle intersection for which we need the linear solution can have some conditions for early termination. Thus, the function should look something like:

```
boolean raytri (ray $\mathbf{r}$, vector3 $\mathbf{a}$, vector3 $\mathbf{b}$, vector3 $\mathbf{c}$, interval $[t_0, t_1]$)
compute $t$
**if** $(t < t_0)$ or $(t > t_1)$ **then**
  **return** false
compute $\gamma$
**if** $(\gamma < 0)$ or $(\gamma > 1)$ **then**
  **return** false
compute $\beta$
**if** $(\beta < 0)$ or $(\beta > 1 - \gamma)$ **then**
  **return** false
**return** true
```

### 10.3.3 Ray-Polygon Intersection

Given a polygon with $m$ vertices $\mathbf{p}_1$ through $\mathbf{p}_m$ and surface normal $\mathbf{n}$, we first compute the intersection points between the ray $\mathbf{e} + t\mathbf{d}$ and the plane containing the polygon with implicit equation

$$(\mathbf{p} - \mathbf{p}_1) \cdot \mathbf{n} = 0.$$

We do this by setting $\mathbf{p} = \mathbf{e} + t\mathbf{d}$ and solving for $t$ to get

$$t = \frac{(\mathbf{p}_1 - \mathbf{e}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}.$$

This allows us to compute $\mathbf{p}$. If $\mathbf{p}$ is inside the polygon, then the ray hits it, and otherwise it does not.

We can answer the question of whether $\mathbf{p}$ is inside the polygon by projecting the point and polygon vertices to the $xy$ plane and answering it there. The easiest way to do this is to send any 2D ray out from $\mathbf{p}$ and to count the number of intersections between that ray and the boundary of the polygon (Sutherland et al., 1974; Glassner, 1989). If the number of intersections is odd, then the point is inside the polygon, and otherwise it is not. This is true, because a ray that goes in must go out, thus creating a pair of intersections. Only a ray that starts inside will not create such a pair. To make computation simple, the 2D ray may as well propagate along the $x$-axis:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \end{bmatrix} + s \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

It is straightforward to compute the intersection of that ray with the edges such as $(x_1, y_1, x_2, y_2)$ for $s \in (0, \infty)$.

A problem arises, however, for polygons whose projection into the $xy$ plane is a line. To get around this, we can choose among the $xy$, $yz$, or $zx$ planes for whichever is best. If we implement our points to allow an indexing operation, e.g., $\mathbf{p}(0) = x_p$ then this can be accomplished as follows:

```
**if** (abs($z_n$) > abs($x_n$)) and (abs($z_n$) > abs($y_n$)) **then**
    index0 = 0
    index1 = 1
**else if** (abs($y_n$) > abs ($x_n$)) **then**
    index0 = 0
    index1 = 2
**else**
    index0 = 1
    index1 = 2
```

Now, all computations can use $\mathbf{p}$(index0) rather than $x_p$, and so on.

## 10.4 A Ray-Tracing Program

We now know how to generate a viewing ray for a given pixel and how to find the intersection with one object. This can be easily extended to a program that produces images similar to the z-buffer or BSP-tree codes of earlier chapters:

```
for each pixel do
  compute viewing ray
  if (ray hits an object with $t \in [0, \infty)$) then
    Compute $\mathbf{n}$
    Evaluate lighting equation and set pixel to that color
  else
    set pixel color to background color
```

Here the statement "if ray hits an object..." can be implemented as a function that tests for hits in the interval $t \in [t_0, t_1]$:

```
hit = false
for each object $\mathbf{o}$ do
  if (object is hit at ray parameter $t$ and $t \in [t_0, t_1]$) then
    hit = true
    hitobject = $\mathbf{o}$
    $t_1 = t$
return hit
```

In an actual implementation, you will need to somehow return either a reference to the object that is hit or at least its normal vector and material properties. This is often done by passing a record/structure with such information. In an object-oriented implementation, it is a good idea to have a class called something like *surface* with derived classes *triangle*, *sphere*, *surface-list*, etc. Anything that a ray can intersect would be under that class. The ray tracing program would then have one reference to a "surface" for the whole model, and new types of objects and efficiency structures can be added transparently.

### 10.4.1 Object-Oriented Design for a Ray-Tracing Program

As mentioned earlier, the key class hierarchy in a ray tracer are the geometric objects that make up the model. These should be subclasses of some geometric object class, and they should support a *hit* function (Kirk & Arvo, 1988). To avoid confusion from use of the word "object," *surface* is the class name often used. With such a class, you can create a ray tracer that has a general interface that assumes little about modeling primitives and debug it using only spheres. An important point is that anything that can be "hit" by a ray should be part of this class hierarchy, e.g., even a collection of surfaces should be considered a subclass of the surface class. This includes efficiency structures, such as bounding volume hierarchies; they can be hit by a ray, so they are in the class.

For example, the "abstract" or "base" class would specify the hit function as well as a bounding box function that will prove useful later:

```
class surface
virtual bool hit(ray $\mathbf{e} + t\mathbf{d}$, real $t_0$, real $t_1$, hit-record rec)
virtual box bounding-box()
```

Here $(t_0, t_1)$ is the interval on the ray where hits will be returned, and rec is a record that is passed by reference; it contains data such as the $t$ at intersection when hit returns true. The type box is a 3D "bounding box", that is two points that define an axis-aligned box that encloses the surface. For example, for a sphere, the function would be implemented by:

```
box sphere::bounding-box()
vector3 min = center - vector3(radius,radius,radius)
vector3 max = center + vector3(radius,radius,radius)
**return** box(min, max)
```

Another class that is useful is material. This allows you to abstract the material behavior and later add materials transparently. A simple way to link objects and materials is to add a pointer to a material in the surface class, although more programmable behavior might be desirable. A big question is what to do with textures; are they part of the material class or do they live outside of the material class? This will be discussed more in Chapter 11.

## 10.5 Shadows

Once you have a basic ray tracing program, shadows can be added very easily. Recall from Chapter 9 that light comes from some direction $\mathbf{l}$. If we imagine ourselves at a point $\mathbf{p}$ on a surface being shaded, the point is in shadow if we "look" in direction $\mathbf{l}$ and see an object. If there are no objects, then the light is not blocked.

This is shown in Figure 10.6, where the ray $\mathbf{p} + t\mathbf{l}$ does not hit any objects and is thus not in shadow. The point $\mathbf{q}$ is in shadow because the ray $\mathbf{q} + t\mathbf{l}$ does hit an object. The vector $\mathbf{l}$ is the same for both points because the light is "far" away. This assumption will later be relaxed. The rays that determine in or out of shadow are called *shadow rays* to distinguish them from viewing rays.

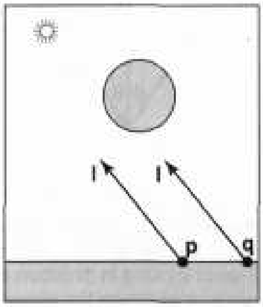


**Figure 10.6.** The point **p** is not in shadow while the point **q** is in shadow.

To get the algorithm for shading, we add an if statement to determine whether the point is in shadow. In a naive implementation, the shadow ray will check for $t \in [0, \infty)$, but because of numerical imprecision, this can result in an inter-

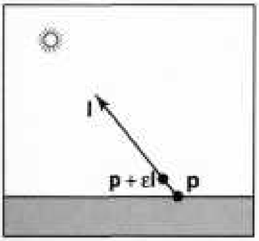


**Figure 10.7.** By testing in the interval starting at $\epsilon$, we avoid numerical imprecision causing the ray to hit the surface **p** is on.

section with the surface on which $\mathbf{p}$ lies. Instead, the usual adjustment to avoid that problem is to test for $t \in [\epsilon, \infty)$ where $\epsilon$ is some small positive constant (Figure 10.7).

If we implement shadow rays for Phong lighting with Equation 9.9 then we have:

```
**function** raycolor( ray $\mathbf{e} + t\mathbf{d}$, real $t_0$, real $t_1$ )
hit-record rec, srec
**if** (scene→hit($\mathbf{e} + t\mathbf{d}$, $t_0$, $t_1$, rec)) **then**
    $\mathbf{p} = \mathbf{e} + \text{rec}.t\mathbf{d}$
    color $c = \text{rec}.c_r\ \text{rec}.c_a$
    **if** (not scene→hit($\mathbf{p} + s\mathbf{l}$, $\epsilon$, $\infty$, srec)) **then**
        vector3 $\mathbf{h} = \text{normalized}(\text{normalized}(\mathbf{l}) + \text{normalized}(-\mathbf{d}))$
        $c = c + \text{rec}.c_r\ c_l \max(0, \text{rec}.\mathbf{n} \cdot \mathbf{l}) + c_l \text{rec}.c_p (\mathbf{h} \cdot \text{rec}.\mathbf{n})^{\text{rec}.p}$
    **return** $c$
**else**
    **return** background-color
```

Note that the ambient color is added in either case. If there are multiple light sources, we can send a shadow ray and evaluate the diffuse/phong terms for each light. The code above assumes that $\mathbf{d}$ and $\mathbf{l}$ are not necessarily unit vectors. This is crucial for $\mathbf{d}$, in particular, if we wish to cleanly add *instancing* later.

## 10.6 Specular Reflection

It is straightforward to add *specular* reflection to a ray-tracing program. The key observation is shown in Figure 10.8 where a viewer looking from direction $\mathbf{e}$ sees what is in direction $\mathbf{r}$ as seen from the surface. The vector $\mathbf{r}$ is found using a variant of the Phong lighting reflection Equation 9.6. There are sign changes because the vector $\mathbf{d}$ points toward the surface in this case, so,

$$\mathbf{r} = \mathbf{d} - 2(\mathbf{d} \cdot \mathbf{n})\mathbf{n}, \tag{10.4}$$

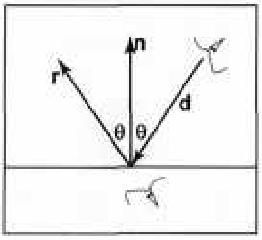


**Figure 10.8.** When looking into a perfect mirror, the viewer looking in direction **d** will see whatever the viewer "below" the surface would see in direction **r**.

In the real world, some energy is lost when the light reflects from the surface, and this loss can be different for different colors. For example, gold reflects yellow more efficiently than blue, so it shifts the colors of the objects it reflects. This can be implemented by adding a recursive call in *raycolor*:

```
color $c = c + c_s$raycolor($\mathbf{p} + s\mathbf{r}$, $\epsilon$, $\infty$)
```

where $c_s$ is the specular RGB color. We need to make sure we test for $s \in [\epsilon, \infty)$

for the same reason as we did with shadow rays; we don't want the reflection ray to hit the object that generates it.

The problem with the recursive call above is that it may never terminate. For example, if a ray starts inside a room, it will bounce forever. This can be fixed by adding a maximum recursion depth. The code will be more efficient if a reflection ray is generated only if $c_s$ is not zero (black).

## 10.7 Refraction

Another type of specular object is a *dielectric*—a transparent material that refracts light. Diamonds, glass, water, and air are dielectrics. Dielectrics also filter light; some glass filters out more red and blue light than green light, so the glass takes on a green tint. When a ray travels from a medium with refractive index $n$ into one with a refractive index $n_t$, some of the light is transmitted, and it bends. This is shown for $n_t > n$ in Figure 10.9. Snell's law tells us that

$$n \sin\theta = n_t \sin\phi.$$

Computing the sine of an angle between two vectors is usually not as convenient as computing the cosine which is a simple dot product for the unit vectors such as we have here. Using the trigonometric identity $\sin^2\theta + \cos^2\theta = 1$, we can derive a refraction relationship for cosines:

$$\cos^2\phi = 1 - \frac{n^2\left(1 - \cos^2\theta\right)}{n_t^2}.$$

Note that if $n$ and $n_t$ are reversed, then so are $\theta$ and $\phi$ as shown on the right of Figure 10.9.

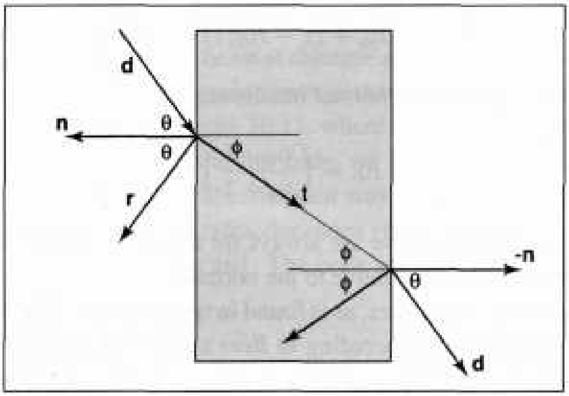


**Figure 10.9.** Snell's Law describes how the angle $\phi$ depends on the angle $\theta$ and the refractive indices of the object and the surrounding medium.

To convert $\sin\phi$ and $\cos\phi$ into a 3D vector, we can set up a 2D orthonormal basis in the plane of $\mathbf{n}$ and $\mathbf{d}$.

From Figure 10.10, we can see that $\mathbf{n}$ and $\mathbf{b}$ form an orthonormal basis for the plane of refraction. By definition, we can describe $\mathbf{t}$ in terms of this basis:

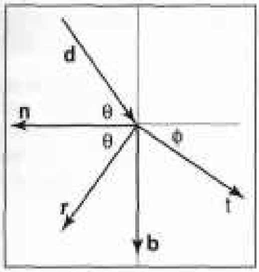


**Figure 10.10.** The vectors **n** and **b** form a 2D orthonormal basis that is parallel to the transmission vector **t**.

$$\mathbf{t} = \sin\phi\mathbf{b} - \cos\phi\mathbf{n}.$$

Since we can describe $\mathbf{d}$ in the same basis, and $\mathbf{d}$ is known, we can solve for $\mathbf{b}$:

$$\mathbf{d} = \sin\theta\mathbf{b} - \cos\theta\mathbf{n},$$
$$\mathbf{b} = \frac{\mathbf{d} + \mathbf{n}\cos\theta}{\sin\theta}.$$

This means that we can solve for $\mathbf{t}$ with known variables:

$$\begin{aligned}\mathbf{t} &= \frac{n\,(\mathbf{d} + \mathbf{n}\cos\theta))}{n_t} - \mathbf{n}\cos\phi \\ &= \frac{n\,(\mathbf{d} - \mathbf{n}(\mathbf{d}\cdot\mathbf{n}))}{n_t} - \mathbf{n}\sqrt{1 - \frac{n^2\,(1 - (\mathbf{d}\cdot\mathbf{n})^2)}{n_t^2}}.\end{aligned}$$

Note that this equation works regardless of which of $n$ and $n_t$ is larger. An immediate question is, "What should you do if the number under the square root is negative?" In this case, there is no refracted ray and all of the energy is reflected. This is known as *total internal reflection*, and it is responsible for much of the rich appearance of glass objects.

The reflectivity of a dielectric varies with the incident angle according to the *Fresnel equations*. A nice way to implement something close to the Fresnel equations is to use the *Schlick approximation* (Schlick, 1994a),

$$R(\theta) = R_0 + (1 - R_0)\,(1 - \cos\theta)^5,$$

where $R_0$ is the reflectance at normal incidence:

$$R_0 = \left(\frac{n_t - 1}{n_t + 1}\right)^2.$$

Note that the $\cos\theta$ terms above are always for the angle in air (the larger of the internal and external angles relative to the normal).

For homogeneous impurities, as is found in typical glass, a light-carrying ray's intensity will be attenuated according to *Beer's Law*. As the ray travels through the medium it loses intensity according to $dI = -CI\,dx$, where $dx$ is distance. Thus, $dI/dx = -CI$. We can solve this equation and get the exponential $I = k\exp(-Cx) + k'$. The degree of attenuation is described by the RGB attenuation

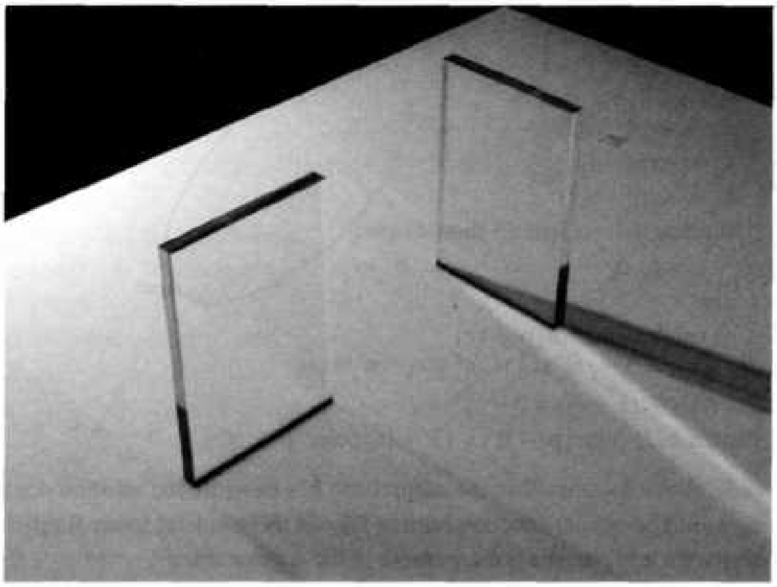


**Figure 10.11.** The color of the glass is affected by total internal reflection and Beer's Law. The amount of light transmitted and reflected is determined by the Fresnel equations. The complex lighting on the ground plane was computed using particle tracing as described in Chapter 23. (See also Plate VI.)

constant $a$, which is the amount of attenuation after one unit of distance. Putting in boundary conditions, we know that $I(0) = I_0$, and $I(1) = aI(0)$. The former implies $I(x) = I_0 \exp(-Cx)$. The latter implies $I_0 a = I_0 \exp(-C)$, so $-C = \ln(a)$. Thus, the final formula is

$$I(s) = I(0)e^{-\ln(a)s},$$

where $I(s)$ is the intensity of the beam at distance $s$ from the interface. In practice, we reverse-engineer $a$ by eye, because such data is rarely easy to find. The effect of Beer's Law can be seen in Figure 10.11, where the glass takes on a green tint.

To add transparent materials to our code, we need a way to determine when a ray is going "into" an object. The simplest way to do this is to assume that all objects are embedded in air with refractive index very close to 1.0, and that surface normals point "out" (toward the air). The code segment for rays and dielectrics with these assumptions is:

```
**if** (p is on a dielectric) **then**
  r = reflect($\mathbf{d}$, $\mathbf{n}$ )
  **if** ($\mathbf{d} \cdot \mathbf{n} < 0$) **then**
    refract($\mathbf{d}$, $\mathbf{n}$,n, $\mathbf{t}$ )
    $c = -\mathbf{d} \cdot \mathbf{n}$
    $k_r = k_g = k_b = 1$
else
    $k_r = \exp(-a_r t)$
    $k_g = \exp(-a_g t)$
    $k_b = \exp(-a_b t)$
    if refract($\mathbf{d}$, -$\mathbf{n}$,1/n, $\mathbf{t}$ ) then
        $c = \mathbf{t} \cdot \mathbf{n}$
    else
        return $k$*color($\mathbf{p} + t\mathbf{r}$)
$R_0 = (n-1)^2/(n+1)^2$
$R = R_0 + (1 - R_0)(1 - c)^5$
return $k(R$ color($\mathbf{p} + t\mathbf{r}$) + $(1 - R)$ color($\mathbf{p} + t\mathbf{t}$))
```

The code above assumes that the natural log has been folded into the constants $(a_r, a_g, a_b)$. The *refract* function returns false if there is total internal reflection, and otherwise it fills in the last argument of the argument list.

## 10.8 Instancing

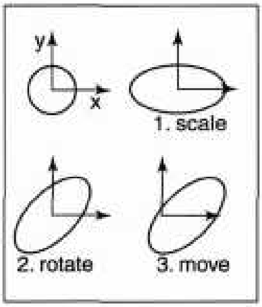


**Figure 10.12.** An instance of a circle with a series of three transforms is an ellipse.

An elegant property of ray tracing is that it allows very natural *instancing*. The basic idea of instancing is to distort all points on an object by a transformation matrix before the object is displayed. For example, if we transform the unit circle (in 2D) by a scale factor $(2, 1)$ in $x$ and $y$, respectively, then rotate it by $45°$, and move one unit in the $x$-direction, the result is an ellipse with an eccentricity of 2 and a long axis along the $x = -y$-direction centered at $(0, 1)$ (Figure 10.12). The key thing that makes that entity an "instance" is that we store the circle and the composite transform matrix. Thus, the explicit construction of the ellipse is left as a future procedure operation at render time.

The advantage of instancing in ray tracing is that we can choose the space in which to do intersection. If the base object is composed of a set of points, one of which is $\mathbf{p}$, then the transformed object is composed of that set of points transformed by matrix $\mathbf{M}$, where the example point is transformed to $\mathbf{Mp}$. If we have a ray $\mathbf{a} + t\mathbf{b}$ which we want to intersect with the transformed object, we can instead intersect an *inverse-transformed ray* with the untransformed object (Figure 10.13). There are two potential advantages to computing in the untransformed space (i.e., the right-hand side of Figure 10.13):

1. the untransformed object may have a simpler intersection routine, e.g., a sphere versus an ellipsoid;

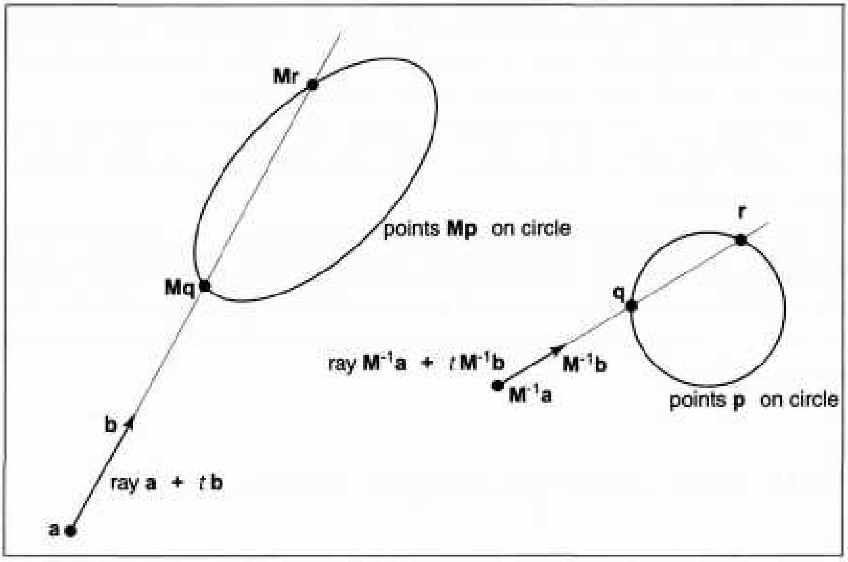


**Figure 10.13.** The ray intersection problem in the two spaces are just simple transforms of each other. The object is specified as a sphere plus matrix **M**. The ray is specified in the transformed (world) space by location **a** and direction **b**.

2. many transformed objects can share the same untransformed object thus reducing storage, e.g., a traffic jam of cars, where individual cars are just transforms of a few base (untransformed) models.

As discussed in Section 6.2.2, surface normal vectors transform differently. With this in mind and using the concepts illustrated in Figure 10.13, we can determine the intersection of a ray and an object transformed by matrix $\mathbf{M}$. If we create an instance class of type *surface*, we need to create a *hit* function:

```
instance::hit(ray $\mathbf{a} + t\mathbf{b}$, real $t_0$, real $t_1$, hit-record rec)
ray $\mathbf{r}' = \mathbf{M}^{-1}\mathbf{a} + t\mathbf{M}^{-1}\mathbf{b}$
if (base-object→hit($\mathbf{r}'$, $t_0$, $t_1$, rec)) then
    rec.$\mathbf{n} = (\mathbf{M}^{-1})^T$rec.$\mathbf{n}$
    return true
else
    return false
```

An elegant thing about this function is that the parameter rec.$t$ does not need to be changed, because it is the same in either space. Also note that we need not compute or store the matrix $\mathbf{M}$.

This brings up a very important point: the ray direction $\mathbf{b}$ must *not* be restricted to a unit-length vector, or none of the infrastructure above works. For this reason, it is useful not to restrict ray directions to unit vectors.

For the purpose of solid texturing, you may want to record the local coordinates of the hitpoint and return this in the hit-record. This is just ray $\mathbf{r}'$ advanced by parameter rec.$t$.

To implement the bounding-box function of class instance, we can just take the eight corners of the bounding box of the base object and transform all of them by $\mathbf{M}$, and then take the bounding box of those eight points. That will not necessarily yield the tightest bounding box, but it is general and straightforward to implement.

## 10.9 Sub-Linear Ray-Object Intersection

In the earlier ray-object intersection pseudocode, all objects are looped over, checking for intersections. For $N$ objects, this is an $O(N)$ linear search and is thus slow for large values of $N$. Like most search problems, the ray-object intersection can be computed in sub-linear time using "divide and conquer" techniques, provided we can create an ordered data structure as a preprocess. There are many techniques to do this.

This section discusses three of these techniques in detail: bounding volume hierarchies (Rubin & Whitted, 1980; Whitted, 1980; Goldsmith & Salmon, 1987), uniform spatial subdivision (Cleary, Wyvill, Birtwistle, & Vatti, 1983; Fujimoto, Tanaka, & Iwata, 1986; Amanatides & Woo, 1987), and binary space partition-

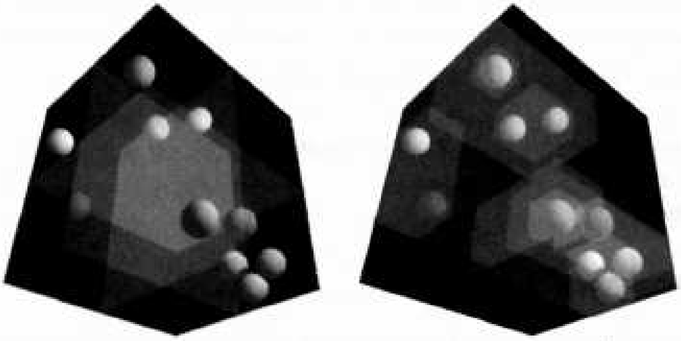


**Figure 10.14.** Left: a uniform partitioning of space. Right: adaptive bounding-box hierarchy. *Image courtesy David DeMarle.*

ing (Glassner, 1984; Jansen, 1986; Havran, 2000). An example of the first two strategies is shown in Figure 10.14.

### 10.9.1 Bounding Boxes

A key operation in most intersection acceleration schemes is computing the intersection of a ray with a bounding box (Figure 10.15). This differs from conventional intersection tests in that we do not need to know where the ray hits the box; we only need to know whether it hits the box.

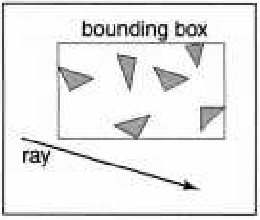


**Figure 10.15.** The ray is only tested for intersection with the surfaces if it hits the bounding box.

To build an algorithm for ray-box intersection, we begin by considering a 2D ray whose direction vector has positive $x$ and $y$ components. We can generalize this to arbitrary 3D rays later. The 2D bounding box is defined by two horizontal and two vertical lines:

$$\begin{aligned} x &= x_{\text{min}}, \\ x &= x_{\text{max}}, \\ y &= y_{\text{min}}, \\ y &= y_{\text{max}}. \end{aligned}$$

The points bounded by these lines can be described in interval notation:

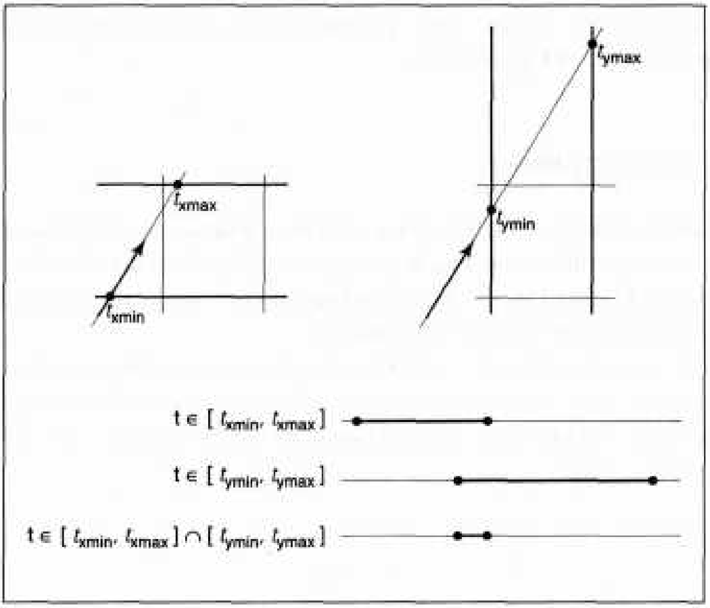
$$(x, y) \in [x_{\text{min}}, x_{\text{max}}] \times [y_{\text{min}}, y_{\text{max}}].$$

As shown in Figure 10.16, the intersection test can be phrased in terms of these intervals. First, we compute the ray parameter where the ray hits the line $x = x_{\text{min}}$:

$$t_{\text{xmin}} = \frac{x_{\text{min}} - x_e}{x_d}.$$

We then make similar computations for $t_{\text{xmax}}$, $t_{\text{ymin}}$, and $t_{\text{ymax}}$. The ray hits the box if and only if the intervals $[t_{\text{xmin}}, t_{\text{xmax}}]$ and $[t_{\text{ymin}}, t_{\text{ymax}}]$ overlap, i.e., their intersection is non-empty. In pseudocode this algorithm is:

```
$t_{\text{xmin}} = (x_{\text{min}} - x_e)/x_d$
$t_{\text{xmax}} = (x_{\text{max}} - x_e)/x_d$
$t_{\text{ymin}} = (y_{\text{min}} - y_e)/y_d$
$t_{\text{ymax}} = (y_{\text{max}} - y_e)/y_d$
**if** $(t_{\text{xmin}} > t_{\text{ymax}})$ or $(t_{\text{ymin}} > t_{\text{xmax}})$ **then**
  **return** false
**else**
  **return** true
```



$t_{ymax}$
$t_{xmax}$
$t_{ymin}$
$t_{xmin}$
t ∈ [ $t_{xmin}$, $t_{xmax}$ ]
t ∈ [ $t_{ymin}$, $t_{ymax}$ ]
t ∈ [ $t_{xmin}$, $t_{xmax}$ ] ∩ [ $t_{ymin}$, $t_{ymax}$ ]

**Figure 10.16.** The ray will be inside the interval $x \in [x_{\text{min}}, x_{\text{max}}]$ for some interval in its parameter space $t \in [t_{\text{xmin}}, t_{\text{xmax}}]$. A similar interval exists for the $y$ interval. The ray intersects the box if it is in both the $x$ interval and $y$ interval at the same time, i.e., the intersection of the two one-dimensional intervals is not empty.

The if statement may seem non-obvious. To see the logic of it, note that there is no overlap if the first interval is either entirely to the right or entirely to the left of the second interval.

The first thing we must address is the case when $x_d$ or $y_d$ is negative. If $x_d$ is negative, then the ray will hit $x_{\text{max}}$ before it hits $x_{\text{min}}$. Thus the code for computing $t_{\text{xmin}}$ and $t_{\text{xmax}}$ expands to:

```
**if** $(x_d \geq 0)$ **then**
    $t_{\text{xmin}} = (x_{\text{min}} - x_e)/x_d$
    $t_{\text{xmax}} = (x_{\text{max}} - x_e)/x_d$
**else**
    $t_{\text{xmin}} = (x_{\text{max}} - x_e)/x_d$
    $t_{\text{xmax}} = (x_{\text{min}} - x_e)/x_d$
```

A similar code expansion must be made for the $y$ cases. A major concern is that horizontal and vertical rays have a zero value for $y_d$ and $x_d$, respectively. This will cause divide by zero which may be a problem. However, before addressing this directly, we check whether IEEE floating point computation handles these cases gracefully for us. Recall from Section 1.6 the rules for divide by zero: for any positive real number $a$,

$$\begin{aligned} +a/0 &= +\infty; \\ -a/0 &= -\infty. \end{aligned}$$

Consider the case of a vertical ray where $x_d = 0$ and $y_d > 0$. We can then calculate

$$\begin{aligned} t_{\text{xmin}} &= \frac{x_{\text{min}} - x_e}{0}; \\ t_{\text{xmax}} &= \frac{x_{\text{max}} - x_e}{0}. \end{aligned}$$

There are three possibilities of interest:

1. $x_e \leq x_{\text{min}}$ (no hit);

2. $x_{\text{min}} < x_e < x_{\text{max}}$ (hit);

3. $x_{\text{max}} \leq x_e$ (no hit).

For the first case we have

$$\begin{aligned} t_{\text{xmin}} &= \frac{\text{positive number}}{0}; \\ t_{\text{xmax}} &= \frac{\text{positive number}}{0}. \end{aligned}$$

This yields the interval $(t_{\text{xmin}}, t_{\text{xmin}}) = (\infty, \infty)$. That interval will not overlap with any interval, so there will be no hit, as desired. For the second case, we have

$$\begin{aligned} t_{\text{xmin}} &= \frac{\text{negative number}}{0}; \\ t_{\text{xmax}} &= \frac{\text{positive number}}{0}. \end{aligned}$$

This yields the interval $(t_{\text{xmin}}, t_{\text{xmin}}) = (-\infty, \infty)$ which will overlap with all intervals and thus will yield a hit as desired. The third case results in the interval $(-\infty, -\infty)$ which yields no hit, as desired. Because these cases work as desired, we need no special checks for them. As is often the case, IEEE floating point conventions are our ally. However, there is still a problem with this approach.

Consider the code segment:

```
if ($x_d \geq 0$) then
    $t_{\text{min}} = (x_{\text{min}} - x_e)/x_d$
    $t_{\text{max}} = (x_{\text{max}} - x_e)/x_d$
else
    $t_{\text{min}} = (x_{\text{max}} - x_e)/x_d$
    $t_{\text{max}} = (x_{\text{min}} - x_e)/x_d$
```

This code breaks down when $x_d = -0$. This can be overcome by testing on the reciprocal of $x_d$ (A. Williams, Barrus, Morley, & Shirley, 2005):

```
$a = 1/x_d$
if ($a \geq 0$) then
    $t_{\text{min}} = a(x_{\text{min}} - x_e)$
    $t_{\text{max}} = a(x_{\text{max}} - x_e)$
else
    $t_{\text{min}} = a(x_{\text{max}} - x_e)$
    $t_{\text{max}} = a(x_{\text{min}} - x_e)$
```

### 10.9.2 Hierarchical Bounding Boxes

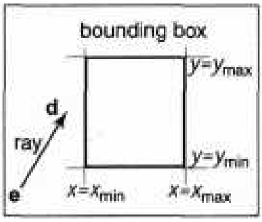


**Figure 10.17.** A 2D ray $\mathbf{e} + t\mathbf{d}$ is tested against a 2D bounding box.

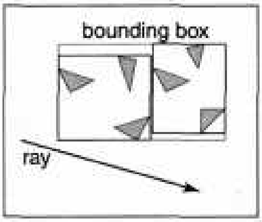


**Figure 10.18.** The bounding boxes can be nested by creating boxes around subsets of the model.

The basic idea of hierarchical bounding boxes can be seen by the common tactic of placing an axis-aligned 3D bounding box around all the objects as shown in Figure 10.17. Rays that hit the bounding box will actually be more expensive to compute than in a brute force search, because testing for intersection with the box is not free. However, rays that miss the box are cheaper than the brute force search. Such bounding boxes can be made hierarchical by partitioning the set of objects in a box and placing a box around each partition as shown in Figure 10.18. The data structure for the hierarchy shown in Figure 10.19 might be a tree with the large bounding box at the root and the two smaller bounding boxes as left and right subtrees. These would in turn each point to a list of three triangles. The intersection of a ray with this particular hard-coded tree would be:

```
if (ray hits root box) then
    if (ray hits left subtree box) then
        check three triangles for intersection
    if (ray intersects right subtree box) then
        check other three triangles for intersection
    if (an intersections returned from each subtree) then
        return the closest of the two hits
    else if (a intersection is returned from exactly one subtree) then
      return that intersection
    else
      return false
  else
    return false
```

Some observations related to this algorithm are that there is no geometric ordering between the two subtrees, and there is no reason a ray might not hit both subtrees. Indeed, there is no reason that the two subtrees might not overlap.

A key point of such data hierarchies is that a box is guaranteed to bound all objects that are below it in the hierarchy, but they are *not* guaranteed to contain all objects that overlap it spatially, as shown in Figure 10.19. This makes this geometric search somewhat more complicated than a traditional binary search on strictly ordered one-dimensional data. The reader may note that several possible optimizations present themselves. We defer optimizations until we have a full hierarchical algorithm.

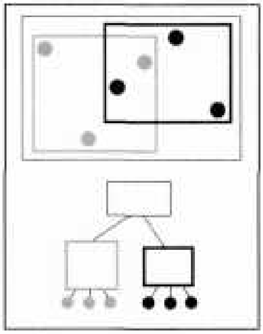


**Figure 10.19.** The grey box is a tree node that points to the three grey spheres, and the thick black box points to the three black spheres. Note that not all spheres enclosed by the box are guaranteed to be pointed to by the corresponding tree node.

If we restrict the tree to be binary and require that each node in the tree have a bounding box, then this traversal code extends naturally. Further, assume that all nodes are either leaves in the tree and contain a primitive, or that they contain one or two subtrees.

The *bvh-node* class should be of type surface, so it should implement *surface::hit*. The data it contains should be simple:

```
class bvh-node subclass of surface
virtual bool hit(ray $\mathbf{e} + t\mathbf{d}$, real $t_0$, real $t_1$, hit-record rec)
virtual box bounding-box()
surface-pointer left
surface-pointer right
box bbox
```

The traversal code can then be called recursively in an object-oriented style:

```
bool bvh-node::hit(ray $\mathbf{a} + t\mathbf{b}$, real $t_0$, real $t_1$, hit-record rec)
if (bbox.hitbox($\mathbf{a} + t\mathbf{b}$, $t_0$, $t_1$)) then
  hit-record lrec, rrec
  left-hit = (left $\neq$ NULL) and (left $\rightarrow$ hit($\mathbf{a} + t\mathbf{b}$, $t_0$, $t_1$, lrec))
  right-hit = (right $\neq$ NULL) and (right $\rightarrow$ hit($\mathbf{a} + t\mathbf{b}$, $t_0$, $t_1$, rrec))
  if (left-hit and right-hit) then
    if (lrec.t < rrec.t) then
      rec = lrec
    else
      rec = rrec
    return true
  else if (left-hit) then
    rec = lrec
    return true
  else if (right-hit) then
    rec = rrec
    return true
  else
    return false
else
  return false
```

Note that because *left* and *right* point to surfaces rather than bvh-nodes specifically, we can let the virtual functions take care of distinguishing between internal and leaf nodes; the appropriate hit function will be called. Note, that if the tree is built properly, we can eliminate the check for left being NULL. If we want to eliminate the check for right being NULL, we can replace NULL right pointers with a redundant pointer to left. This will end up checking left twice, but will eliminate the check throughout the tree. Whether that is worth it will depend on the details of tree construction.

There are many ways to build a tree for a bounding volume hierarchy. It is convenient to make the tree binary, roughly balanced, and to have the boxes of sibling subtrees not overlap too much. A heuristic to accomplish this is to sort the surfaces along an axis before dividing them into two sublists. If the axes are defined by an integer with $x = 0$, $y = 1$, and $z = 2$ we have:

```
bvh-node::bvh-node(object-array A, int AXIS)
N = A.length
if (N= 1) then
  left = A[0]
  right = NULL
  bbox = bounding-box(A[0])
else if (N= 2) then
  left-node = A[0]
  right-node = A[1]
  bbox = combine(bounding-box(A[0]), bounding-box(A[1]))
else
  sort A by the object center along AXIS
```

```
    left= new bvh-node(A[0..N/2 − 1], (AXIS +1) mod 3)
    right = new bvh-node(A[N/2..N−1], (AXIS +1) mod 3)
    bbox = combine(left-node → bbox, right-node → bbox)
```

The quality of the tree can be improved by carefully choosing AXIS each time. One way to do this is to choose the axis such that the sum of the volumes of the bounding boxes of the two subtrees is minimized. This change compared to rotating through the axes will make little difference for scenes composed of isotopically distributed small objects, but it may help significantly in less well-behaved scenes. This code can also be made more efficient by doing just a partition rather than a full sort.

Another, and probably better, way to build the tree is to have the subtrees contain about the same amount of space rather than the same number of objects. To do this we partition the list based on space:

```
bvh-node::bvh-node(object-array A, int AXIS)
N = A.length
**if** (N = 1) **then**
    left = A[0]
    right = NULL
    bbox = bounding-box(A[0])
**else if** (N = 2) **then**
    left = A[0]
    right = A[1]
    bbox = combine(bounding-box(A[0]), bounding-box(A[1]))
**else**
    find the midpoint $m$ of the bounding box of A along AXIS
    partition A into lists with lengths k and (N-k) surrounding $m$
    left = new node(A[0..k], (AXIS +1) mod 3)
    right = new node(A[k+1..N−1], (AXIS +1) mod 3)
    bbox = combine(left-node → bbox, right-node → bbox)
```

Although this results in an unbalanced tree, it allows for easy traversal of empty space and is cheaper to build because partitioning is cheaper than sorting.

### 10.9.3 Uniform Spatial Subdivision

Another strategy to reduce intersection tests is to divide space. This is fundamentally different from dividing objects as was done with hierarchical bounding volumes:

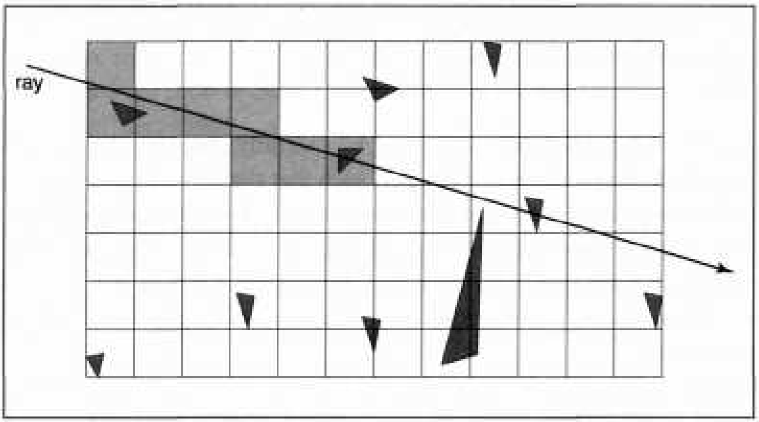


**Figure 10.20.** In uniform spatial subdivision, the ray is tracked forward through cells until an object in one of those cells is hit. In this example, only objects in the shaded cells are checked.

- In hierarchical bounding volumes, each object belongs to one of two sibling nodes, whereas a point in space may be inside both sibling nodes.
- In spatial subdivision, each point in space belongs to exactly one node, whereas objects may belong to many nodes.

The scene is partitioned into axis-aligned boxes. These boxes are all the same size, although they are not necessarily cubes. The ray traverses these boxes as shown in Figure 10.20. When an object is hit, the traversal ends.

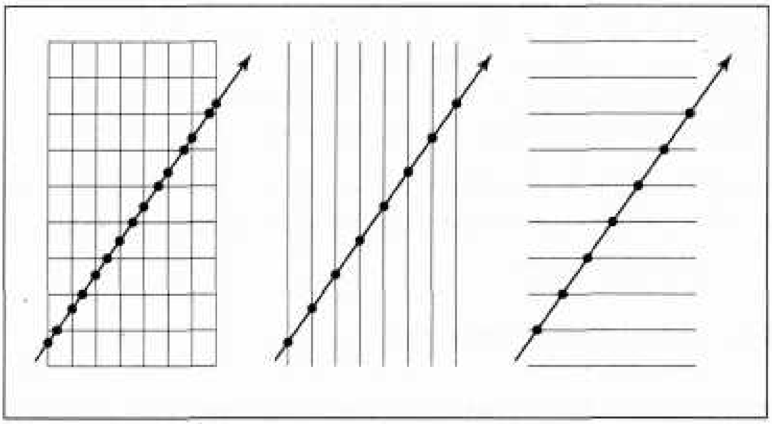


**Figure 10.21.** Although the pattern of cell hits seems irregular (left), the hits on sets of parallel planes are very even.

The grid itself should be a subclass of surface and should be implemented as a 3D array of pointers to surface. For empty cells these pointers are NULL. For cells with one object, the pointer points to that object. For cells with more than one object, the pointer can point to a list, another grid, or another data structure, such as a bounding volume hierarchy.

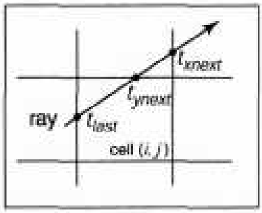


**Figure 10.22.** To decide whether we advance right or upwards, we keep track of the intersections with the next vertical and horizontal boundary of the cell.

This traversal is done in an incremental fashion. The regularity comes from the way that a ray hits each set of parallel planes, as shown in Figure 10.21. To see how this traversal works, first consider the 2D case where the ray direction has positive $x$ and $y$ components and starts outside the grid. Assume the grid is bounded by points $(x_{\min}, y_{\min})$ and $(x_{\max}, y_{\max})$. The grid has $n_x$ by $n_y$ cells.

Our first order of business is to find the index $(i, j)$ of the first cell hit by the ray $\mathbf{e} + t\mathbf{d}$. Then, we need to traverse the cells in an appropriate order. The key parts to this algorithm are finding the initial cell $(i, j)$ and deciding whether to increment $i$ or $j$ (Figure 10.22). Note that when we check for an intersection with objects in a cell, we restrict the range of $t$ to be within the cell (Figure 10.23). Most implementations make the 3D array of type "pointer to surface." To improve the locality of the traversal, the array can be tiled as discussed in Section 13.4.

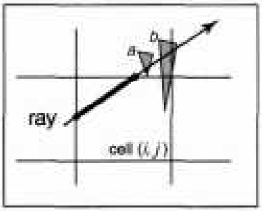


**Figure 10.23.** Only hits within the cell should be reported. Otherwise the case above would cause us to report hitting object *b* rather than object *a*.

### 10.9.4 Binary-Space Partitioning

We can also partition space in a hierarchical data structure such as a binary-space-partitioning tree (BSP tree). This is similar to the BSP tree used for a painter's algorithm in Chapter 8, but it usually uses axis-aligned cutting planes for easier ray intersection. A node in this structure might contain a single cutting plane and a left and right subtree. These subtrees would contain all objects on either side of the cutting plane. Objects that pass through the plane would be in each subtree. If we assume the cutting plane is parallel to the $yz$ plane at $x = D$, then the node class is:

```
class bsp-node subclass of surface
  virtual bool hit(ray $\mathbf{e} + t\mathbf{d}$, real $t_0$, real $t_1$, hit-record rec)
  virtual box bounding-box()
  surface-pointer left
  surface-pointer right
  real $D$
```

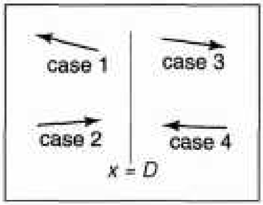


**Figure 10.24.** The four cases of how a ray relates to the BSP cutting plane $x = D$.

We generalize this to $y$ and $z$ cutting planes later. The intersection code can then be called recursively in an object-oriented style. The code considers the four cases shown in Figure 10.24. For our purposes, the origin of these rays is a point at parameter $t_0$:

$$\mathbf{p} = \mathbf{a} + t_0\mathbf{b}.$$

The four cases are:

1. The ray only interacts with the left subtree, and we need not test it for intersection with the cutting plane. It occurs for $x_p < D$ and $x_b < 0$.
2. The ray is tested against the left subtree, and if there are no hits, it is then tested against the right subtree. We need to find the ray parameter at $x = D$, so we can make sure we only test for intersections within the subtree. This case occurs for $x_p < D$ and $x_b > 0$.
3. This case is analogous to case 1 and occurs for $x_p > D$ and $x_b > 0$.
4. This case is analogous to case 2 and occurs for $x_p > D$ and $x_b < 0$.

The resulting traversal code handling these cases in order is:

```
bool bsp-node::hit( ray $\mathbf{a} + t\mathbf{b}$, real $t_0$, real $t_1$, hit-record rec)
$x_p = x_a + t_0 x_b$
**if** $(x_p < D)$ **then**
  **if** $(x_b < 0)$ **then**
    **return** (left $\neq$ NULL) and (left$\rightarrow$hit($\mathbf{a} + t\mathbf{b}$, $t_0$, $t_1$, rec))
  $t = (D - x_a)/x_b$
  **if** $(t > t_1)$ **then**
    **return** (left $\neq$ NULL) and (left$\rightarrow$hit($\mathbf{a} + t\mathbf{b}$, $t_0$, $t_1$, rec))
  **if** (left $\neq$ NULL) and (left$\rightarrow$hit($\mathbf{a} + t\mathbf{b}$, $t_0$, $t$, rec)) **then**
    **return** true
  **return** (right $\neq$ NULL) and (right$\rightarrow$hit($\mathbf{a} + t\mathbf{b}$, $t$, $t_1$, rec))
**else**
  analogous code for cases 3 and 4
```

This is very clean code. However, to get it started, we need to hit some root object that includes a bounding box so we can initialize the traversal, $t_0$ and $t_1$. An issue we have to address is that the cutting plane may be along any axis. We can add an integer index *axis* to the *bsp-node* class. If we allow an indexing operator for points, this will result in some simple modifications to the code above, for example,

$x_p = x_a + t_0 x_b$

would become

$u_p = a[\text{axis}] + t_0 b[\text{axis}]$

which will result in some additional array indexing, but will not generate more branches.

While the processing of a single bsp-node is faster than processing a bvh-node, the fact that a single surface may exist in more than one subtree means there are more nodes and, potentially, a higher memory use. How "well" the trees are built determines which is faster. Building the tree is similar to building the BVH tree. We can pick axes to split in a cycle, and we can split in half each time, or we can try to be more sophisticated in how we divide.

## 10.10 Constructive Solid Geometry

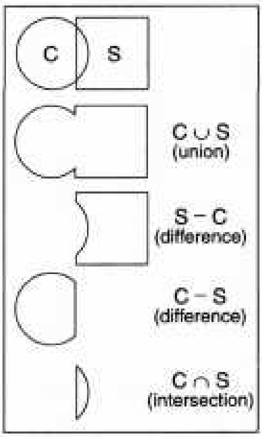


**Figure 10.25.** The basic CSG operations on a 2D circle and square.

One nice thing about ray tracing is that any geometric primitive whose intersection with a 3D line can be computed can be seamlessly added to a ray tracer. It turns out to also be straightforward to add *constructive solid geometry* (CSG) to a ray tracer (Roth, 1982). The basic idea of CSG is to use set operations to combine solid shapes. These basic operations are shown in Figure 10.25. The operations can be viewed as *set* operations. For example, we can consider $C$ the set of all points in the circle, and $S$ the set of all points in the square. The intersection operation $C \cap S$ is the set of all points that are both members of $C$ and $S$. The other operations are analogous.

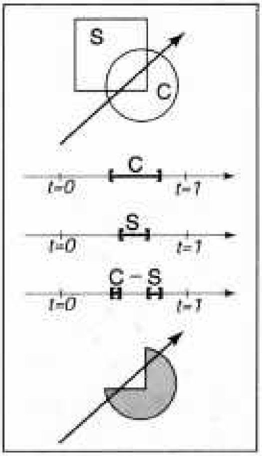


**Figure 10.26.** Intervals are processed to indicate how the ray hits the composite object.

Although one can do CSG directly on the model, if all that is desired is an image, we do not need to explicitly change the model. Instead, we perform the set operations directly on the rays as they interact with a model. To make this natural, we find all the intersections of a ray with a model rather than just the closest. For example, a ray $\mathbf{a} + t\mathbf{b}$ might hit a sphere at $t = 1$ and $t = 2$. In the context of CSG, we think of this as the ray being inside the sphere for $t \in [1, 2]$. We can compute these "inside intervals" for all of the surfaces and do set operations on those intervals (recall Section 2.1.2). This is illustrated in Figure 10.26, where the hit intervals are processed to indicate that there are two intervals inside the difference object. The first hit for $t > 0$ is what the ray actually intersects.

In practice, the CSG intersection routine must maintain a list of intervals. When the first hitpoint is determined, the material property and surface normal is that associated with the hitpoint. In addition, you must pay attention to precision issues because there is nothing to prevent the user from taking two objects that abut and taking an intersection. This can be made robust by eliminating any interval whose thickness is below a certain tolerance.

## 10.11 Distribution Ray Tracing

For some applications, ray-traced images are just too "clean." This effect can be mitigated using *distribution ray tracing* (Cook et al., 1984) . The conventionally ray-traced images look clean, because everything is crisp; the shadows are perfectly sharp, the reflections have no fuzziness, and everything is in perfect focus. Sometimes we would like to have the shadows be soft (as they are in real life), the reflections be fuzzy as with brushed metal, and the image have variable degrees of focus as in a photograph with a large aperture. While accomplishing these things from first principles is somewhat involved (as is developed in Chapter 23), we can get most of the visual impact with some fairly simple changes to the basic ray tracing algorithm. In addition, the framework gives us a relatively simple way to antialias (recall Section 3.7) the image.

### 10.11.1 Antialiasing

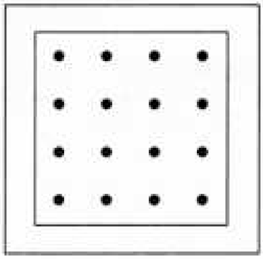


Figure 10.27. Sixteen regular samples for a single pixel.

Recall that a simple way to antialias an image is to compute the average color for the area of the pixel rather than the color at the center point. In ray tracing, our computational primitive is to compute the color at a point on the screen. If we average many of these points across the pixel, we are approximating the true average. If the screen coordinates bounding the pixel are $[i, i+1] \times [j, j+1]$, then we can replace the loop:

```
**for** each pixel $(i, j)$ **do**
    $c_{ij}$ = ray-color($i + 0.5, j + 0.5$)
```

with code that samples on a regular $n \times n$ grid of samples within each pixel:

```
**for** each pixel $(i, j)$ **do**
    $c = 0$
    **for** $p = 0$ to $n - 1$ **do**
        **for** $q = 0$ to $n - 1$ **do**
            $c = c +$ ray-color($i + (p + 0.5)/n, j + (q + 0.5)/n$)
    $c_{ij} = c/n^2$
```

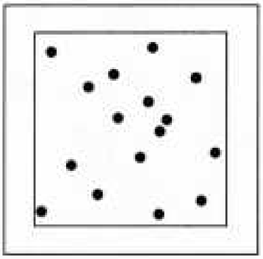


Figure 10.28. Sixteen random samples for a single pixel.

This is usually called *regular sampling*. The 16 sample locations in a pixel for $n = 4$ are shown in Figure 10.27. Note that this produces the same answer as rendering a traditional ray-traced image with one sample per pixel at $n_x n$ by $n_y n$ resolution and then averaging blocks of $n$ by $n$ pixels to get a $n_x$ by $n_y$ image.

One potential problem with taking samples in a regular pattern within a pixel is that regular artifacts such as moiré patterns can arise. These artifacts can be turned into noise by taking samples in a random pattern within each pixel as shown in Figure 10.28. This is usually called *random sampling* and involves just a small change to the code:

```
for each pixel $(i, j)$ do
    $c = 0$
    for $p = 1$ to $n^2$ do
        $c = c +$ ray-color$(i + \xi, j + \xi)$
    $c_{ij} = c/n^2$
```

Here $\xi$ is a call that returns a uniform random number in the range $[0, 1)$. Unfortunately, the noise can be quite objectionable unless many samples are taken. A compromise is to make a hybrid strategy that randomly perturbs a regular grid:

```
for each pixel $(i, j)$ do
    $c = 0$
    for $p = 0$ to $n - 1$ do
        for $q = 0$ to $n - 1$ do
            $c = c +$ ray-color$(i + (p + \xi)/n, j + (q + \xi)/n)$
    $c_{ij} = c/n^2$
```

That method is usually called *jittering* or *stratified sampling* (Figure 10.29).

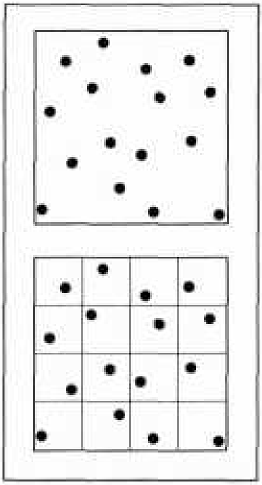


Figure 10.29. Sixteen stratified (jittered) samples for a single pixel shown with and without the bins highlighted. There is exactly one random sample taken within each bin.

### 10.11.2 Soft Shadows

The reason shadows are hard to handle in standard ray tracing is that lights are infinitesimal points or directions and are thus either visible or invisible. In real life, lights have non-zero area and can thus be partially visible. This idea is shown in 2D in Figure 10.30. The region where the light is entirely invisible is called the *umbra*. The partially visible region is called the *penumbra*. There is not a commonly used term for the region not in shadow, but it is sometimes called the *anti-umbra*.

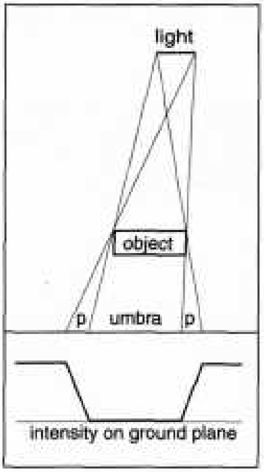


Figure 10.30. A soft shadow has a gradual transition from the unshadowed to shadowed region. The transition zone is the "penumbra" denoted by *p* in the figure.

The key to implementing soft shadows is to somehow account for the light being an area rather than a point. An easy way to do this is to approximate the light with a distributed set of $N$ point lights each with one $N$th of the intensity of the base light. This concept is illustrated at the left of Figure 10.31 where nine lights are used. You can do this in a standard ray tracer, and it is a common trick to get soft shadows in an off-the-shelf renderer. There are two potential problems with this technique. First, typically dozens of point lights are needed to achieve visually smooth results, which slows down the program a great deal. The second problem is that the shadows have sharp transitions inside the penumbra.

Distribution ray tracing introduces a small change in the shadowing code. Instead of representing the area light at a discrete number of point sources, we represent it as an infinite number and choose one at random for each viewing ray.

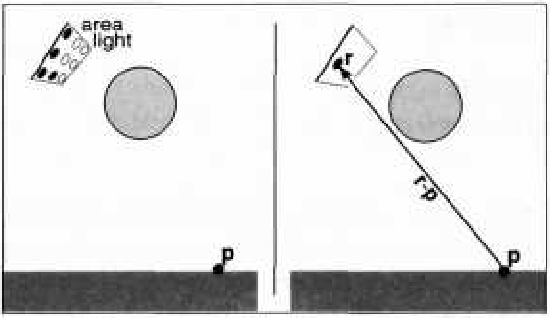


Figure 10.31. Left: an area light can be approximated by some number of point lights; four of the nine points are visible to **p** so it is in the penumbra. Right: a random point on the light is chosen for the shadow ray, and it has some chance of hitting the light or not.

This amounts to choosing a random point on the light for any surface point being lit as is shown at the right of Figure 10.31.

If the light is a parallelogram specified by a corner point $\mathbf{c}$ and two edge vectors $\mathbf{a}$ and $\mathbf{b}$ (Figure 10.32), then choosing a random point $\mathbf{r}$ is straightforward:

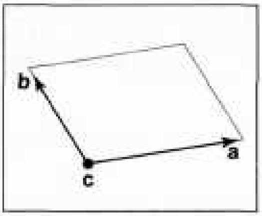


Figure 10.32. The geometry of a parallelogram light specified by a corner point and two edge vectors.

$$\mathbf{r} = \mathbf{c} + \xi_1 \mathbf{a} + \xi_2 \mathbf{b},$$

where $\xi_1$ and $\xi_2$ are uniform random numbers in the range $[0, 1)$.

We then send a shadow ray to this point as shown at the right in Figure 10.31. Note that the direction of this ray is not unit length, which may require some modification to your basic ray tracer depending upon its assumptions.

We would really like to jitter points on the light. However, it can be dangerous to implement this without some thought. We would not want to always have the ray in the upper left-hand corner of the pixel generate a shadow ray to the upper left-hand corner of the light. Instead we would like to scramble the samples, such that the pixel samples and the light samples are each themselves jittered, but so that there is no correlation between pixel samples and light samples. A good way to accomplish this is to generate two distinct sets of $n^2$ jittered samples and pass samples into the light source routine:

```
for each pixel $(i, j)$ do
  $c = 0$
  generate $N = n^2$ jittered 2D points and store in array r[ ]
  generate $N = n^2$ jittered 2D points and store in array s[ ]
  shuffle the points in array s[ ]
  for $p = 0$ to $N - 1$ do
    $c = c + $ ray-color($i + $ r[$p$].x(), $j + $ r[$p$].y(), s[$p$])
  $c_{ij} = c/N$
```

This shuffle routine eliminates any coherence between arrays $r$ and $s$. The shadow routine will just use the 2D random point stored in $s[p]$ rather than calling the random number generator. A shuffle routine for an array indexed from 0 to $N-1$ is:

```
for $i = N - 1$ downto 1 do
    choose random integer $j$ between 0 and $i$ inclusive
    swap array elements $i$ and $j$
```

### 10.11.3 Depth of Field

The soft focus effects seen in most photos can be simulated by collecting light at a non-zero size "lens" rather than at a point. This is called *depth of field*. The lens collects light from a cone of directions that has its apex at a distance where everything is in focus (Figure 10.33). We can place the "window" we are sampling on the plane where everything is in focus (rather than at the $z = n$ plane as we did previously), and the lens at the eye. The distance to the plane where everything is in focus we call the *focus plane*, and the distance to it is set by the user, just as the distance to the focus plane in a real camera is set by the user or range finder.

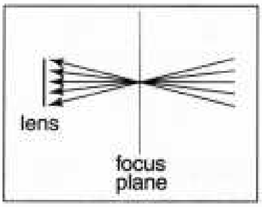


**Figure 10.33.** The lens averages over a cone of directions that hit the pixel location being sampled.

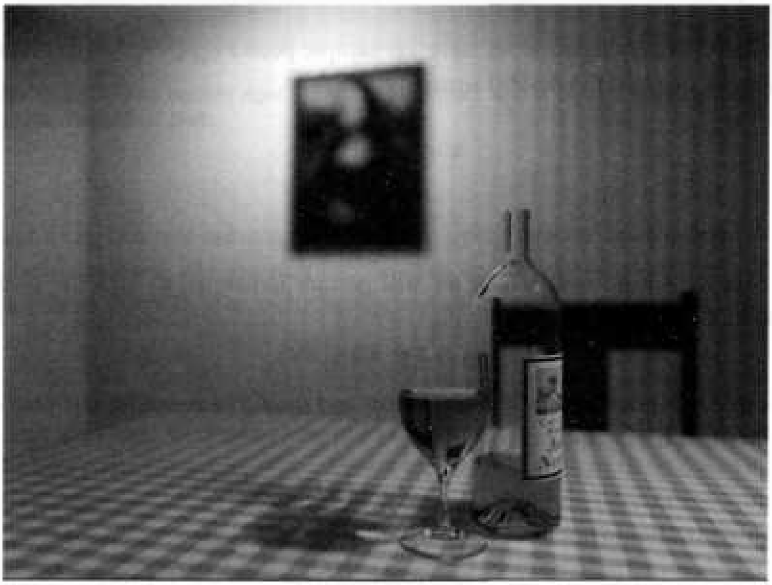


**Figure 10.34.** An example of depth of field. The caustic in the shadow of the wine glass is computed using particle tracing as described in Chapter 23. (See also Plate VII.)

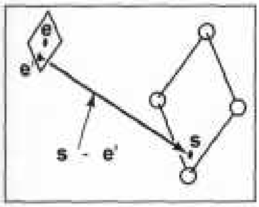


Figure 10.35. To create depth-of-field effects, the eye is randomly selected from a square region.

To be most faithful to a real camera, we should make the lens a disk. However, we will get very similar effects with a square lens (Figure 10.35). So we choose the side-length of the lens and take random samples on it. The origin of the view rays will be these perturbed positions rather than the eye position. Again, a shuffling routine is used to prevent correlation with the pixel sample positions. An example using 25 samples per pixel and a large disk lens is shown in Figure 10.34.

### 10.11.4 Glossy Reflection

Some surfaces, such as brushed metal, are somewhere between an ideal mirror and a diffuse surface. Some discernible image is visible in the reflection but it is blurred. We can simulate this by randomly perturbing ideal specular reflection rays as shown in Figure 10.36.

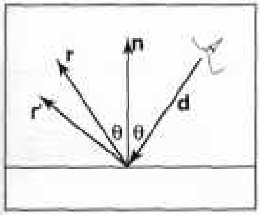


Figure 10.36. The reflection ray is perturbed to a random vector r'.

Only two details need to be worked out: how to choose the vector $\mathbf{r}'$, and what to do when the resulting perturbed ray is below the surface from which the ray is reflected. The latter detail is usually settled by returning a zero color when the ray is below the surface.

To choose $\mathbf{r}'$, we again sample a random square. This square is perpendicular to $\mathbf{r}$ and has width $a$ which controls the degree of blur. We can set up the square's orientation by creating an orthonormal basis with $\mathbf{w} = \mathbf{r}$ using the techniques in Section 2.4.6. Then, we create a random point in the 2D square with side length $a$ centered at the origin. If we have 2D sample points $(\xi, \xi') \in [0,1]^2$, then the analogous point on the desired square is

$$u = -\frac{a}{2} + \xi a,$$
$$v = -\frac{a}{2} + \xi' a.$$

Because the square over which we will perturb is parallel to both the $\mathbf{u}$ and $\mathbf{v}$ vectors, the ray $\mathbf{r}'$ is just

$$\mathbf{r}' = \mathbf{r} + u\mathbf{u} + v\mathbf{v}.$$

Note that $\mathbf{r}'$ is not necessarily a unit vector and should be normalized if your code requires that for ray directions.

### 10.11.5 Motion Blur

We can add a blurred appearance to objects as shown in Figure 10.37. This is called *motion blur* and is the result of the image being formed over a non-zero

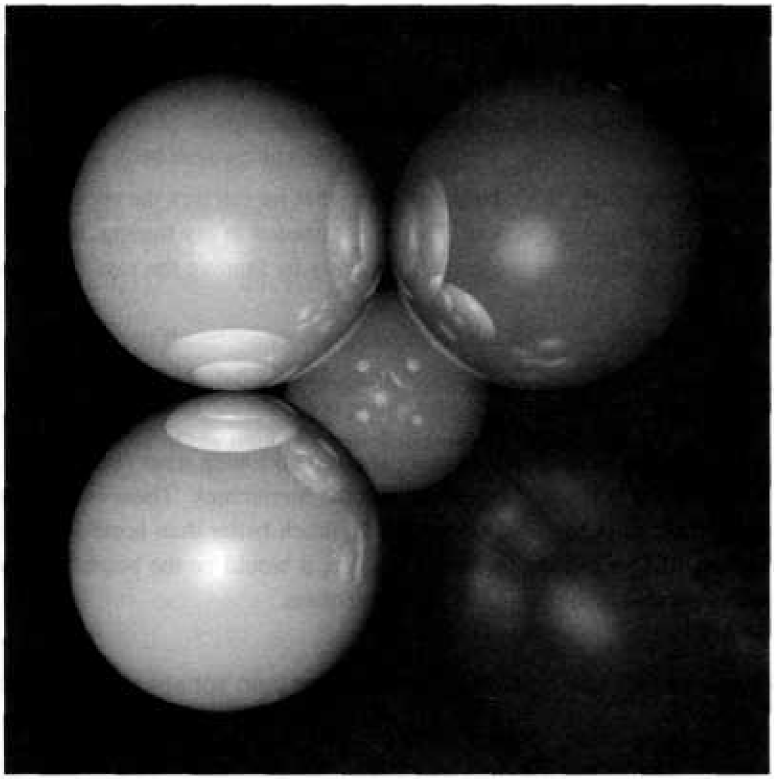


**Figure 10.37.** The bottom right sphere is in motion, and a blurred appearance results. *Image courtesy Chad Barb.*

span of time. In a real camera, the aperture is open for some time interval during which objects move. We can simulate the open aperture by setting a time variable ranging from $T_0$ to $T_1$. For each viewing ray we choose a random time,

$$T = T_0 + \xi(T_1 - T_0).$$

We may also need to create some objects to move with time. For example, we might have a moving sphere whose center travels from $\mathbf{c}_0$ to $\mathbf{c}_1$ during the interval. Given $T$, we could compute the actual center and do a ray–intersection with that sphere. Because each ray is sent at a different time, each will encounter the sphere at a different position, and the final appearance will be blurred. Note that the bounding box for the moving sphere should bound its entire path so an efficiency structure can be built for the whole time interval (Glassner, 1988).

## Frequently Asked Questions

- Why is there no perspective matrix in ray tracing?

The perspective matrix in a z-buffer exists so that we can turn the perspective projection into a parallel projection. This is not needed in ray tracing, because it is easy to do the perspective projection implicitly by fanning the rays out from the eye.

- What is the best ray-intersection efficiency structure?

The most popular structures are binary space partitioning trees (BSP trees), uniform subdivision grids, and bounding volume hierarchies. There is no clear-cut answer for which is best, but all are much, much better than brute-force search in practice. If I were to implement only one, it would be the bounding volume hierarchy because of its simplicity and robustness.

- Why do people use bounding boxes rather than spheres or ellipsoids?

Sometimes spheres or ellipsoids are better. However, many models have polygonal elements that are tightly bounded by boxes, but they would be difficult to tightly bind with an ellipsoid.

- Can ray tracing be made interactive?

For sufficiently small models and images, any modern PC is sufficiently powerful for ray tracing to be interactive. In practice, multiple CPUs with a shared frame buffer are required for a full-screen implementation. Computer power is increasing much faster than screen resolution, and it is just a matter of time before conventional PCs can ray trace complex scenes at screen resolution.

- Is ray tracing useful in a hardware graphics program?

Ray tracing is frequently used for *picking*. When the user clicks the mouse on a pixel in a 3D graphics program, the program needs to determine which object is visible within that pixel. Ray tracing is an ideal way to determine that.

## Exercises

1. What are the ray parameters of the intersection points between ray $(1, 1, 1)+t(-1, -1, -1)$ and the sphere centered at the origin with radius 1? Note: this is a good debugging case.

2. What are the barycentric coordinates and ray parameter where the ray $(1, 1, 1) + t(-1, -1, -1)$ hits the triangle with vertices $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$? Note: this is a good debugging case.

3. Do a back of the envelope computation of the approximate time complexity of ray tracing on "nice" (non-adversarial) models. Split your analysis into the cases of preprocessing and computing the image, so that you can predict the behavior of ray tracing multiple frames for a static model.

# 11

# Texture Mapping

The shading models presented in Chapter 9 assume that a diffuse surface has uniform reflectance $c_r$. This is fine for surfaces such as blank paper or painted walls, but it is inefficient for objects such as a printed sheet of paper. Such objects have an appearance whose complexity arises from variation in reflectance properties. While we could use such small triangles that the variation is captured by varying the reflectance properties of the triangles, this would be inefficient.

The common technique to handle variations of reflectance is to store the reflectance as a function or a a pixel-based image and "map" it onto a surface (Catmull, 1975). The function or image is called a *texture map*, and the process of controlling reflectance properties is called *texture mapping*. This is not hard to implement once you understand the coordinate systems involved. Texture mapping can be classified by several different properties:

1. the dimensionality of the texture function,

2. the correspondences defined between points on the surface and points in the texture function, and

3. whether the texture function is primarily procedural or primarily a table look-up.

These items are usually closely related, so we will somewhat arbitrarily classify textures by their dimension. We first cover 3D textures, often called *solid* textures or *volume* textures. We will then cover 2D textures, sometimes called *image* textures. When graphics programmers talk about textures without specifying dimension, they usually mean 2D textures. However, we begin with 3D textures because, in many ways, they are easier to understand and implement. At the end of the chapter we discuss bump mapping and displacement mapping which use textures to change surface normals and position, respectively. Although those methods modify properties other than reflectance, the images/functions they use are still called textured. This is consistent with common usage where any image used to modify object appearance is called a texture.

## 11.1 3D Texture Mapping

In previous chapters we used $c_r$ as the diffuse reflectance at a point on an object. For an object that does not have a solid color, we can replace this with a function $c_r(\mathbf{p})$ which maps 3D points to RGB colors (Peachey, 1985; Perlin, 1985). This function might just return the reflectance of the object that contains $\mathbf{p}$. But for objects with *texture*, we should expect $c_r(\mathbf{p})$ to vary as $\mathbf{p}$ moves across a surface. One way to do this is to create a 3D texture that defines an RGB value at every point in 3D space. We will only call it for points $\mathbf{p}$ on the surface, but it is usually easier to define it for all 3D points than a potentially strange 2D subset of points that are on an arbitrary surface. Such a strategy is clearly suitable for surfaces that are "carved" from a solid medium, such as a marble sculpture.

Note that in a ray-tracing program, we have immediate access to the point $\mathbf{p}$ seen through a pixel. However, for a z-buffer or BSP-tree program, we only know the point after projection into device coordinates. We will show how to resolve this problem in Section 11.4.1.

### 11.1.1 3D Stripe Textures

There are a surprising number of ways to make a striped texture. Let's assume we have two colors $c_0$ and $c_1$ that we want to use to make the stripe color. We need some oscillating function to switch between the two colors. An easy one is a sine:

```
RGB stripe( point $\mathbf{p}$ )
if ($\sin(x_\mathbf{p}) > 0$) then
  return $c_0$
else
  return $c_1$
```

We can also make the stripe's width $w$ controllable:

```
RGB stripe( point p, real $w$)
if $(\sin(\pi x_p/w) > 0)$ then
   return $c_0$
else
   return $c_1$
```

If we want to interpolate smoothly between the stripe colors, we can use a parameter $t$ to vary the color linearly:

```
RGB stripe( point p, real $w$ )
$t = (1 + \sin(\pi p_x/w))/2$
return $(1 - t)c_0 + tc_1$
```

These three possibilities are shown in Figure 11.1.

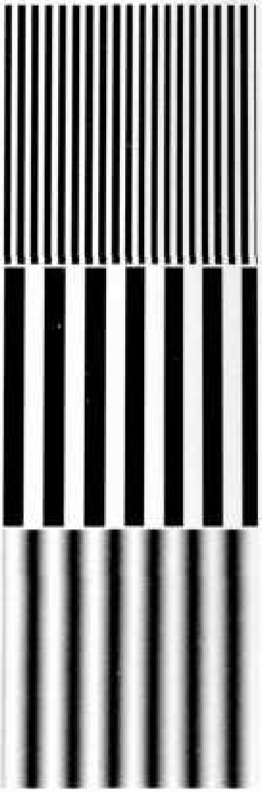


Figure 11.1. Various stripe textures result from drawing a regular array of *xy* points while keeping *z* constant.

### 11.1.2 Texture Arrays

Another way we can specify texture in space is to store a 3D array of color values and to associate a spatial position to each of these values. We first discuss this for 2D arrays in 2D space. Such textures can be applied in 3D by using two of the dimensions, e.g. $x$ and $y$, to determine what texture values are used. We then extend those 2D results to 3D.

We will assume the two dimensions to be mapped are called $u$ and $v$. We also assume we have an $n_x$ by $n_y$ image that we use as the texture. Somehow we need every $(u, v)$ to have an associated color found from the image. A fairly standard way to make texturing work for $(u, v)$ is to first remove the integer portion of $(u, v)$ so that it lies in the unit square. This has the effect of "tiling" the entire $uv$ plane with copies of the now-square texture (Figure 11.2). We then use one of three interpolation strategies to compute the image color for that coordinate. The simplest strategy is to treat each image pixel as a constant colored rectangular tile (Figure 11.3 (a). To compute the colors, we apply $c(u, v) = c_{ij}$, where $c(u, v)$ is the texture color at $(u, v)$ and $c_{ij}$ is the pixel color for pixel indices:

$$\begin{aligned} i &= \lfloor un_x \rfloor, \\ j &= \lfloor vn_y \rfloor; \end{aligned} \tag{11.1}$$

$\lfloor x \rfloor$ is the floor of $x$, $(n_x, n_y)$ is the size of the image being textured, and the indices start at $(i, j) = (0, 0)$. This method for a simple image is shown in Figure 11.3 (b).

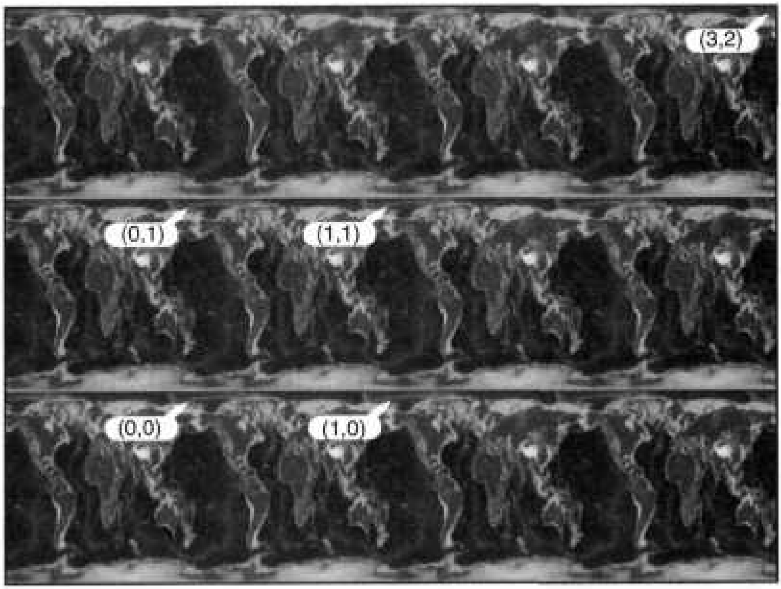


**Figure 11.2.** The tiling of an image onto the $(u,v)$ plane. Note that the input image is rectangular, and that this rectangle is mapped to a unit square on the $(u,v)$ plane.

For a smoother texture, a bilinear interpolation can be used as shown in Figure 11.3 (c). Here we use the formula

$$\begin{aligned} c(u,v) = {} & (1-u')(1-v')c_{ij} \\ & + u'(1-v')c_{(i+1)j} \\ & + (1-u')v'c_{i(j+1)} \\ & + u'v'c_{(i+1)(j+1)} \end{aligned}$$

where

$$\begin{aligned} u' &= n_x u - \lfloor n_x u \rfloor, \\ v' &= n_y v - \lfloor n_y v \rfloor. \end{aligned}$$

The discontinuities in the derivative in intensity can cause visible mach bands, so hermite smoothing can be used:

$$\begin{aligned} c(u,v) = {} & (1-u'')(1-v'')c_{ij} + \\ & + u''(1-v'')c_{(i+1)j} \\ & + (1-u'')v''c_{i(j+1)} \\ & + u''v''c_{(i+1)(j+1)}, \end{aligned}$$

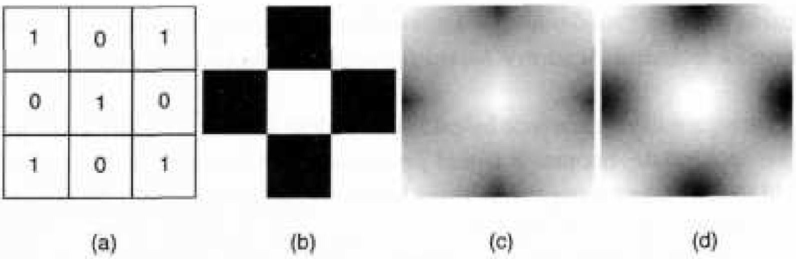


**Figure 11.3.** (a) The image on the left has nine pixels that are all either black or white. The three interpolation strategies are (b) nearest-neighbor, (c) bilinear, and (d) hermite.

where

$$\begin{aligned} u'' &= 3(u')^2 - 2(u')^3, \\ v'' &= 3(v')^2 - 2(v')^3, \end{aligned}$$

which results in Figure 11.3 (d).

In 3D, we have a 3D array of values. All of the ideas from 2D extend naturally. As an example, let's assume that we will do *trilinear* interpolation between values. First, we compute the texture coordinates $(u', v', w')$ and the lower indices $(i, j, k)$ of the array element to be interpolated:

$$\begin{aligned} c(u, v, w) = {} & (1 - u')(1 - v')(1 - w')c_{ijk} \\ & + u'(1 - v')(1 - w')c_{(i+1)jk} \\ & + (1 - u')v'(1 - w')c_{i(j+1)k} \\ & + (1 - u')(1 - v')w'c_{ij(k+1)} \\ & + u'v'(1 - w')c_{(i+1)(j+1)k} \\ & + u'(1 - v')w'c_{(i+1)j(k+1)} \\ & + (1 - u')v'w'c_{i(j+1)(k+1)} \\ & + u'v'w'c_{(i+1)(j+1)(k+1)}, \end{aligned} \tag{11.2}$$

where

$$\begin{aligned} u' &= n_x u - \lfloor n_x u \rfloor, \\ v' &= n_y v - \lfloor n_y v \rfloor, \\ w' &= n_z w - \lfloor n_z w \rfloor. \end{aligned} \tag{11.3}$$

### 11.1.3 Solid Noise

Although regular textures such as stripes are often useful, we would like to be able to make "mottled" textures such as we see on birds' eggs. This is usually done by using a sort of "solid noise," usually called *Perlin noise* after its inventor, who received a technical Academy Award for its impact in the film industry (Perlin, 1985).

Getting a noisy appearance by calling a random number for every point would not be appropriate, because it would just be like "white noise" in TV static. We would like to make it smoother without losing the random quality. One possibility is to blur white noise, but there is no practical implementation of this. Another possibility is to make a large lattice with a random number at every lattice point, and then interpolate these random points for new points between lattice nodes; this is just a 3D texture array as described in the last section with random numbers in the array. This technique makes the lattice too obvious. Perlin used a variety of tricks to improve this basic lattice technique so the lattice was not so obvious. This results in a rather baroque-looking set of steps, but essentially there are just three changes from linearly interpolating a 3D array of random values. The first change is to use Hermite interpolation to avoid mach bands, just as can be done with regular textures. The second change is the use of random vectors rather than values, with a dot product to derive a random number; this makes the underlying grid structure less visually obvious by moving the local minima and maxima off the grid vertices. The third change is to use a 1D array and hashing to create a virtual 3D array of random vectors. This adds computation to lower memory use. Here is his basic method:

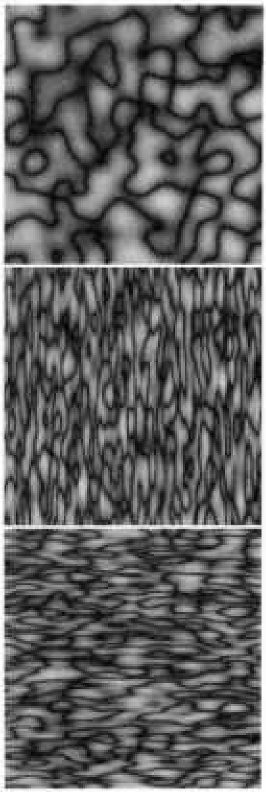


Figure 11.4. Absolute value of solid noise, and noise for scaled *x* and *y* values.

$$n(x, y, z) = \sum_{i=\lfloor x \rfloor}^{\lfloor x \rfloor + 1} \sum_{j=\lfloor y \rfloor}^{\lfloor y \rfloor + 1} \sum_{k=\lfloor z \rfloor}^{\lfloor z \rfloor + 1} \Omega_{ijk}(x - i, y - j, z - k),$$

where $(x, y, z)$ are the Cartesian coordinates of $\mathbf{x}$, and

$$\Omega_{ijk}(u, v, w) = \omega(u)\omega(v)\omega(w)\left(\Gamma_{ijk} \cdot (u, v, w)\right),$$

and $\omega(t)$ is the cubic weighting function:

$$\omega(t) = \begin{cases} 2|t|^3 - 3|t|^2 + 1 & \text{if } |t| < 1, \\ 0 & \text{otherwise.} \end{cases}$$

The final piece is that $\Gamma_{ijk}$ is a random unit vector for the lattice point $(x, y, z) = (i, j, k)$. Since we want any potential $ijk$, we use a pseudorandom table:

$$\Gamma_{ijk} = \mathbf{G}\left(\phi(i + \phi(j + \phi(k)))\right),$$

where $\mathbf{G}$ is a precomputed array of $n$ random unit vectors, and $\phi(i) = P[i \mod n]$ where $P$ is an array of length $n$ containing a permutation of the integers 0 through $n-1$. In practice, Perlin reports $n = 256$ works well. To choose a random unit vector $(v_x, v_y, v_z)$ first set

$$\begin{aligned} v_x &= 2\xi \ \ -1, \\ v_y &= 2\xi' \ -1, \\ v_z &= 2\xi'' -1, \end{aligned}$$

where $\xi, \xi', \xi''$ are canonical random numbers (uniform in the interval $[0, 1)$). Then, if $(v_x^2 + v_y^2 + v_z^2) < 1$, make the vector a unit vector. Otherwise keep setting it randomly until its length is less than one, and then make it a unit vector. This is an example of a *rejection method*, which will be discussed more in Chapter 14. Essentially, the "less than" test gets a random point in the unit sphere, and the vector for the origin to that point is uniformly random. That would not be true of random points in the cube, so we "get rid" of the corners with the test.

Because solid noise can be positive or negative, it must be transformed before being converted to a color. The absolute value of noise over a ten by ten square is shown in Figure 11.4, along with stretched versions. There versions are stretched by scaling the points input to the noise function.

The dark curves are where the original noise function changed from positive to negative. Since noise varies from $-1$ to 1, a smoother image can be achieved by using $(\text{noise}+1)/2$ for color. However, since noise values close to 1 or $-1$ are rare, this will be a fairly smooth image. Larger scaling can increase the contrast (Figure 11.5).

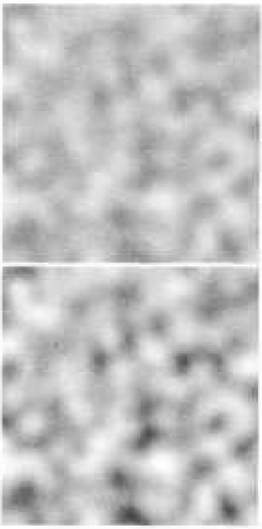


**Figure 11.5.** Using 0.5(noise+1) (top) and 0.8(noise+1) (bottom) for intensity.

### 11.1.4 Turbulence

Many natural textures contain a variety of feature sizes in the same texture. Perlin uses a pseudofractal "turbulence" function:

$$n_t(\mathbf{x}) = \sum_i \frac{|n(2^i \mathbf{x})|}{2^i}$$

This effectively repeatedly adds scaled copies of the noise function on top of itself as shown in Figure 11.6.

The turbulence can be used to distort the stripe function:

```
RGB turbstripe( point $\mathbf{p}$, double w )
double $t = (1 + \sin(k_1 z_p + \text{turbulence}(k_2 \mathbf{p}))/w)/2$
**return** $t * s0 + (1 - t) * s1$
```

Various values for $k_1$ and $k_2$ were used to generate Figure 11.7.

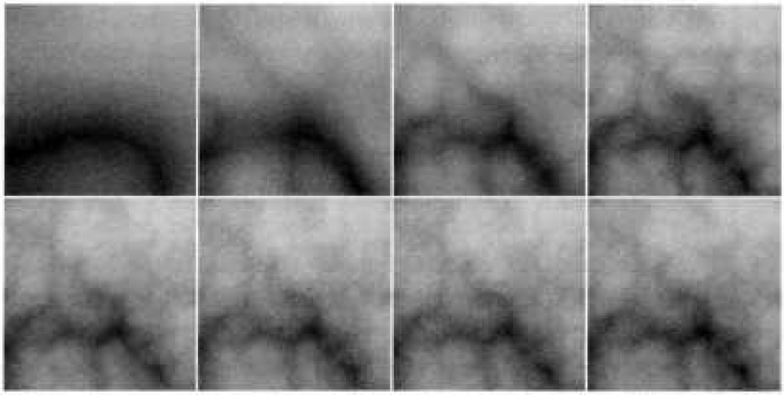


**Figure 11.6.** Turbulence function with (from top left to bottom right) one through eight terms in the summation.

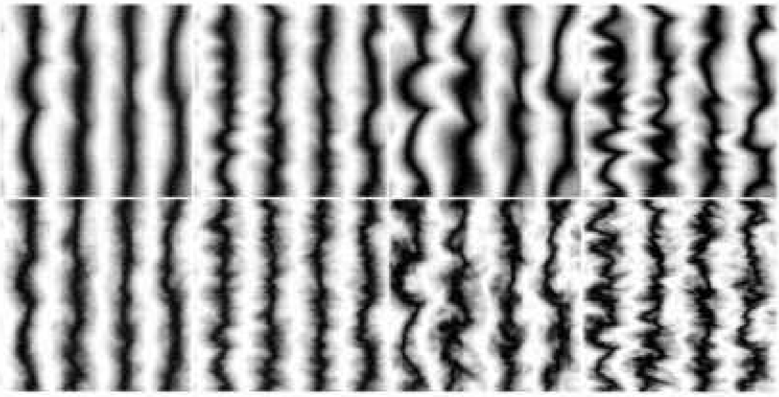


**Figure 11.7.** Various turbulent stripe textures with different $k_1$ , $k_2$. The top row has only the first term of the turbulence series.

## 11.2 2D Texture Mapping

For 2D texture mapping, we use a 2D coordinate, often called $uv$, which is used to create a reflectance $R(u, v)$. The key is to take an image and associate a $(u, v)$ coordinate system on it so that it can, in turn, be associated with points on a 3D surface. For example, if the latitudes and longitudes on the world map are associated with a polar coordinate system on the sphere, we get a globe (Figure 11.8).

It is crucial that the coordinates on the image and the object match in "just the right way." As a convention, the coordinate system on the image is set to be the unit square $(u, v) \in [0, 1]^2$. For $(u, v)$ outside of this square, only the fractional parts of the coordinates are used resulting in a tiling of the plane (Figure 11.2).

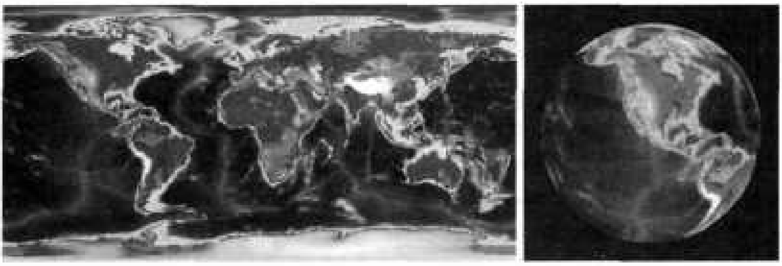


**Figure 11.8.** A Miller cylindrical projection map world map and its placement on the sphere. The distortions in the texture map (i.e., Greenland being so large) exactly correspond to the shrinking that occurs when the map is applied to the sphere.

Note that the image has a different number of pixels horizontally and vertically, so the image pixels have a non-uniform aspect ratio in $(u, v)$ space.

To map this $(u, v) \in [0, 1]^2$ image onto a sphere, we first compute the polar coordinates. Recall the spherical coordinate system described by Equation 2.24. For a sphere of radius $R$ with center $(c_x, c_y, c_z)$, the parametric equation of the sphere is

$$\begin{aligned} x &= x_c + R\cos\phi\sin\theta, \\ y &= y_c + R\sin\phi\sin\theta, \\ z &= z_c + R\cos\theta. \end{aligned}$$

We can find $(\theta, \phi)$:

$$\theta = \arccos\left(\frac{z - z_c}{R}\right),$$

$$\phi = \operatorname{arctan2}(y - y_c, x - x_c),$$

where $\operatorname{arctan2}(a, b)$ is the the *atan2* of most math libraries which returns the arctangent of $a/b$. Because $(\theta, \phi) \in [0, \pi] \times [-\pi, \pi]$, we convert to $(u, v)$ as follows, after first adding $2\pi$ to $\phi$ if it is negative:

$$\begin{aligned} u &= \frac{\phi}{2\pi}, \\ v &= \frac{\pi - \theta}{\pi}. \end{aligned}$$

This mapping is shown in Figure 11.8. There is a similar, although likely more complicated way, to generate coordinates for most 3D shapes.

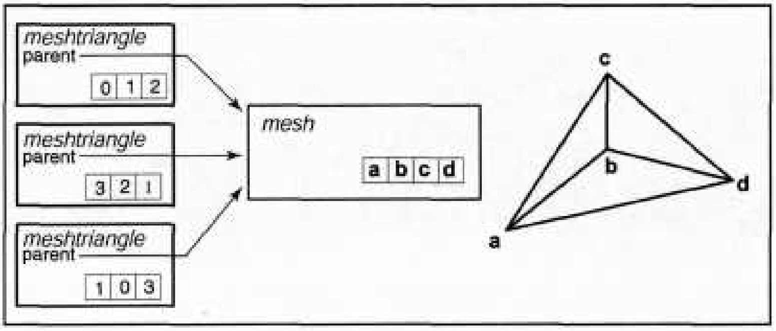


**Figure 11.9.** A three triangle mesh with four vertices.

## 11.3 Tessellated Models

Most real-world models are composed of complexes of triangles with shared vertices. These are usually known as *triangular meshes* or *triangular irregular networks* (TINs). Most graphics programs need to make these models without using too much storage and with the ability to handle texture-maps.

A simple triangular mesh is shown in Figure 11.9. You could store these three triangles as independent entities, and thus store point $\mathbf{p}_1$ three times and the other vertices twice each for a total of nine stored points (three vertices for each of three triangles), or you could try to somehow share the common vertices and store only four. So instead of

```
class triangle
material m
vector3 $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$
```

you would have two classes:

```
class mesh
material m
array of vector3 vertices
```

and

```
class meshtriangle
pointer to mesh meshptr
int $i_0$, $i_1$, $i_2$,
```

where $i_0$, $i_1$, and $i_2$ are indices into the *vertices* array. Either the triangle class or the mesh class will work. Is there a space advantage for the mesh class? Typically, a large mesh has each vertex being stored by about six triangles, although there can be any number for extreme cases. This means about two triangles for each shared vertex. If you have $n$ triangles, then there are about $n/2$ vertices in the shared case and $3n$ in the unshared case. But, when you share, you need an additional $3n$ integers and $n$ pointers. Since you don't have to store the material in each mesh triangle, that saves $n$ pointers, which cancels out the storage for *meshptr*. If we assume that the data for floats, pointers, and ints all require the same storage (a dubious assumption), the triangles will take $10n$ storage units and the mesh will take $5.5n$ storage units. So the mesh reduces the storage by about a factor of two; this seems to hold for most implementations. Is this factor of two worth the complication? I think the answer is yes as soon as you start adding "properties" to the vertices.

Each vertex can have material parameters, texture coordinates, irradiances, and essentially any parameter that a renderer might use. In practice, these parameters are bilinearly interpolated across the triangle. So, if a triangle is intersected at barycentric coordinates $(\beta, \gamma)$, you interpolate the $(u, v)$ coordinates the same way you interpolate points. Recall that the point at barycentric coordinate $(\beta, \gamma)$ is

$$\mathbf{p}(\beta, \gamma) = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}).$$

A similar equation applies for $(u, v)$:

$$\begin{aligned} u(\beta, \gamma) &= u_a + \beta(u_b - u_a) + \gamma(u_c - u_d), \\ v(\beta, \gamma) &= v_a + \beta(v_b - v_a) + \gamma(v_c - v_a). \end{aligned}$$

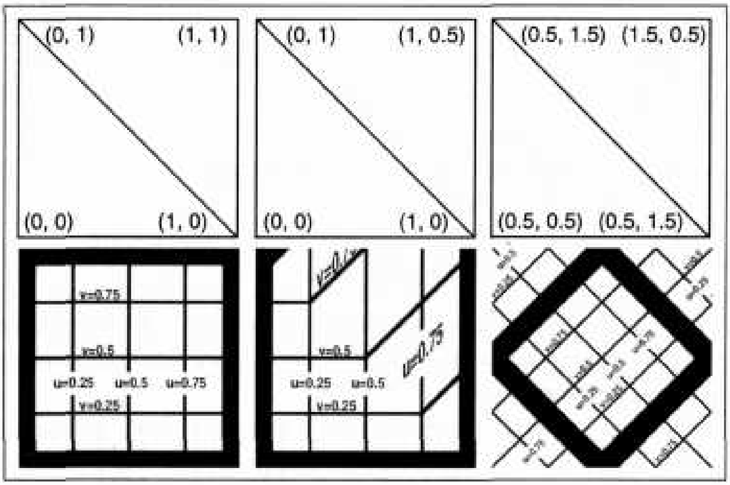


**Figure 11.10.** Various mesh textures obtained by changing $(u,v)$ coordinates stored at vertices.

Several ways a texture can be applied by changing the $(u, v)$ at triangle vertices are shown in Figure 11.10. This sort of calibration texture map makes it easier to understand the texture coordinates of your objects during debugging (Figure 11.11).

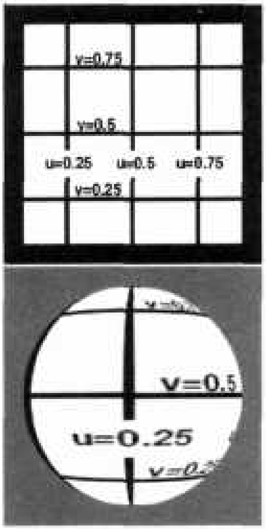


**Figure 11.11.** Top: a calibration texture map. Bottom: the sphere viewed along the *y*-axis.

## 11.4 Texture Mapping for Rasterized Triangles

We would like to get the same texture images whether we use a ray tracing program or a rasterization method, such as a z-buffer. There are some subtleties in achieving this with correct-looking perspective, but we can address this at the rasterization stage. The reason things are not straightforward is that just interpolating texture coordinates in screen space results in incorrect images, as shown for the grid texture shown in Figure 11.12. Because things in perspective get smaller as the distance to the viewer increases, the lines that are evenly spaced in 3D should compress in 2D image space. More careful interpolation of texture coordinates is needed to accomplish this.

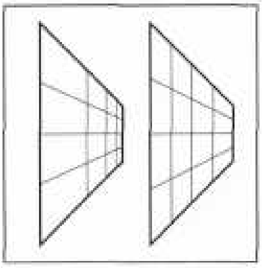


**Figure 11.12.** Left: correct perspective. Right: interpolation in screen space.

### 11.4.1 Perspective Correct Textures

We can implement texture mapping on triangles by interpolating the $(u, v)$ coordinates, modifying the rasterization method of Section 3.6, but this results in the problem shown at the right of Figure 11.12. A similar problem occurs for triangles if screen-space barycentric coordinates are used as in the following rasterization code:

```
for all $x$ do
  for all $y$ do
    compute $(\alpha, \beta, \gamma)$ for $(x, y)$
    if $\alpha \in (0, 1)$ and $\beta \in (0, 1)$ and $\gamma \in (0, 1)$ then
      $\mathbf{t} = \alpha\mathbf{t}_0 + \beta\mathbf{t}_1 + \gamma\mathbf{t}_2$
      drawpixel $(x, y)$ with color texture($\mathbf{t}$) for a solid texture
      or with texture($\beta, \gamma$) for a 2D texture.
```

This code will generate images, but there is a problem. To unravel the basic problem, let's consider the progression from world space $\mathbf{q}$ to homogeneous point $\mathbf{r}$ to homogenized point $\mathbf{s}$:

$$\begin{bmatrix} x_q \\ y_q \\ z_q \\ 1 \end{bmatrix} \xrightarrow{\text{transform}} \begin{bmatrix} x_r \\ y_r \\ z_r \\ h_r \end{bmatrix} \xrightarrow{\text{homogenize}} \begin{bmatrix} x_r/h_r \\ y_r/h_r \\ z_r/h_r \\ 1 \end{bmatrix} \equiv \begin{bmatrix} x_s \\ y_s \\ z_s \\ 1 \end{bmatrix}.$$

If we use screen space, we are interpolating in s. However, we would like to be interpolating in space q or r, where the homogeneous division has not yet nonlinearly distorted the barycentric coordinates of the triangle.

The key observation is that $1/h_r$ is interpolated with no distortion. Likewise, so is $u/h_r$ and $v/h_r$. In fact, so is $k/h_r$, where $k$ is any quantity that varies linearly across the triangle. Recall from Section 7.4 that if we transform all points along the line segment between points q and Q and homogenize, we have

$$\mathbf{s} + \frac{h_R t}{h_r + t(h_R - h_r)}(\mathbf{S} - \mathbf{s}),$$

but if we linearly interpolate in the homogenized space we have

$$\mathbf{s} + a(\mathbf{S} - \mathbf{s}).$$

Although those lines sweep out the same points, typically $a \neq t$ for the same points on the line segment. However, if we interpolate $1/h$, we *do* get the same answer regardless of which space we interpolate in. To see this is true, confirm (Exercise 2):

$$\frac{1}{h_r} + \frac{h_R t}{h_r + t(h_R - h_r)}\left(\frac{1}{h_R} - \frac{1}{h_r}\right) = \frac{1}{h_r} + t\left(\frac{1}{h_R} - \frac{1}{h_r}\right) \qquad (11.4)$$

This ability to interpolate $1/h$ linearly with no error in the transformed space allows us to correctly texture triangles. Perhaps the least confusing way to deal with this distortion is to compute the world space barycentric coordinates of the triangle $(\beta_w, \gamma_w)$ in terms of screen space coordinates $(\beta, \gamma)$. We note that $\beta_s/h$ and $\gamma_s/h$ can be interpolated linearly in screen space. For example, at the screen space position associated with screen space barycentric coordinates $(\beta, \gamma)$, we can interpolate $\beta_w/h$ without distortion. Because $\beta_w = 0$ at vertex 0 and vertex 2, and $\beta_w = 1$ at vertex 1, we have

$$\frac{\beta_s}{h} = \frac{0}{h_0} + \beta\left(\frac{1}{h_1} - \frac{0}{h_0}\right) + \gamma\left(\frac{0}{h_2} - \frac{0}{h_0}\right). \qquad (11.5)$$

Because of all the zero terms, Equation 11.5 is fairly simple. However, to get $\beta_w$ from it, we must know $h$. Because we know $1/h$ is linear in screen space, we have

$$\frac{1}{h} = \frac{1}{h_0} + \beta\left(\frac{1}{h_1} - \frac{1}{h_0}\right) + \gamma\left(\frac{1}{h_2} - \frac{1}{h_0}\right). \qquad (11.6)$$

Dividing Equation 11.5 by Equation 11.6 gives

$$\beta_w = \frac{\frac{\beta}{h_1}}{\frac{1}{h_0} + \beta\left(\frac{1}{h_1} - \frac{1}{h_0}\right) + \gamma\left(\frac{1}{h_2} - \frac{1}{h_0}\right)}.$$

Multiplying numerator and denominator by $h_0h_1h_2$ and doing a similar set of manipulations for the analogous equations in $\gamma_w$ gives

$$\begin{aligned}\beta_w &= \frac{h_0h_2\beta}{h_1h_2 + h_2\beta(h_0 - h_1) + h_1\gamma(h_0 - h_2)},\\ \gamma_w &= \frac{h_0h_1\gamma}{h_1h_2 + h_2\beta(h_0 - h_1) + h_1\gamma(h_0 - h_2)}.\end{aligned} \tag{11.7}$$

Note that the two denominators are the same.

For triangles that use the perspective matrix from Chapter 7, recall that $w = z/n$ where $z$ is the distance from the viewer perpendicular to the screen. Thus, for that matrix $1/z$ also varies linearly. We can use this fact to modify our scan-conversion code for three points $\mathbf{t}_i = (x_i, y_i, z_i, h_i)$ that have been passed through the viewing matrices, but have not been homogenized:

```
Compute bounds for $x = x_i/h_i$ and $y = y_i/h_i$
for all $x$ do
  for all $y$ do
    compute $(\alpha, \beta, \gamma)$ for $(x, y)$
    if ($\alpha \in [0, 1]$ and $\beta \in [0, 1]$ and $\gamma \in [0, 1]$) then
      $d = h_1h_2 + h_2\beta(h_0 - h_1) + h_1\gamma(h_0 - h_2)$
      $\beta_w = h_0h_2\beta/d$
      $\gamma_w = h_0h_1\gamma/d$
      $\alpha_w = 1 - \beta_w - \gamma_w$
      $u = \alpha_w u_0 + \beta_w u_1 + \gamma_w u_2$
      $v = \alpha_w v_0 + \beta_w v_1 + \gamma_w v_2$
      drawpixel $(x, y)$ with color texture$(u, v)$
```

For solid textures, just recall that by the definition of barycentric coordinates

$$\mathbf{p} = (1 - \beta_w - \gamma_w)\mathbf{p}_0 + \beta_w\mathbf{p}_1 + \gamma_w\mathbf{p}_2,$$

where $\mathbf{p}_i$ are the world space vertices. Then, just call a solid texture routine for point $\mathbf{p}$.

## 11.5 Bump Textures

Although, so far, we have only discussed changing reflectance using texture, you can also change the surface normal to give an illusion of fine-scale geometry on the surface. We can apply a *bump map* which perturbs the surface normal (J. F. Blinn, 1978).

One way to do this is:

```
vector3 $n$ = surfaceNormal($x$)
$n$ += $k_1 * $ vectorTurbulence($k_2 * x$)
return $t * s0 + (1 - t) * s1$
```

This is shown in Figure 11.13.

To implement *vectorTurbulence*, we first need *vectorNoise* which produces a simple spatially-varying 3D vector:

$$n_v(x, y, z) = \sum_{i=\lfloor x \rfloor}^{\lfloor x \rfloor + 1} \sum_{j=\lfloor y \rfloor}^{\lfloor y \rfloor + 1} \sum_{k=\lfloor z \rfloor}^{\lfloor z \rfloor + 1} \Gamma_{ijk}\omega(x)\omega(y)\omega(z).$$

Then, *vectorTurbulence* is a direct analog of turbulence: sum a series of scaled versions of *vectorNoise*.

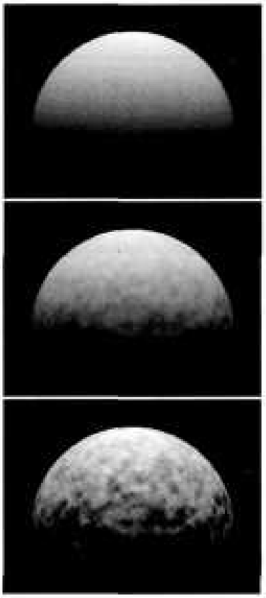


**Figure 11.13.** Vector turbulence on a sphere of radius 1.6. Lighting directly from above. Top: $k_1 = 0$. Middle: $k_1 = 0.08$, $k_2 = 8$. Bottom: $k_1 = 0.24$, $k_2$=8.

## 11.6 Displacement Mapping

One problem with Figure 11.13 is that the bumps neither cast shadows nor affect the silhouette of the object. These limitations occur because we are not really changing any geometry. If we want more realism, we can apply a *displacement map* (Cook et al., 1987). A displacement map actually changes the geometry using a texture. A common simplification is that the displacement will be in the direction of the surface normal.

If we take all points **p** on a surface, with associated surface normal vectors **n**, then we can make a new surface using a 3D texture $d(\mathbf{p})$:

$$\mathbf{p}' = \mathbf{p} + f(\mathbf{p})\mathbf{n}.$$

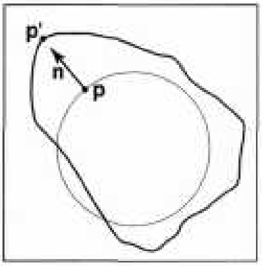


**Figure 11.14.** The points **p** on the circle are each displaced in the direction of **n** by the function $f(\mathbf{p})$. If $f$ is continuous, then the resulting points $\mathbf{p}'$ form a continuous surface.

This concept is shown in Figure 11.14.

Displacement mapping is straightforward to implement in a z-buffer code by storing the surface to be displaced as a fine mesh of many triangles. Each vertex in the mesh can then be displaced along the normal vector direction. This results in large models, but it is quite robust.

## 11.7 Environment Maps

Often we would like to have a texture-mapped background and for objects to have specular reflections of that background. This can be accomplished using

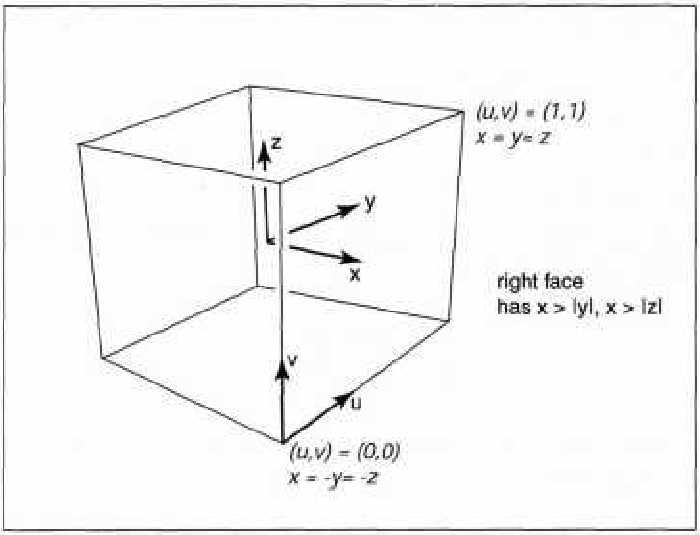


**Figure 11.15.** The cube map has six axis-aligned textures that store the background. The right face contains a single texture.

*environment maps* (J. F. Blinn, 1976). An environment map can be implemented as a background function that takes in a viewing direction $\mathbf{b}$ and returns an RGB color from a texture map. There are many ways to store environment maps. For example, we can use a spherical table indexed by spherical coordinates. In this section, we will instead describe a cube-based table with six square texture maps, often called a *cube map*.

The basic idea of a cube map is that we have an infinitely large cube with a texture on each face. Because the cube is large, the origin of a ray does not change what the ray "sees." This is equivalent to an arbitrarily-sized cube that is queried by a ray whose origin is at the Cartesian origin. As an example of how a given direction $\mathbf{b}$ is converted to $(u, v)$ coordinates, consider the right face of Figure 11.15. Here we have $x_b$ as the maximum magnitude component. In that case, we can compute $(u, v)$ for that texture to be

$$u = \frac{y + x}{2x},$$
$$v = \frac{z + x}{2x}.$$

There are analogous formulas for the other five faces.

So for any reflection ray $\mathbf{a} + t\mathbf{b}$ we return *cubemap*($\mathbf{b}$) for the background color. In a z-buffer implementation, we need to perform this calculation on a pixel-by-pixel basis. If at a given pixel we know the viewing direction $\mathbf{c}$ and the surface normal vector $\mathbf{n}$, we can compute the reflected direction $\mathbf{b}$ (Figure 11.16). We can do this by modifying Equation 9.6 to get

$$\mathbf{b} = -\mathbf{c} + \frac{2(\mathbf{c} \cdot \mathbf{n})\mathbf{n}}{\|\mathbf{c}\|^2}. \tag{11.8}$$

Here the denominator of the fraction accounts for the fact that $\mathbf{c}$ may not be a unit vector. Because we need to know $\mathbf{b}$ at each pixel, we can either compute $\mathbf{b}$ at each triangle vertex and interpolate $\mathbf{b}$ in a perspective correct manner, or we can interpolate $\mathbf{n}$ and compute $\mathbf{b}$ for each pixel. This will allow us to call *cubemap*($\mathbf{b}$) at each pixel.

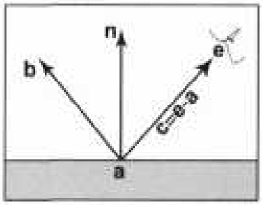


**Figure 11.16.** The vector **b** is the reflection of vector **c** with respect to the surface normal **n**.

## 11.8 Shadow Maps

The basic observation to be made about a shadow map is that if we rendered the scene using the location of a light source as the eye, the visible surfaces would all be lit, and the hidden surfaces would all be in shadow. This can be used to determine whether a point being rasterized is in shadow (L. Williams, 1978). First, we rasterize the scene from the point of view of the light source using matrix $\mathbf{M}_s$. This matrix is just the same as the full transform matrix $\mathbf{M}$ used for viewing in Section 7.3, but it uses the light position for the eye and the light's main direction for the view-plane normal.

Recall that the matrix $\mathbf{M}$ takes an $(x, y, z)$ in world coordinates and converts it to an $(x', y', z')$ in relation to the screen. While rasterizing in a perspectively correct manner, we can get the $(x, y, z)$ that is seen through the center of each pixel. If we also rasterize that point using $\mathbf{M}_s$ and round the resulting $x$ and $y$ coordinates, we will get

$$(i, j, \text{depth}).$$

We can compare this depth with the $z$ value in the shadow depth map at pixel $(i, j)$. If it is the same, then the point is lit, and otherwise it is in shadow. Because of computational inaccuracies, we should actually test whether the points are the same to within a small constant.

Because we typically don't want the light to only be within a square window, often a *spot light* is used. This attenuates the value of the light source based on closeness to the sides of the shadow buffer. For example, if the shadow buffer is $n$ by $n$ pixels, then for pixel $(i, j)$ in the shadow buffer, we can apply the attenuation coefficient based on the fractional radius $r$:

$$r = \sqrt{\left(\frac{2i-n}{n}\right)^2 + \left(\frac{2j-n}{n}\right)^2}.$$

Any radially decreasing function will then give a spot-like look.

## Frequently Asked Questions

- How do I implement displacement mapping in ray tracing?

There is no ideal way to do it. Generating all the triangles and caching the geometry when necessary will prevent memory overload (Pharr & Hanrahan, 1996; Pharr, Kolb, Gershbein, & Hanrahan, 1997). Trying to intersect the displaced surface directly is possible when the displacement function is restricted (Patterson, Hoggar, & Logie, 1991; Heidrich & Seidel, 1998; Smits, Shirley, & Stark, 2000).

- Why don't my images with textures look realistic?

Humans are good at seeing small imperfections in surfaces. Geometric imperfections are typically absent in computer-generated images that use texture maps for details, so they look "too smooth."

- My textured animations look bad when there are many texels visible inside a pixel. What should I do?

The problem is that the texture resolution is too high for that image. We would like a smaller down-sampled version of the texture. However, if we move closer, such a down-sampled texture would look too blurry. What we really need is to be able to dynamically choose the texture resolution based on viewing conditions so that about one texel is visible through each pixel. A common way to do that is to use *MIP-mapping* (L. Williams, 1983). That technique establishes a multiresolution set of textures and chooses one of the textures for each polygon or pixel. Typically the resolutions vary by a factor of two, e.g., $512^2$, $256^2$, $128^2$, etc.

## Notes

The discussion of perspective-correct textures is based on *Fast Shadows and Lighting Effects Using Texture Mapping* (Segal, Korobkin, Widenfelt, Foran, & Haeberli, 1992) and on *3D Game Engine Design* (Eberly, 2000).

## Exercises

1. Find several ways to implement an infinite 2D checkerboard using surface and solid techniques. Which is best?

2. Verify that Equation 11.4 is a valid equality using brute-force algebra.

3. How could you implement solid texturing by using the z-buffer depth and a matrix transform?

# 12

# A Full Graphics Pipeline

So far we have covered how to rasterize triangles and how to use transformation matrices and z-buffers/BSP trees to create perspective views of 3D triangles. Although this is the core of most modern graphics systems, there are a number of details that must be addressed before our system is complete. We have not yet addressed the case where some or all of a triangle is outside the view volume; this is handled by a process called "clipping;" parts of triangles outside the view-volume are cut away or "clipped" (Sutherland et al., 1974; Cyrus & Beck, 1978; J. Blinn & Newell, 1978; Liang & Barsky, 1984). The other important details in this chapter are related to improving efficiency and appearance in a graphics pipeline.

## 12.1 Clipping

A common operation in graphics is *clipping*, where one geometric entity "cuts" another. For example, if you clip a triangle against the plane $x = 0$, the plane cuts the triangle. In most applications of clipping, the portion of the triangle on the "wrong" side of the plane is discarded. Here the wrong side is whichever side is specified by the details of the application. This operation for a single plane is shown in Figure 12.1.

This section discusses the basic implementation of a clipping module. Those interested in implementing an industrial-speed clipper should see the book by Blinn mentioned in the notes at the end of this chapter.

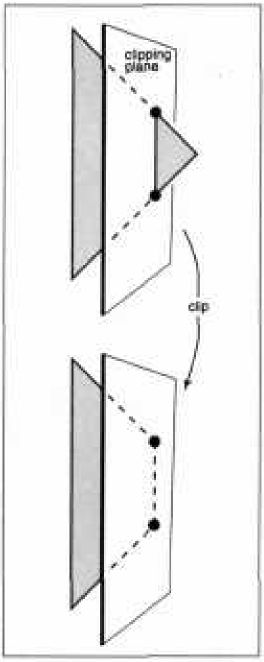


**Figure 12.1.** A polygon is clipped against a clipping plane. The portion "inside" the plane is retained.

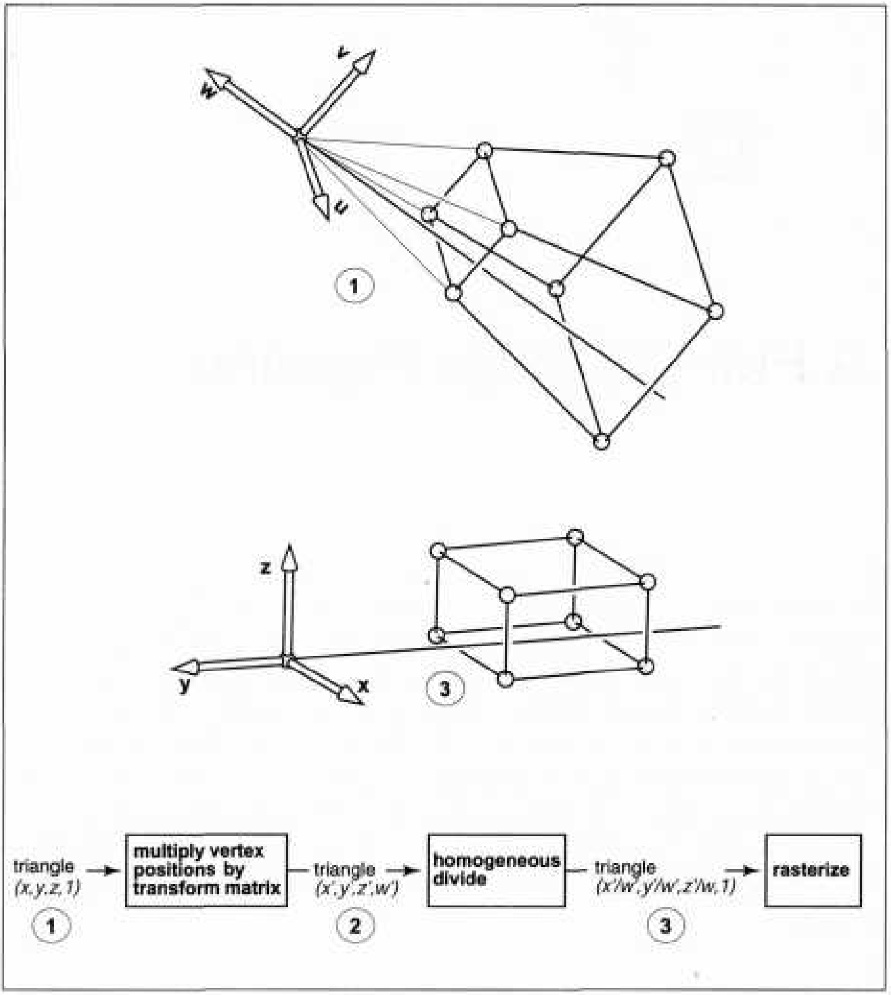


**Figure 12.2.** A bare-bones graphics pipeline with three possibilities for where to do clipping. The geometry associated with Options 1 and 3 are illustrated above the pipeline. The geometry associated with Option 2 is four-dimensional and thus too hard to depict.

## 12.2 Location of Clipping Segment of the Pipeline

The basic graphics pipeline takes triangles with vertices in world coordinates and:

- multiplies each vertex by a transformation matrix,
- divides each component of the vertex by its homogeneous coordinate,
- rasterizes the triangle.

The big question for clipping is where in the pipeline to do it. The possible locations are shown in Figure 12.2; they are:

1. in world coordinates using the six planes that bound the truncated viewing pyramid,
2. in the 4D transformed space before the homogeneous divide,
3. in the transformed 3D space with respect to the six axis-aligned planes.

Any of the possibilities can be effectively implemented (J. Blinn, 1996). For all of them, the triangle-based implementation for a single triangle is:

```
for each of six planes do
  if (triangle entirely outside of plane) then
    break (triangle is not visible)
  else if triangle spans plane then
    clip triangle
    if (quadrilateral is left) then
      break into two triangles
```

The only question is which six planes to use at what stage of the pipeline.

### 12.2.1 Clipping After the Perspective Divide (Option 3)

At first glance, it seems that Option 3 is the easiest to implement and the most efficient. The six plane equations are simple and efficient to evaluate:

$$\begin{aligned} -x + l &= 0 \\ x - r &= 0 \\ -y + b &= 0 \\ y - t &= 0 \\ -z + n &= 0 \\ z - f &= 0 \end{aligned}$$

These plane equations are set up to be positive for any point outside the view volume. However, Option 3 is in fact the most problematic for a subtle reason (see Figure 12.3). Although the perspective transform does preserve depth order for depths greater than zero, it has a discontinuity at zero depth. Recall the actual transform after the homogeneous divide:

$$z' = n + f - \frac{fn}{z}.$$

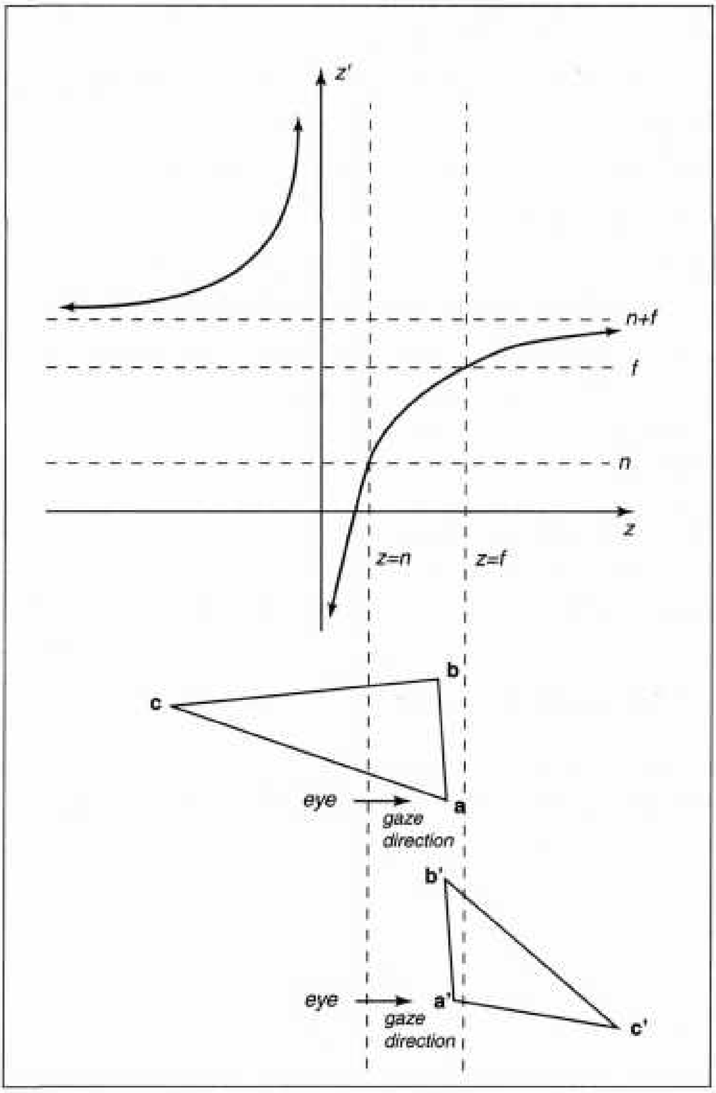


**Figure 12.3.** The depth $z$ is transformed to the depth $z'$ by the perspective transform. Note that when $z$ moves from positive to negative, $z'$ switches from negative to positive. Thus vertices behind the eye are moved in front of the eye beyond $z' = n + f$. This can make clipping complicated, for example, for the triangle shown.

### 12.2.2 Clipping Before the Transform (Option 1)

Option 1 has a straightforward implementation. The only question is, "What are the six plane equations?" Because these equations are the same for all triangles rendered in the single image, we do not need to compute them very efficiently. For this reason, we can just invert the transform shown in Figure 5.11 and apply it to the eight vertices of the transformed view volume:

$$\begin{aligned}(x, y, z) = &(l, b, n)\\ &(r, b, n)\\ &(l, t, n)\\ &(r, t, n)\\ &(l, b, f)\\ &(r, b, f)\\ &(l, t, f)\\ &(r, t, f)\end{aligned}$$

The plane equations can be inferred from here. Alternatively, we can use vector geometry to get the planes directly from the viewing parameters.

### 12.2.3 Clipping in Homogeneous Coordinates (Option 2)

Surprisingly, the option usually implemented is that of clipping in homogeneous coordinates before the divide. Here the view volume is 4D, and it is bounded by 3D volumes (hyperplanes). These are:

$$\begin{aligned}-x + lw &= 0\\ x - rw &= 0\\ -y + bw &= 0\\ y - tw &= 0\\ -z + nw &= 0\\ z - fw &= 0\end{aligned}$$

These planes are quite simple, so the efficiency is better than for Option 1. They still can be improved by transforming the view volume $[l, r] \times [b, t] \times [n, f]$ to $[0, 1]^3$. It turns out that the clipping of the triangles is not much more complicated than in 3D.

### 12.2.4 Clipping Against a Plane

No matter which option we choose, we must clip against a plane. Recall from Section 2.7.2 that the implicit equation for a plane through point $\mathbf{q}$ with normal $\mathbf{n}$ is

$$f(\mathbf{p}) = \mathbf{n} \cdot (\mathbf{p} - \mathbf{q}) = 0.$$

This is often written

$$f(\mathbf{p}) = \mathbf{n} \cdot \mathbf{p} + D = 0. \tag{12.1}$$

Interestingly, this equation not only describes a 3D plane, but it also describes a line in 2D and the volume analog of a plane in 4D. All of these entities are usually called planes in their appropriate dimension.

If we have a line segment between points $\mathbf{a}$ and $\mathbf{b}$, we can "clip" it against a plane using the techniques for cutting the edges of 3D triangles in BSP tree programs described in Section 8.1.3. Here, the points $\mathbf{a}$ and $\mathbf{b}$ are tested to determine whether they are on opposite sides of the plane $f(\mathbf{p}) = 0$ by checking whether $f(\mathbf{a})$ and $f(\mathbf{b})$ have different signs. Typically $f(\mathbf{p}) < 0$ is defined to be "inside" the plane, and $f(\mathbf{p}) > 0$ is "outside" the plane. If the plane does split the line, then we can solve for the intersection point by substituting the equation for the parametric line,

$$\mathbf{p} = \mathbf{a} + t(\mathbf{b} - \mathbf{a}),$$

into the $f(\mathbf{p}) = 0$ plane of Equation 12.1. This yields

$$\mathbf{n} \cdot (\mathbf{a} + t(\mathbf{b} - \mathbf{a})) + D = 0.$$

Solving for $t$ gives

$$t = \frac{\mathbf{n} \cdot \mathbf{a} + D}{\mathbf{n} \cdot (\mathbf{a} - \mathbf{b})}.$$

We can then find the intersection point and "shorten" the line.

To clip a triangle, we again can follow Section 8.1.3 to produce one or two triangles .

## 12.3 An Expanded Graphics Pipeline

A full graphics pipeline typically adds shading and some efficiency modes to make a full system. These pipelines are usually optimized to render large numbers of small triangles. This section discusses several issues which must be handled in the pipeline.

### 12.3.1 Culling

When the entire triangle lies outside the view volume, it can be *culled*, i.e., eliminated from the pipeline. In practice, perfect culling is more expensive than letting the clipping module eliminate the object. Culling is especially helpful when many triangles are grouped into an object with an associated bounding volume. If the bounding volume lies outside the view volume, then so do all the triangles that make up the object. For example, if we have 1000 triangles bounded by a single sphere with center $\mathbf{c}$ and radius $r$, we can check whether the sphere lies outside the clipping plane,

$$(\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0,$$

where $\mathbf{a}$ is a point on the plane, and $\mathbf{p}$ is a variable. This is equivalent to checking whether the signed distance from the center of the sphere $\mathbf{c}$ to the plane is greater than $+r$. This amounts to the check that

$$\frac{(\mathbf{c} - \mathbf{a}) \cdot \mathbf{n}}{||\mathbf{n}||} > r.$$

Note that the sphere may overlap the plane even in a case where all the triangles do lie outside the plane. Thus, this is a conservative test. How conservative the test is depends on how well the sphere bounds the object.

### 12.3.2 Lighting and Shading

Lighting and shading must be placed somewhere in the pipeline. Traditionally, lighting has been done at vertices early in the pipeline, and pixels between the vertices have their colors set using barycentric interpolation at the rasterization stage. However, if normal interpolation is used at the rasterization stage, then it makes more sense to defer lighting until the rasterization stage. This is the trend in most modern pipelines.

## 12.4 Backface Elimination

When polygonal models are closed, i.e., they bound a closed space with no holes, then they are often assumed to have outward facing normal vectors as discussed in Chapter 9. For such models, the polygons that face away from the eye are certain to be overdrawn by polygons that face the eye. Thus, those polygons can be culled before the pipeline even starts. The test for this condition is the same one used for silhouette drawing given in Section 9.3.1.

## 12.5 Triangle Strips and Fans

Two fundamental primitives associated with most modern graphics pipelines are *triangle strips* and *triangle fans*.

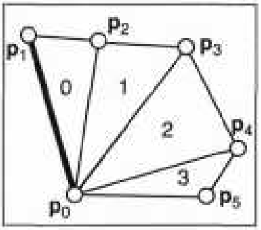


**Figure 12.4.** A triangle fan.

A triangle fan is shown in Figure 12.4. In a simple triangle-drawing pipeline, the needed calls would be:

draw $(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)$
draw $(\mathbf{p}_0, \mathbf{p}_2, \mathbf{p}_3)$
draw $(\mathbf{p}_0, \mathbf{p}_3, \mathbf{p}_4)$
draw $(\mathbf{p}_0, \mathbf{p}_4, \mathbf{p}_5)$

Note that this requires twelve vertices to pass through the pipeline, although there are only six distinct vertices. The triangle fan assumes an axis vertex and a series of other vertices being swept out like the ends of a collapsible fan. This function call is something like:

triangle-fan $(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4, \mathbf{p}_5)$

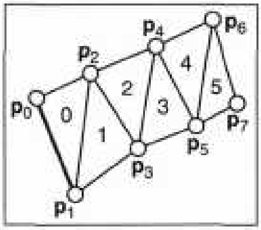


**Figure 12.5.** A triangle strip.

The first vertex is assumed to be the axis. Often the API will have a function call for each vertex so that a variable number of arguments in not required in the call.

The triangle strip is a similar concept, but it is designed for more traditional meshes. Here, vertices are added alternating top and bottom in a linear strip as shown in Figure 12.5. Long triangle strips will save approximately a factor of three if the program is vertex-bound.

It might seem that triangle strips are only useful if the strips are long. However, that is not the case. If the runtime is proportional to the number of vertices transferred, the savings are as follows:

| strip length | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 16 | 100 | $\infty$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| relative time | 1.00 | 0.67 | 0.56 | 0.50 | 0.47 | 0.44 | 0.43 | 0.42 | 0.38 | 0.34 | 0.33 |

So, in fact, there is a rather rapid diminishing return as the strips grow longer, and the optimal strip size in practice should not exceed the buffer size on the hardware. Thus, even for an unstructured mesh, it is worthwhile to use some greedy algorithm to gather them into short strips.

## 12.6 Preserved State

When processing vertices for scan conversion and lighting, it is necessary to know the shading mode, e.g., is lighting on or off. In addition, it is necessary to know the color and/or material properties of each vertex. Many APIs support the ability

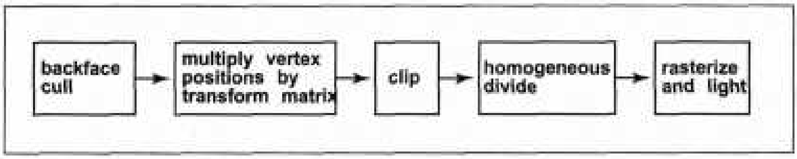


**Figure 12.6.** A full graphics pipeline. Polygons enter at the left and pass one by one through each stage of the pipeline.

to share such state information across triangles. For example, a simple implementation that does not share state information might have the calls:

set-triangle-attributes $T_1$
draw-triangle $T_1$
set-triangle-attributes $T_2$
draw-triangle $T_2$

Here both $T_1$ and $T_2$ are processed through the graphics pipeline each with its own set of attributes, such as vertex color. Instead, in APIs/hardware that support the sharing feature, we can set the state for all triangles going through the pipeline:

set-state-triangle-attributes
draw-triangle $T_1$
draw-triangle $T_2$

Here both triangles are drawn with the same attributes. This can result in a considerable efficiency increase on some hardware.

Another state that can be saved is the geometry itself. This is useful in applications where a significant fraction of the geometry does not change from frame to frame. The geometry is saved in a *display list*. This can result in increased efficiency for two reasons. First, if the hardware allows, the display list can live on the graphics board and does not need to be transferred across the bus from main memory to the graphics board each frame. Second, the list can be optimized to improve performance. For example, triangles that share the same vertex/shading properties can be grouped so that they can live within the same set of calls with shared attributes.

## 12.7 A Full Graphics Pipeline

As mentioned in the clipping section, the pipeline can be arranged in a number of ways. The modern trend seems to be to move shading to the rasterization stage because better visual quality results. Such a pipeline is shown in Figure 12.6.

A number of other issues must be addressed in a full pipeline that the programmer should be somewhat aware of when efficiency lapses. Otherwise, they are principally in the domain of hardware designers. These include:

- Are polygons rasterized directly in the pipeline or are they triangulated at the beginning of the pipeline?
- How are textures handled? Are they stored at multiple levels of detail (e.g., MIP-maps (L. Williams, 1983))? Is there a fixed-size texture memory?

## Frequently Asked Questions

**• I've often seen clipping discussed at length, and it is a much more involved process than that described in this chapter. What is going on here?**

The clipping described in this chapter works, but lacks optimizations that an industrial-strength clipper would have. These optimizations are discussed in detail in Blinn's definitive work listed in the chapter notes.

**• The documentation for my API talks about "scene graphs" and "matrix stacks." Are these part of the graphics pipeline?**

The graphics pipeline is certainly designed with these in mind, and whether we define them as part of the pipeline is a matter of taste. This book delays their discussion until the next chapter.

## Notes

A wonderful book about designing a graphics pipeline is *Jim Blinn's Corner: A Trip Down the Graphics Pipeline* (J. Blinn, 1996)). Many nice details of the pipeline and culling are in *3D Game Engine Design* (Eberly, 2000) and *Real-Time Rendering* (Akenine-Möller & Haines, 2002).

## Exercises

1. Suppose that in the perspective transform we have $n = 1$ and $f = 2$. Under what circumstances will we have a "reversal" where a vertex before and after the perspective transform flips from in front of to behind the eye or vice-versa?

2. Is there any reason not to clip in $x$ and $y$ after the perspective divide (see Figure 11.2, stage 3)?

13

# Data Structures for Graphics

There are a variety of data structures that seem to pop up repeatedly in graphics applications. This chapter talks about three basic and unrelated data structures that are among the most common and useful. There are many variants of these data structures, but the basic ideas behind them can be conveyed using an example of each.

First the winged-edge data structure for storing tessellated geometric models is discussed (Baumgart, 1974). The winged-edge data structure is useful for managing models where the tessellation changes, such as in subdivision or simplification routines.

Next, the scene-graph data structure is presented. These are rapidly becoming well supported features of all new graphics APIs because they are so useful in managing objects and transformations.

Finally, the tiled multidimensional array is presented. Originally developed to help paging performance, such structures are now crucial for memory locality on machines regardless of whether the array fits in main memory.

## 13.1 Triangle Meshes

One of the most common model representations is a polygonal mesh as discussed in Section 11.3. When such meshes are unchanging in the program, the simple structure described in that section is usually sufficient. However, when the meshes are to be modified, more complicated data representations are needed to efficiently answer queries such as:

- given a triangle, what are the three adjacent triangles?
- given an edge, which two triangles share it?
- given a vertex, which faces share it?
- given a vertex, which edges share it?

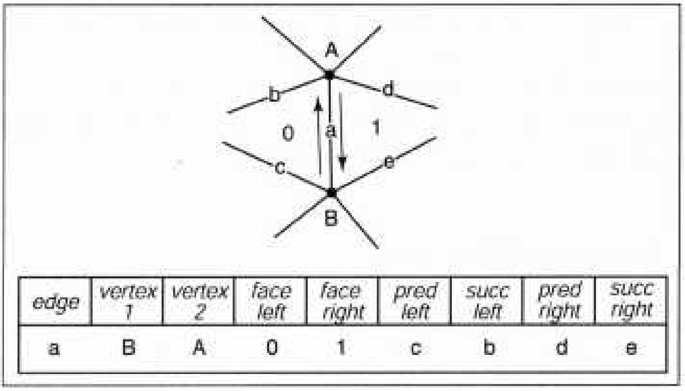
There are many data structures for triangle meshes, polygonal meshes, and polygonal meshes with holes (see the notes at the end of the chapter for references). In many applications the meshes are very large, so an efficient representation can be crucial.

The most straightforward, though bloated, implementation is to have three types: *vertex*, *edge*, and *triangle*. There are a variety of ways to divide the data among these types. While one might be tempted to just store all the relationships, this makes for variable-length data structures that really are not needed. For example, a vertex can have an arbitrarily large number of edges incident to it.

It is best, therefore, to hide the implementation behind a class interface.

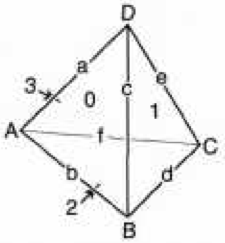
## 13.2 Winged-Edge Data Structure

We can use the class *winged-edge* data structure. This data structure makes edges the first-class citizen of the data structure. This data structure, a more efficient implementation, is illustrated in Figures 13.1 and 13.2.



| edge | vertex 1 | vertex 2 | face left | face right | pred left | succ left | pred right | succ right |
|---|---|---|---|---|---|---|---|---|
| a | B | A | 0 | 1 | c | b | d | e |

**Figure 13.1.** An edge in a winged-edge data structure. Stored with each edge are the face (polygon) to the left of the edge, the face to the right of the edge, and the previous and successor edges in the traversal of each of those faces.



| *edge* | *vertex 1* | *vertex 2* | *face left* | *face right* | *pred left* | *succ left* | *pred right* | *succ right* |
|---|---|---|---|---|---|---|---|---|
| a | A | D | 3 | 0 | f | e | c | b |
| b | A | B | 0 | 2 | a | c | d | f |
| c | B | D | 0 | 1 | b | a | e | d |
| d | B | C | 1 | 2 | c | e | f | b |
| e | C | D | 1 | 3 | d | c | a | f |
| f | C | A | 3 | 2 | e | a | b | d |

| *vertex* | *edge* |
|---|---|
| A | a |
| B | d |
| C | d |
| D | e |

| *face* | *edge* |
|---|---|
| 0 | a |
| 1 | c |
| 2 | d |
| 3 | a |

**Figure 13.2.** A tetrahedron and the associated elements for a winged-edge data structure. The two small tables are not unique; each vertex and face stores any one of the edges with which it is associated.

Note that the winged-edge data structure makes the desired queries in constant time. For example, a face can access one of its edges and follow the traversal pointers to find all of its edges. Those edges store the adjoining face.

As with any data structure, the winged-edge data structure makes a variety of time/space trade-offs. For example, we could eliminate the *prev* references. When we need to know the previous edge, we could follow the successor edges in a circle until we get back to the original edge. This would save space, but it would make the computation of the previous edge take longer. This type of issue has led to a proliferation of mesh data structures (see the chapter notes for more information on those structures).

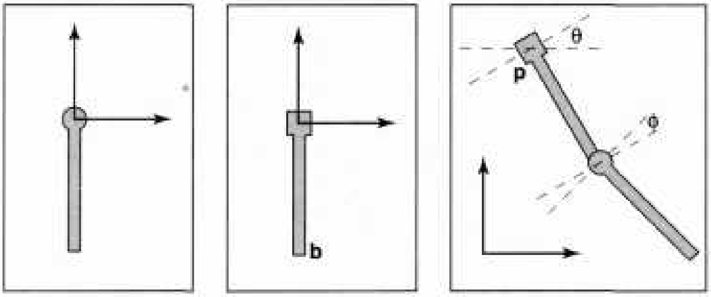


**Figure 13.3.** A hinged pendulum. On the left are the two pieces in their "local" coordinate systems. The hinge of the top piece is at point **b** and the attachment for the bottom piece is at its local origin. The degrees of freedom for the assembled object are the angles $(\theta, \phi)$ and the location **p** of the top hinge.

## 13.3 Scene Graphs

To motivate the scene-graph data structure, we will use the hinged pendulum shown in Figure 13.3. Consider how we would draw the top part of the pendulum:

$\mathbf{M}_1 = \text{rotate}(\theta)$
$\mathbf{M}_2 = \text{translate}(\mathbf{p})$
$\mathbf{M}_3 = \mathbf{M}_2\mathbf{M}_1$
Apply $\mathbf{M}_3$ to all points in upper pendulum

The bottom is more complicated, but we can take advantage of the fact that it is attached to the bottom of the upper pendulum at point $\mathbf{b}$ in the local coordinate system. First, we rotate the lower pendulum so that it is at an angle $\phi$ relative to its initial position. Then, we move it so that its top hinge is at point $\mathbf{b}$. Now it is at the appropriate position in the local coordinates of the upper pendulum, and it can then be moved along with that coordinate system. The composite transform for the lower pendulum is:

$\mathbf{M}_a = \text{rotate}(\phi)$
$\mathbf{M}_b = \text{translate}(\mathbf{b})$
$\mathbf{M}_c = \mathbf{M}_b\mathbf{M}_a$
$\mathbf{M}_d = \mathbf{M}_3\mathbf{M}_c$
Apply $\mathbf{M}_d$ to all points in lower pendulum

Thus, we see that the lower pendulum not only lives in its own local coordinate system, but also that coordinate system itself is moved along with that of the upper pendulum.

**Plate I.** The RGB color cube in 3D and its faces unfolded. Any RGB color is a point in the cube. (See also Figure 3.4.)

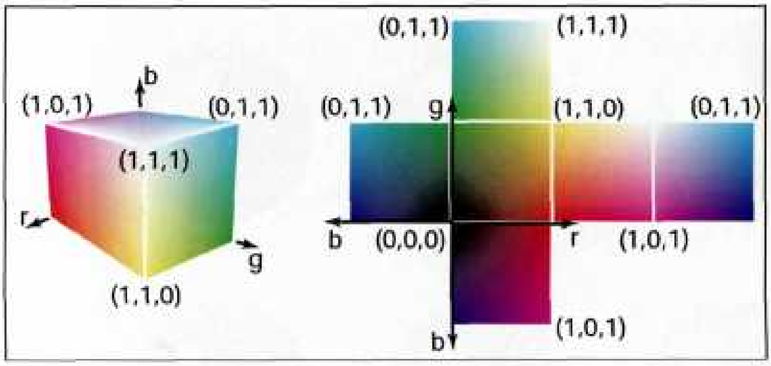


**Plate II.** A colored line switching from red to green. The middle pixel is half red and half green which is a "dark yellow". (See also Figure 3.9.)

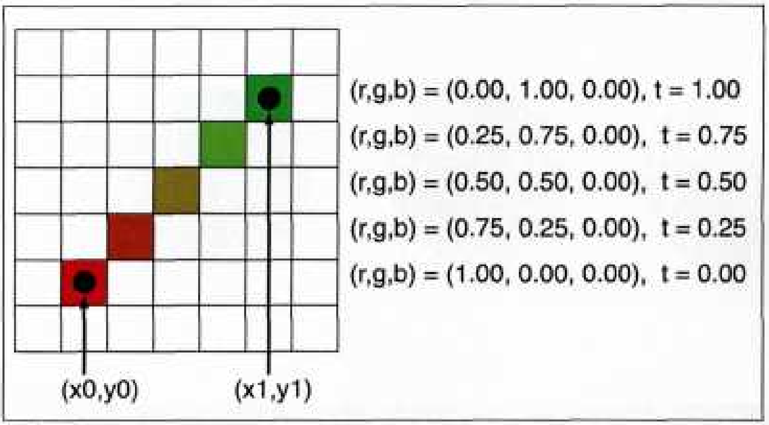
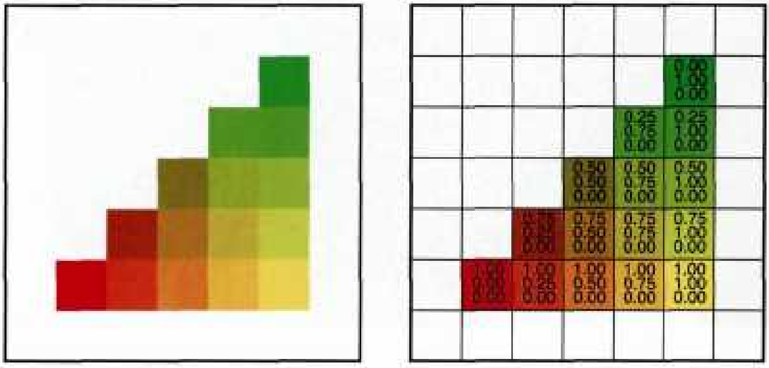

(r,g,b) = (0.00, 1.00, 0.00), t = 1.00
(r,g,b) = (0.25, 0.75, 0.00), t = 0.75
(r,g,b) = (0.50, 0.50, 0.00), t = 0.50
(r,g,b) = (0.75, 0.25, 0.00), t = 0.25
(r,g,b) = (1.00, 0.00, 0.00), t = 0.00
(x0,y0)
(x1,y1)

**Plate III.** A colored triangle with barycentric interpolation. Note that the changes in color components are linear in each row and column as well as along each edge. In fact it is constant along every line, such as the diagonals, as well. (See also Figure 3.10.)



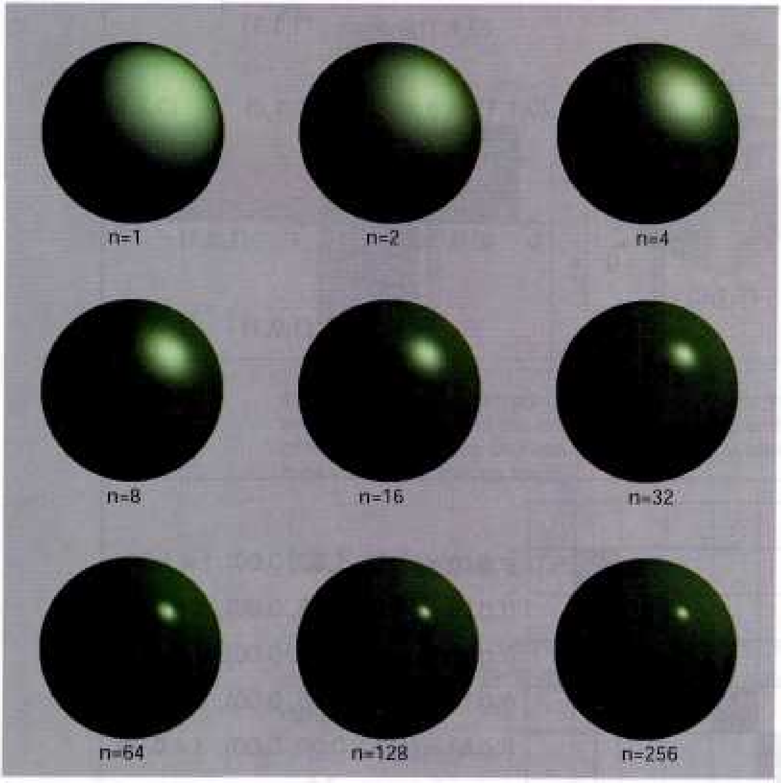


**Plate IV.** The effect of the Phong exponent on highlight characteristics. This uses Equation 9.5 for the highlight. There is also a diffuse component, giving the objects a shiny but non-metallic appearance. *Image courtesy of Nate Robins.* (See also Figure 9.6.)

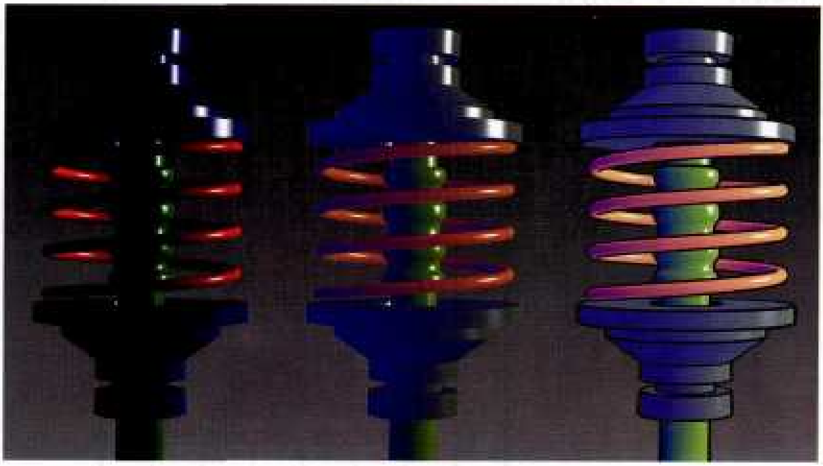


**Plate V.** Left: a Phong-illuminated image. Middle: cool-to-warm shading is not useful without silhouettes. Right: cool-to-warm shading plus silhouettes. *Image courtesy Amy Gooch.* (See also Figure 9.9.)

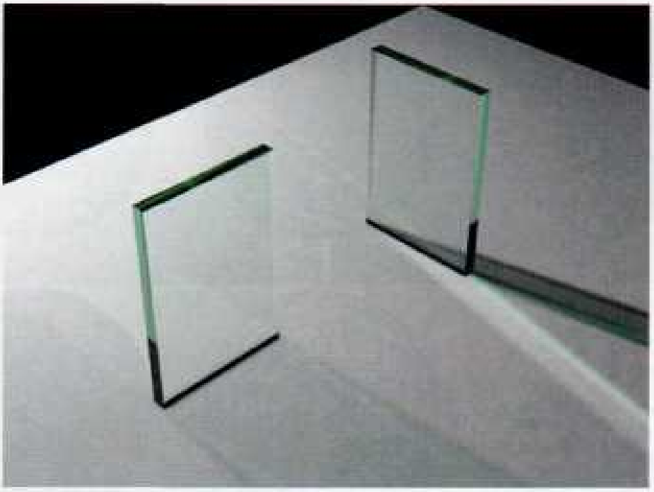


**Plate VI.** The color of the glass is affected by total internal reflection and Beer's Law. The amount of light transmitted and reflected is determined by the Fresnel Equations. The complex lighting on the ground plane was computed using particle tracing as described in Chapter 23. (See also Figure 10.11.)

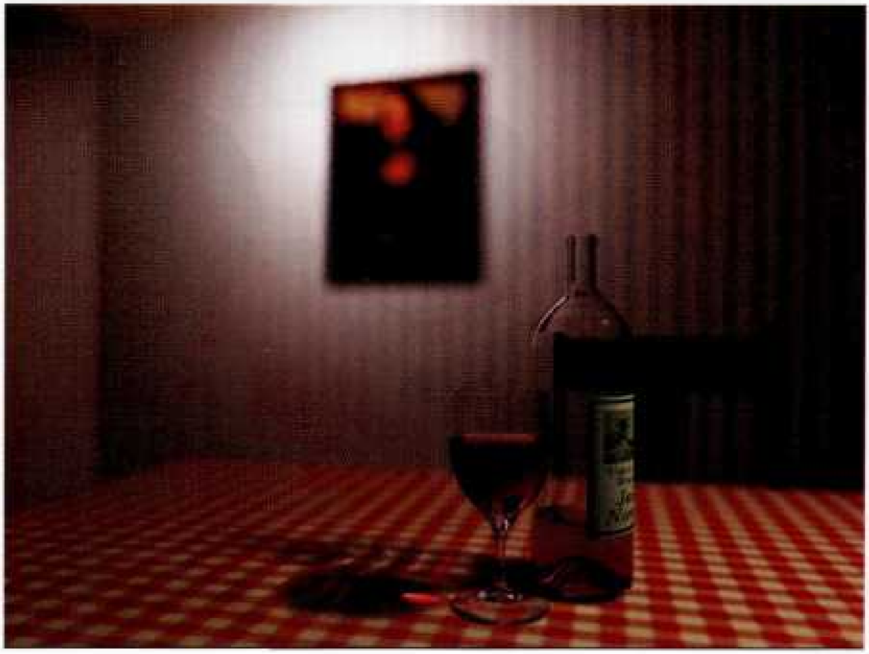


**Plate VII.** An example of depth of field. The caustic in the shadow of the wine glass is computed using particle tracing (Chapter 23). (See also Figure 10.34.)

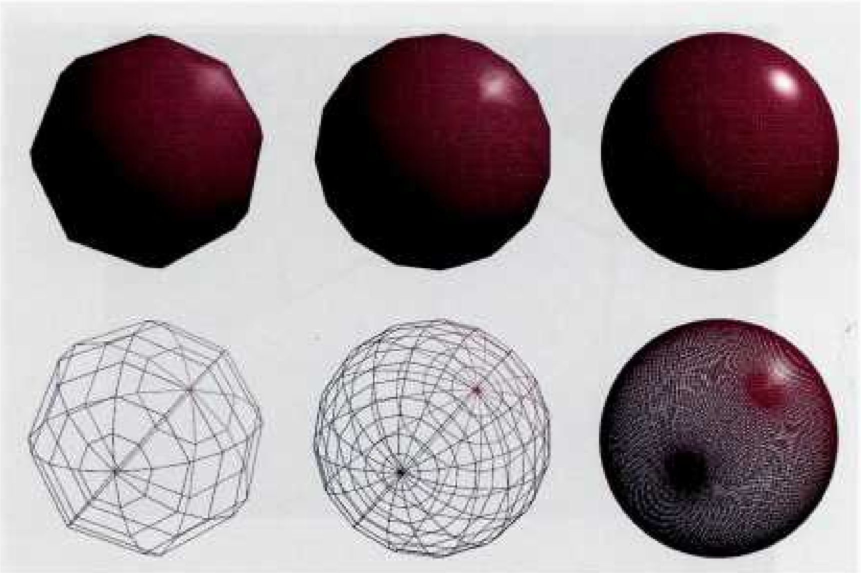


**Plate VIII.** Each sphere is rendered using only a vertex shader that computes Phong shading. Because the computation is being performed on a per-vertex basis, the Phong highlight only begins to appear accurate after the amount of geometry used to model the sphere is increased drastically. (See also Figure 17.7.)

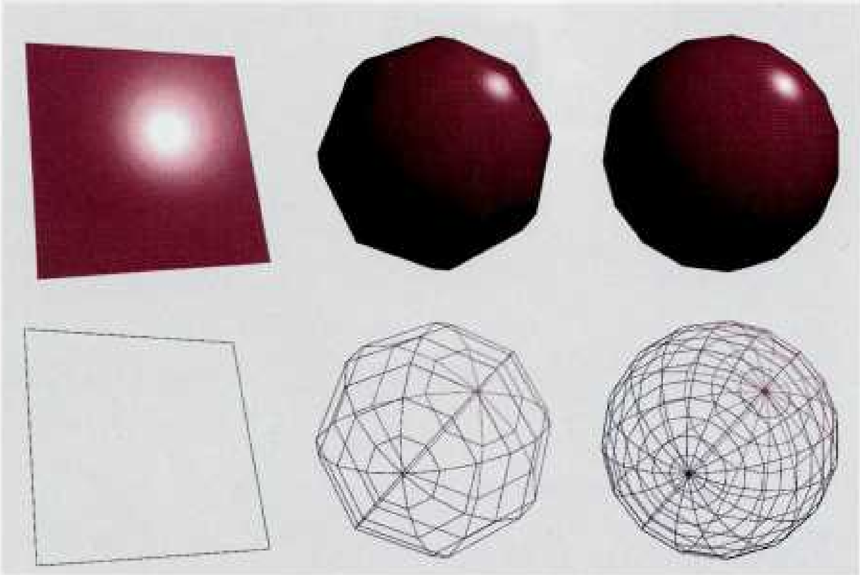
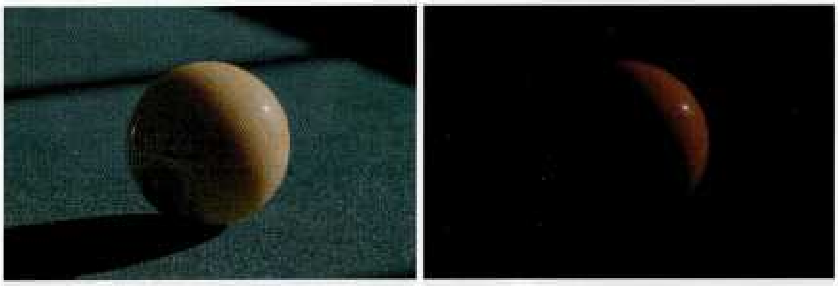


**Plate IX.** The results of running the fragment shader from Section 17.3.4. Note that the Phong highlight does appear on the left-most model which is represented by a single polygon. In fact, because lighting is calculated at the fragment, rather than at each vertex, the more coarsely tessellated sphere models also demonstrate appropriate Phong shading. (See also Figure 17.8.)

**Plate X.** Per-channel gamma correction may desaturate the image. The left image was desaturated with a value of $s = 0.5$. The right image was not desaturated ($s = 1$). (See also Figure 22.11.)



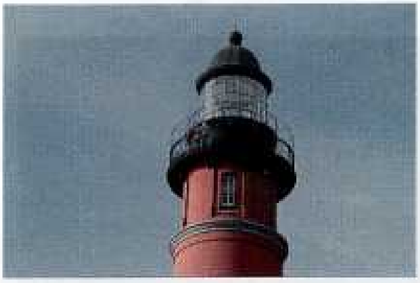
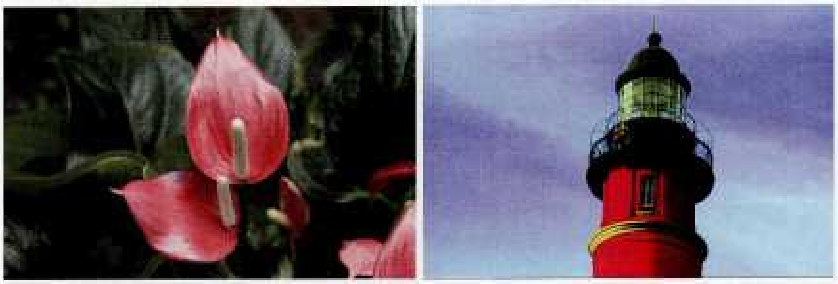


**Plate XI.** Image used for demonstrating the color transfer technique. Results are shown in Color Plates XII and XIV. (See also Figure 22.12 and Figure 22.30.)

**Plate XII.** The image on the left is used to adjust the colors of the image shown in Color Plate XI. The result is shown on the right. (See also Figure 22.13.)



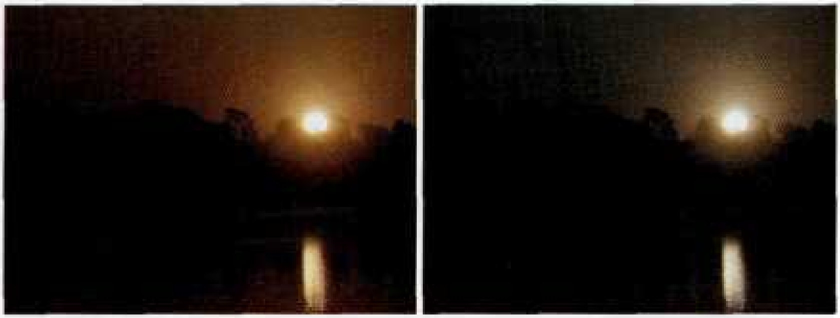


**Plate XIII.** Linear interpolation for color correction. The parameter *c* is set to 0.0 in the left image and to 1.0 in the right image. (See also Figure 22.24.)

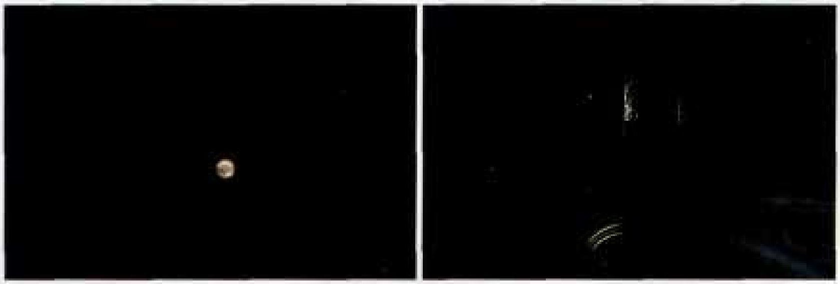
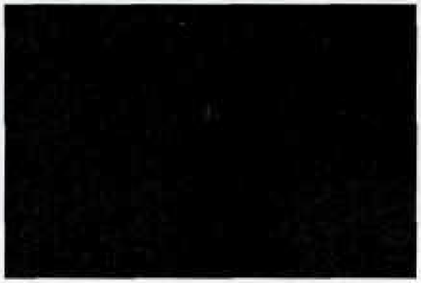


**Plate XIV.** The image on the left is used to transform the image of Color Plate XI into a night scene, shown here on the right. (See also Figure 22.31.)

**Plate XV.** Simulated night scene using the image shown in Color Plate XI. (See also Figure 22.30.)



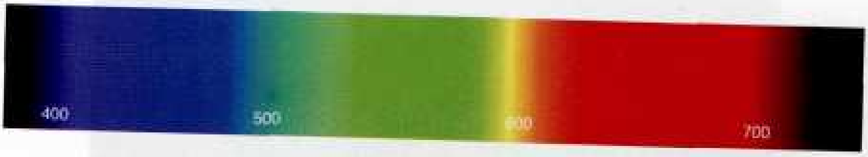


**Plate XVI.** The visible spectrum. Wavelengths are in nanometers.

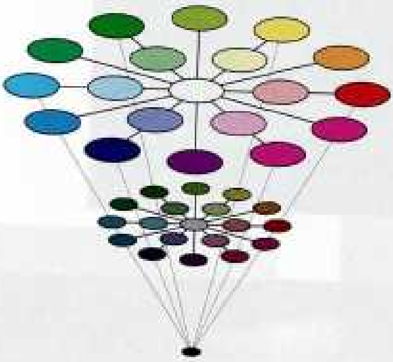


**Plate XVII.** HSV color space. Hue varies around the circle, saturation varies with radius, and value varies with height.

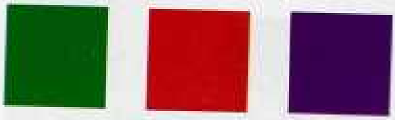


**Plate XVIII.** Which color is closer to red: green or violet?

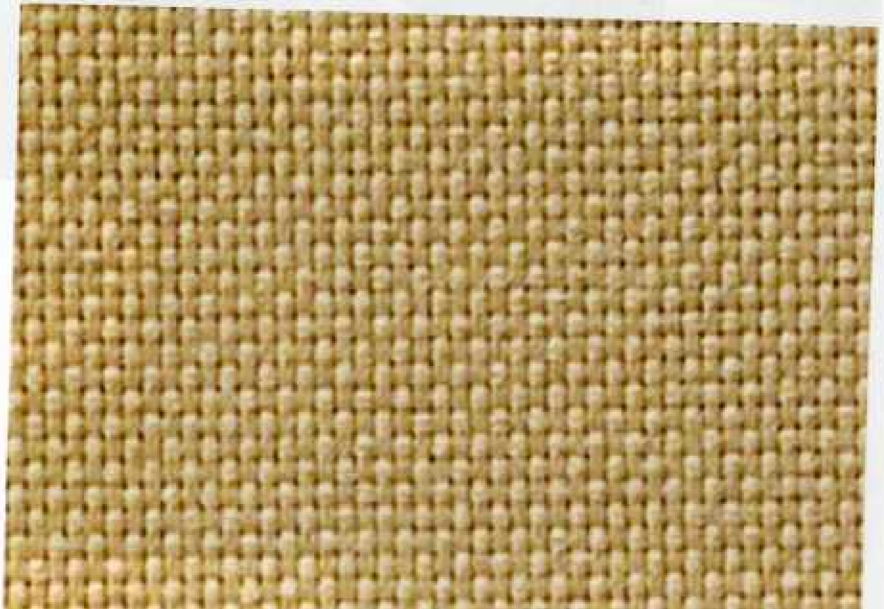


**Plate XIX.** The effect shown in Figure 21.29 is even more powerful when shown in color. *Figure courtesy of Albert Yonas.*

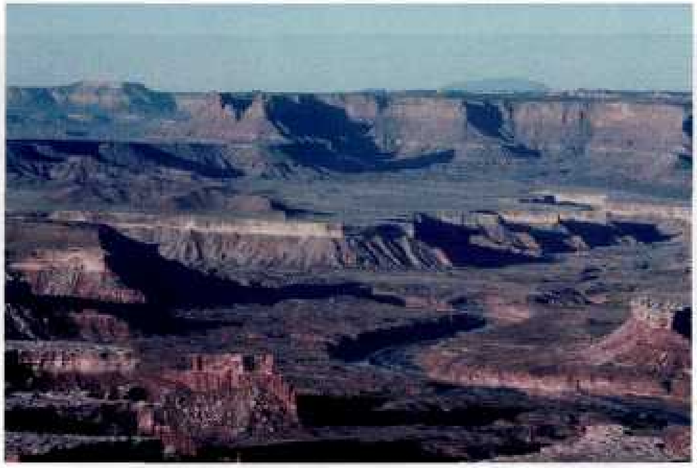


**Plate XX.** Aerial perspective, in which atmospheric effects reduce contrast and shift colors towards blue, provides a depth cue over long distances.

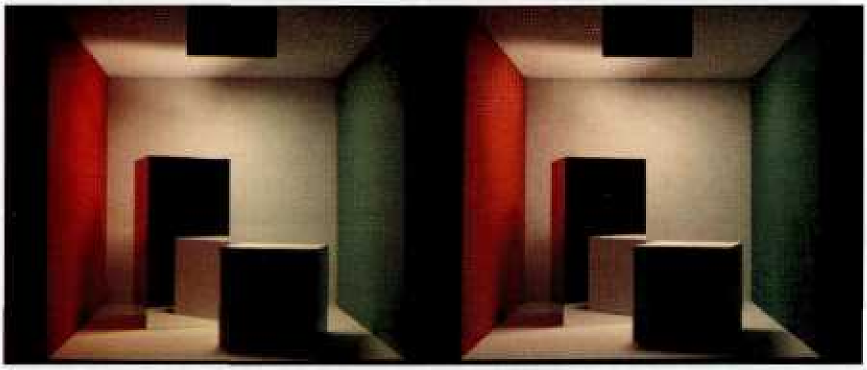


**Plate XXI.** A comparison between a rendering and a photo. *Figure courtesy Sumant Pattanaik and the Cornell Program of Computer Graphics.* (See also Figure 23.9.)

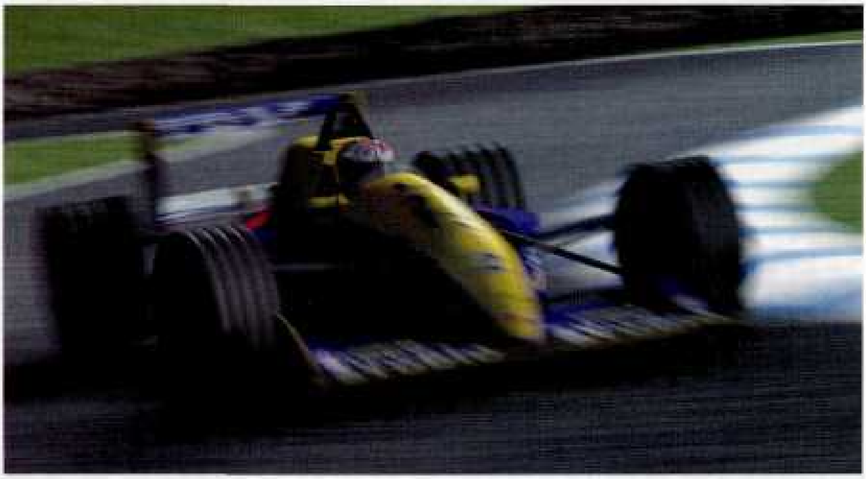


**Plate XXII.** The image shows extreme motion blur effects. The shadows use distribution ray tracing because they are moving during the image. *Model by Joseph Hamdorf and Young Song. Rendering by Eric Levin.*

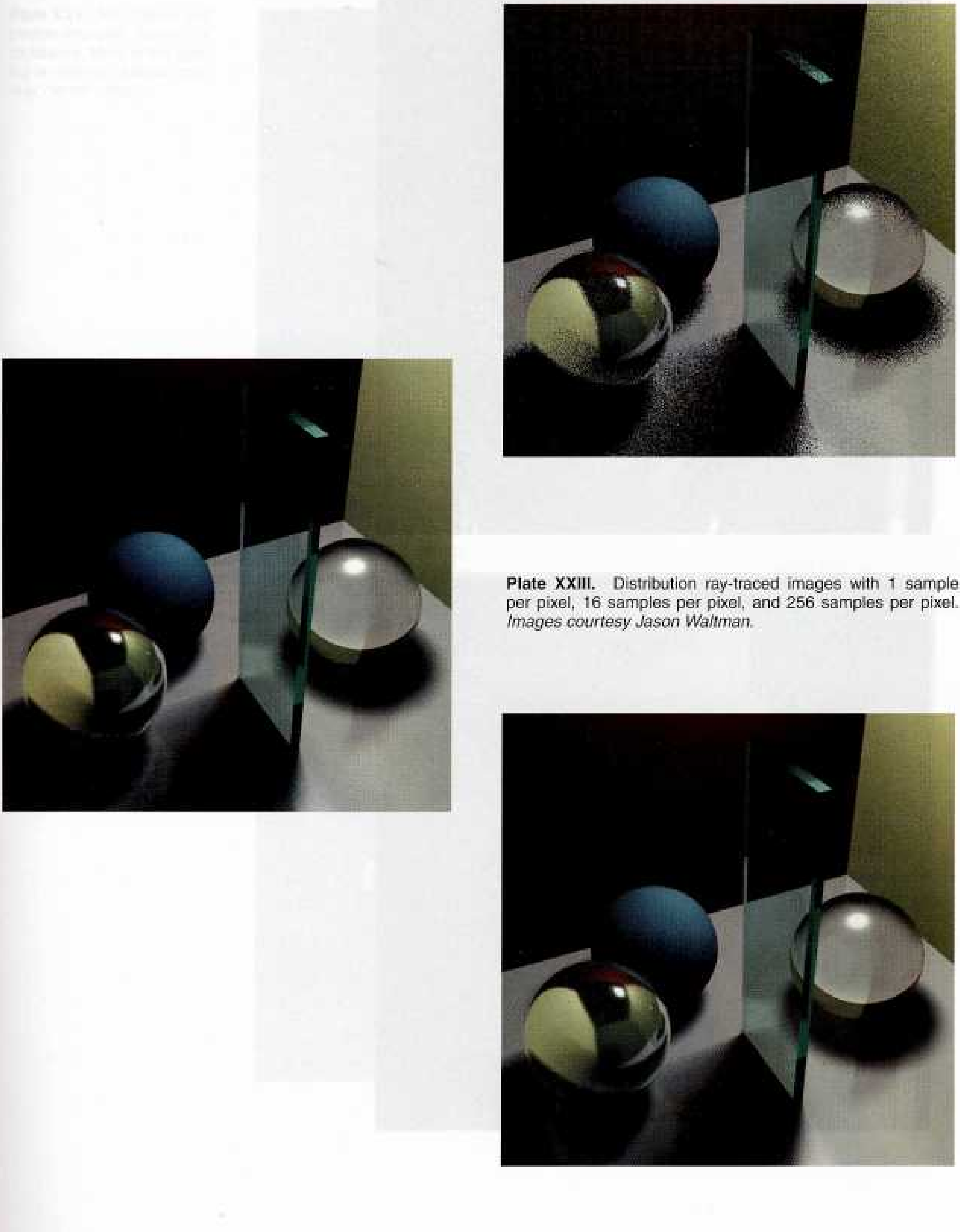
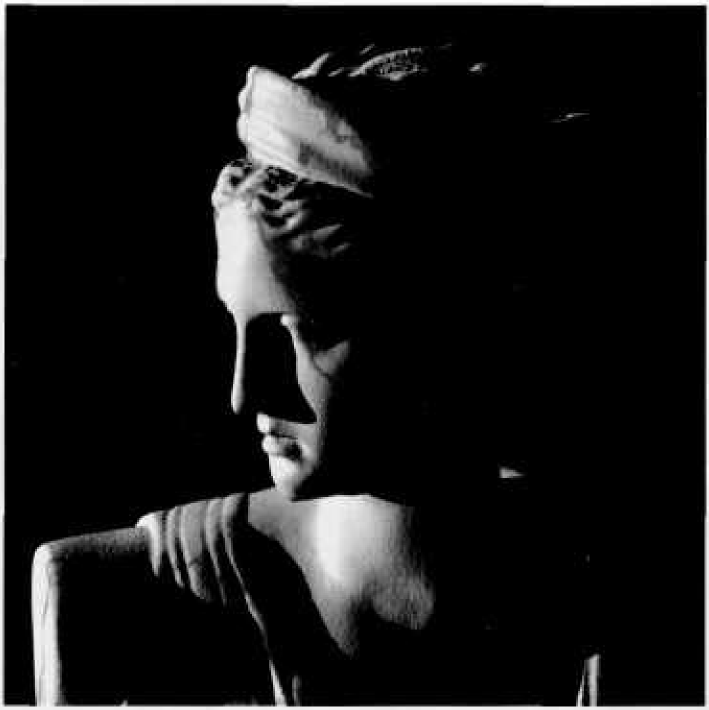


**Plate XXIII.** Distribution ray-traced images with 1 sample per pixel, 16 samples per pixel, and 256 samples per pixel. *Images courtesy Jason Waltman.*



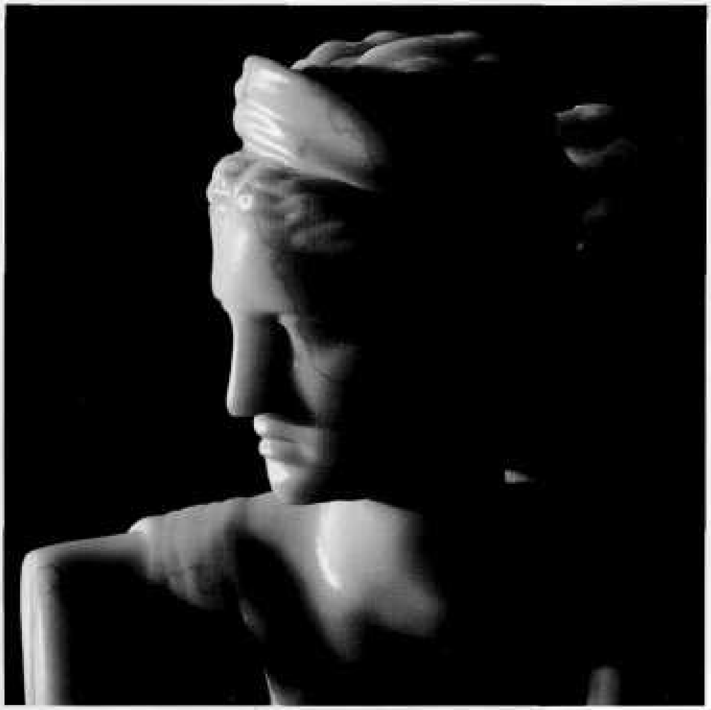


**Plate XXIV.** Top: A diffuse shading model is used. Bottom: Subsurface scattering is allowed using a technique from "A Practical Model for Sub-surface Light Transport," Jensen et al., Proceedings of SIGGRAPH 2001. *Images courtesy Henrik Jensen.*

**Plate XXV.** Ray-traced and photon-mapped image of an interior. Most of the lighting is indirect. *Image courtesy Henrik Jensen.*

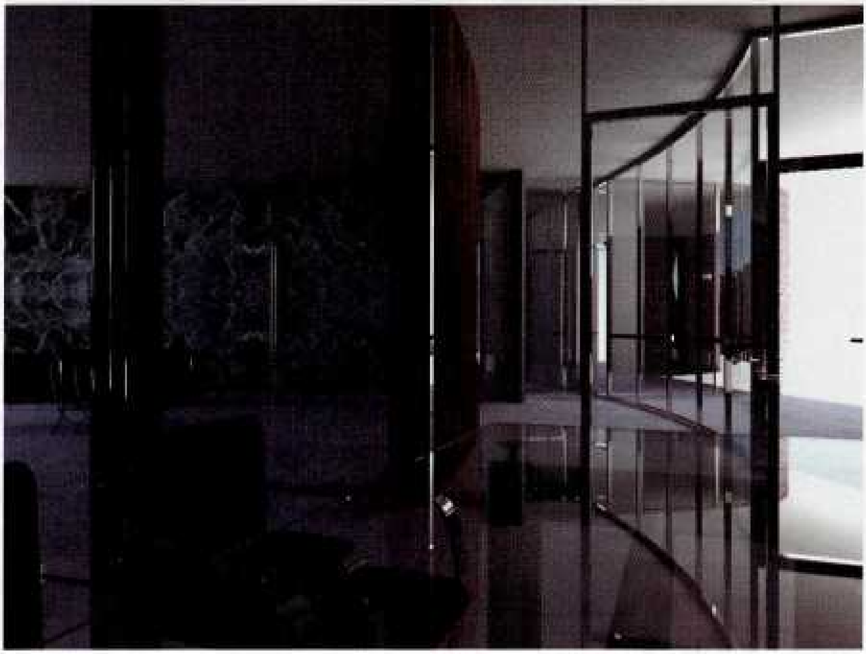
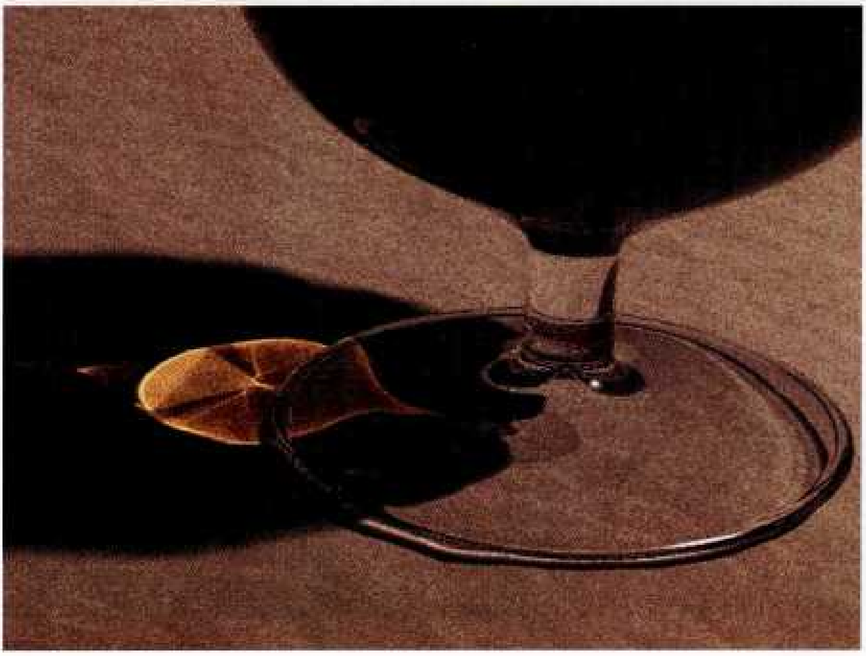


**Plate XXVI.** The brightly colored pattern in the shadow is a "caustic" and is a product of light focused through the glass. It was computed using photon tracing. *Image courtesy Henrik Jensen.*



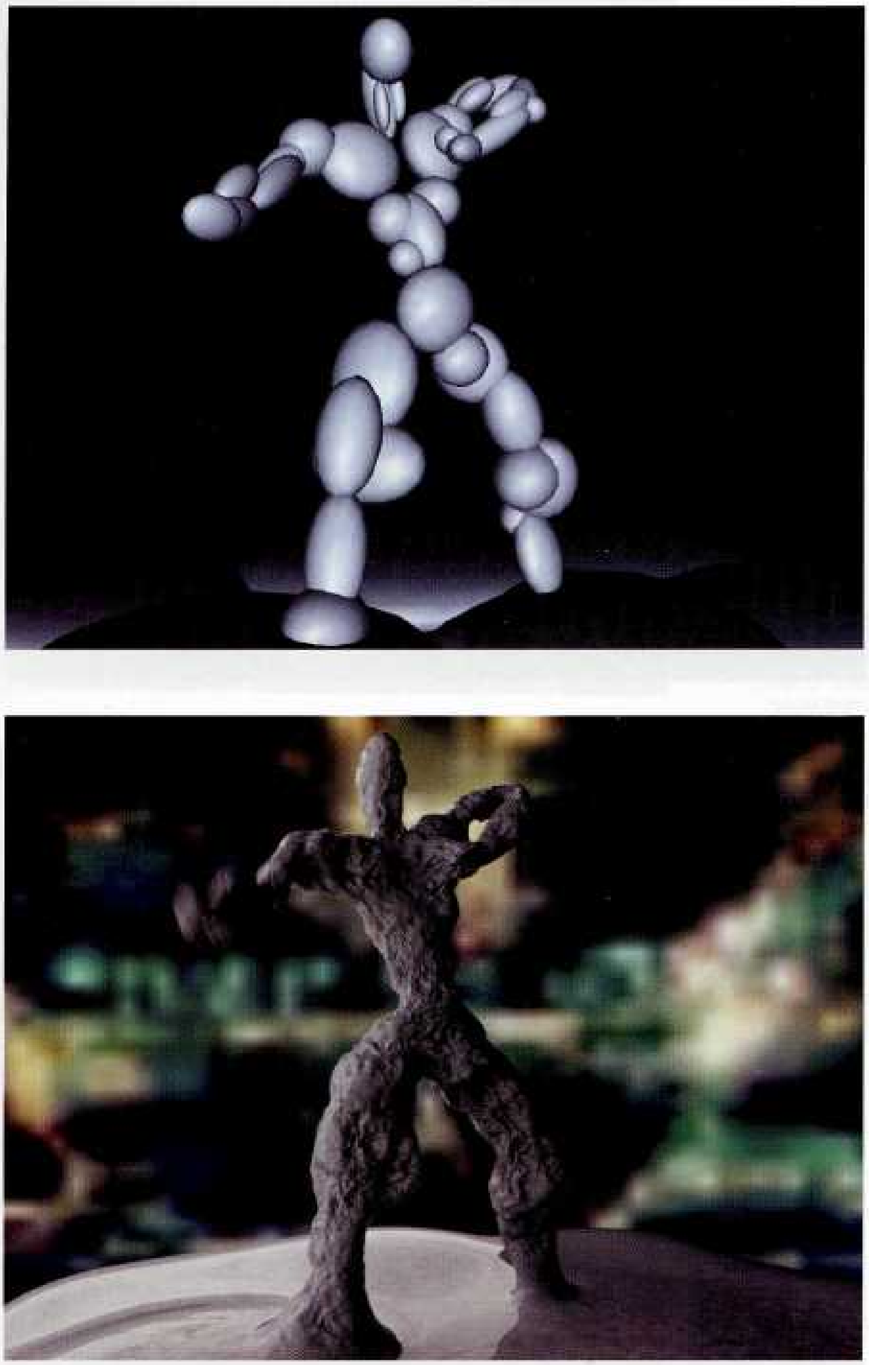


**Plate XXVII.** Top: A set of ellipsoids approximates the model. Bottom: The ellipsoids are used to crete a gravity-like implicit function which is then displaced. *Image courtesy Eric Levin.*

We can encode the pendulum in a data structure that makes management of these coordinate system issues easier, as shown in Figure 13.4. The appropriate matrix to apply to an object is just the product of all the matrices in the chain from the object to the root of the data structure. For example, consider the model of a ferry that has a car that can move freely on the deck of the ferry, and wheels that each move relative to the car as shown in Figure 13.5.

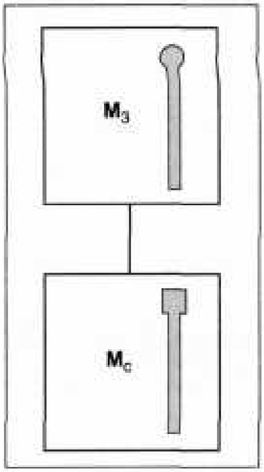


Figure 13.4. The scene graph for the hinged pendulum of Figure 13.3.

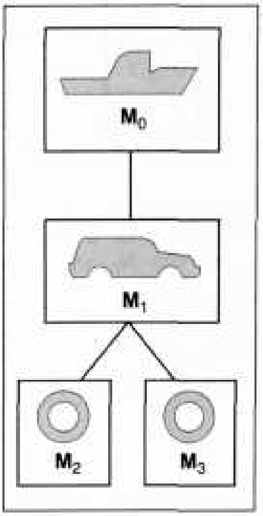


Figure 13.5. A ferry, a car on the ferry, and the wheels of the car (only two shown) are stored in a scene-graph.

As with the pendulum, each object should be transformed by the product of the matrices in the path from the root to the object:

**ferry** transform using $M_0$

**car body** transform using $M_0 M_1$

**left wheel** transform using $M_0 M_1 M_2$

**left wheel** transform using $M_0 M_1 M_3$

An efficient implementation can be achieved using a *matrix stack*, a data structure supported by many APIs. A matrix stack is manipulated using *push* and *pop* operations that add and delete matrices from the right-hand side of a matrix product. For example, calling:

```
push($\mathbf{M}_0$)
push($\mathbf{M}_1$)
push($\mathbf{M}_2$)
```

creates the active matrix $\mathbf{M} = \mathbf{M}_0\mathbf{M}_1\mathbf{M}_2$. A subsequent call to *pop()* strips the last matrix added so that the active matrix becomes: $\mathbf{M} = \mathbf{M}_0\mathbf{M}_1$. Combining the matrix stack with a recursive traversal of a scene graph gives us:

```
function traverse(node)
push($\mathbf{M}_{\text{local}}$)
draw object using composite matrix from stack
traverse(left child)
traverse(right child)
pop()
```

There are many variations on scene graphs but all follow the basic idea above.

## 13.4 Tiling Multidimensional Arrays

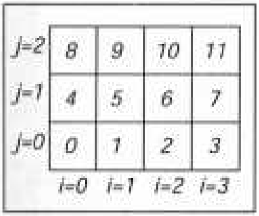


**Figure 13.6.** The memory layout for an untiled 2D array with $N_x = 4$ and $N_y = 3$.

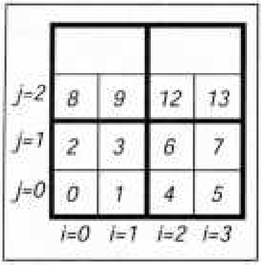


**Figure 13.7.** The memory layout for a tiled 2D array with $N_x = 4$ and $N_y = 3$ and two by two tiles. Note that padding on the top of the array is needed because $N_y$ is not a multiple of the tile size two.

Effectively utilizing the cache hierarchy is a crucial task in designing algorithms for modern architectures. Making sure that multidimensional arrays have data in a "nice" arrangement is accomplished by *tiling*, sometimes also called *bricking*. A traditional 2D array is stored as a 1D array together with an indexing mechanism; for example, an $N_x$ by $N_y$ array is stored in a 1D array of length $N_xN_y$ and the 2D index $(x, y)$ (which runs from $(0, 0)$ to $(N_x - 1, N_y - 1)$) and maps it to the 1D index (running from 0 to $N_xN_y - 1$ using the formula

$$\text{index} = x + N_x y.$$

An example of how that memory lays out is shown in Figure 13.6. A problem with this layout is that although two adjacent array elements that are in the same row are next to each other in memory, two adjacent elements in the same column will be separated by $N_x$ elements in memory. This can cause poor memory locality for large $N_x$. The standard solution to this is to use *tiles* to make memory locality for rows and columns more equal. An example is shown in Figure 13.7 where two by two tiles are used. The details of indexing such an array are discussed in the next section. A more complicated example with two levels of tiling on a 3D array are covered after that.

A key question is what size to make the tiles. In practice, they should be similar to the memory-unit size on the machine. For example, on a machine with 128-byte cache lines, and using 16-bit data values, $n$ is exactly 8. However, using float (32-bit) datasets, $n$ is closer to 5. Because there are also coarser-sized memory units such as pages, hierarchical tiling with similar logic can be useful.

### 13.4.1 One-Level Tiling for 2D Arrays

If we assume an $N_x$ by $N_y$ array decomposed into square $n$ by $n$ tiles (Figure 13.8), then the number of tiles required is

$$B_x = N_x/n,$$
$$B_y = N_y/n.$$

Here, we assume that $n$ divides $N_x$ and $N_y$ exactly. When this is not true, the array should be *padded*. For example, if $N_x = 15$ and $n = 4$, then $N_x$ should be changed to 16. To work out a formula for indexing such an array, we first find the tile indices $(b_x, b_y)$ that give the row/column for the tiles (the tiles themselves form a 2D array):

$$b_x = x \div n,$$
$$b_y = y \div n,$$

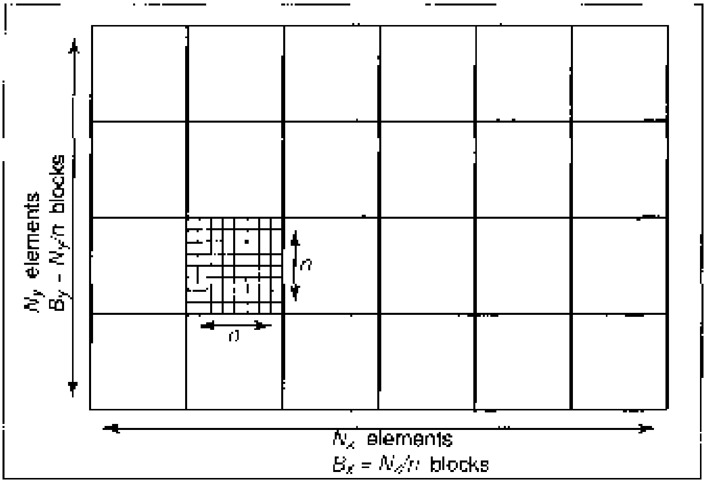


**Figure 13.8.** A tiled 2D array composed of $B_x$ by $B_y$ tiles each of size $n$ by $n$.

where ÷ is integer division, e.g., $12 \div 5 = 2$. If we order the tiles along rows as shown in Figure 13.6, then the index of the first element of the tile $(b_x, b_y)$ is

$$\text{index} = n^2(B_x b_y + b_x).$$

The memory in that tile is arranged like a traditional 2D array as shown in Figure 13.7. The partial offsets $(x', y')$ inside the tile are

$$x' = x \bmod n,$$
$$y' = y \bmod n,$$

where mod is the remainder operator, e.g., $12 \bmod 5 = 2$. Therefore, the offset inside the tile is

$$\text{offset} = y'n + x'.$$

Thus the full formula for finding the 1D index element $(x, y)$ in an $N_x$ by $N_y$ array with $n$ by $n$ tiles is

$$\begin{aligned}\text{index} &= n^2(B_x b_y + b_x) + y'n + x', \\ &= n^2((N_x \div n)(y \div n) + x \div n) + (y \bmod n)n + (x \bmod n).\end{aligned}$$

This expression contains many integer multiplication, divide and modulus operations. On modern processors, these operations are extremely costly. For $n$ that are powers of two, these operations can be converted to bitshifts and bitwise logical operations. However, as noted above, the ideal size is not always a power of two. Some of the multiplications can be converted to shift/add operations, but the divide and modulus operations are more problematic. The indices could be computed incrementally, but this would require tracking counters, with numerous comparisons and poor branch prediction performance.

However, there is a simple solution; note that the index expression can be written as

$$\text{index} = F_x(x) + F_y(y),$$

where

$$\begin{aligned} F_x(x) &= n^2(x \div n) + (x \bmod n), \\ F_y(y) &= n^2(N_x \div n)(y \div n) + (y \bmod n)n. \end{aligned}$$

We tabulate $F_x$ and $F_y$, and use $x$ and $y$ to find the index into the data array. These tables will consist of $N_x$ and $N_y$ elements, respectively. The total size of the tables will fit in the primary data cache of the processor, even for very large data set sizes.

### 13.4.2 Example: Two-Level Tiling for 3D Arrays

Effective TLB utilization is also becoming a crucial factor in algorithm performance. The same technique can be used to improve TLB hit rates in a 3D array by creating $m \times m \times m$ bricks of $n \times n \times n$ cells. For example, a $40 \times 20 \times 19$ volume could be decomposed into $4 \times 2 \times 2$ macrobricks of $2 \times 2 \times 2$ bricks of $5 \times 5 \times 5$ cells. This corresponds to $m = 2$ and $n = 5$. Because 19 cannot be factored by $mn = 10$, one level of padding is needed. Empirically useful sizes are $m = 5$ for 16 bit datasets and $m = 6$ for float datasets.

The resulting index into the data array can be computed for any $(x, y, z)$ triple with the expression

$$\begin{aligned} \text{index} \quad = \quad & ((x \div n) \div m)n^3m^3((N_z \div n) \div m)((N_y \div n) \div m) \\ & +((y \div n) \div m)n^3m^3((N_z \div n) \div m) \\ & +((z \div n) \div m)n^3m^3 \\ & +((x \div n) \bmod m)n^3m^2 \\ & +((y \div n) \bmod m)n^3m \\ & +((z \div n) \bmod m)n^3 \\ & +(x \bmod (n^2))n^2 \\ & +(y \bmod n)n \\ & +(z \bmod n), \end{aligned}$$

where $N_x$, $N_y$ and $N_z$ are the respective sizes of the dataset.

Note that, as in the simpler 2D one-level case, this expression can be written as

$$\text{index} = F_x(x) + F_y(y) + F_z(z),$$

where

$$\begin{aligned}
F_x(x) &= ((x \div n) \div m)n^3m^3((N_z \div n) \div m)((N_y \div n) \div m) \\
&\quad +((x \div n) \bmod m)n^3m^2 \\
&\quad +(x \bmod n)n^2, \\
F_y(y) &= ((y \div n) \div m)n^3m^3((N_z \div n) \div m) \\
&\quad +((y \div n) \bmod m)n^3m + \\
&\quad +(y \bmod n)n, \\
F_z(z) &= ((z \div n) \div m)n^3m^3 \\
&\quad +((z \div n) \bmod m)n^3 \\
&\quad +(z \bmod n).
\end{aligned}$$

## Frequently Asked Questions

- Does tiling really make that much difference in performance?

On some volume rendering applications, a two-level tiling strategy made as much as a factor-of-ten performance difference. When the array does not fit in main memory, it can effectively prevent thrashing in some applications such as image editing.

- How do I store the lists in a winged-edge structure?

For most applications it is feasible to use arrays and indices for the references. However, if many delete operations are to be performed, then it is wise to use linked lists and pointers.

## Notes

The discussion of the winged-edge data structure is based on the course notes of *Ching-Kuang Shene* (Shene, 2003). There are smaller mesh data structures than winged-edge. The trade-offs in using such structures is discussed in *Directed Edges— A Scalable Representation for Triangle Meshes* (Campagna, Kobbelt, &

Seidel, 1998). The tiled-array discussion is based on *Interactive Ray Tracing for Volume Visualization* (Parker, Martin, et al., 1999).

## Exercises

1. What is the memory difference for a simple tetrahedron stored as four independent triangles and one stored in a winged-edge data structure?

2. Diagram a scene graph for a bicycle.

3. How many look-up tables are needed for a single-level tiling of an $n$-dimensional array?

# 14

# Sampling

Many applications in graphics require "fair" sampling of unusual spaces, such as the space of all possible lines. For example, we might need to generate random edges within a pixel, or random sample points on a pixel that vary in density according to some density function. This chapter provides the machinery for such probability operations. These techniques will also prove useful for numerically evaluating complicated integrals using *Monte Carlo integration*, also covered in this chapter.

## 14.1 Integration

Although the words "integral" and "measure" often seem intimidating, they relate to some of the most intuitive concepts found in mathematics, and they should not be feared. For our very non-rigorous purposes, a *measure* is just a function that maps subsets to $\mathbb{R}^+$ in a manner consistent with our intuitive notions of length, area, and volume. For example, on the 2D real plane $\mathbb{R}^2$, we have the area measure $A$ which assigns a value to a set of points in the plane. Note that $A$ is just a function that takes pieces of the plane and returns area. This means the domain of $A$ is all possible subsets of $\mathbb{R}^2$, which we denote as the *power set* $\mathcal{P}(\mathbb{R}^2)$. Thus, we can characterize $A$ in arrow notation:

$$A : \mathcal{P}(\mathbb{R}^2) \rightarrow \mathbb{R}^+.$$

An example of applying the area measure shows that the area of the square with side length one is one:

$$A([a, a+1] \times [b, b+1]) = 1,$$

where $(a, b)$ is just the lower left-hand corner of the square. Note that a single point such as $(3, 7)$ is a valid subset of $\mathbb{R}^2$ and has zero area: $A((3, 7)) = 0$. The same is true of the set of points $S$ on the $x$-axis, $S = (x, y)$ such that $(x, y) \in \mathbb{R}^2$ and $y = 0$, i.e., $A(S) = 0$. Such sets are called *zero measure sets*.

To be considered a measure, a function has to obey certain area-like properties. For example, we have a function $\mu : \mathcal{P}(\mathbb{S}) \to \mathbb{R}^+$. For $\mu$ to be a measure, the following conditions must be true:

1. The measure of the empty set is zero: $\mu(\emptyset) = 0$,

2. The measure of two distinct sets together is the sum of their measure alone. This rule with possible intersections is

   $$\mu(A \cup B) = \mu(A) + \mu(B) - \mu(A \cap B),$$

   where $\cup$ is the set union operator and $\cap$ is the set intersection operator.

When we actually compute measures, we usually use *integration*. We can think of integration as really just notation:

$$A(S) \equiv \int_{x \in S} dA(\mathbf{x}).$$

You can informally read the right-hand side as "take all points $\mathbf{x}$ in the region $S$, and sum their associated differential areas". The integral is often written other ways including

$$\int_S dA, \qquad \int_{\mathbf{x} \in S} d\mathbf{x}, \qquad \int_{\mathbf{x} \in S} dA_{\mathbf{x}}, \qquad \int_{\mathbf{x}} d\mathbf{x}.$$

All of the above formulas represent "the area of region $S$." We will stick with the first one we used, because it is so verbose it avoids ambiguity. To evaluate such integrals analytically, we usually need to lay down some coordinate system and use our bag of calculus tricks to solve the equations. But have no fear if those skills have faded, as we usually have to numerically approximate integrals, and that requires only a few simple techniques which are covered later in this chapter.

Given a measure on a set $\mathbb{S}$, we can always create a new measure by weighting with a non-negative function $w : \mathbb{S} \to \mathbb{R}^+$. This is best expressed in integral notation. For example, we can start with the example of the simple area measure on $[0,1]^2$:

$$\int_{\mathbf{x}\in[0,1]^2} dA(\mathbf{x}),$$

and we can use a "radially weighted" measure by inserting a weighting function of radius squared:

$$\int_{\mathbf{x}\in[0,1]^2} \|\mathbf{x}\|^2 dA(\mathbf{x}).$$

To evaluate this analytically, we can expand using a Cartesian coordinate system with $dA \equiv dx\, dy$:

$$\int_{\mathbf{x}\in[0,1]^2} \|\mathbf{x}\|^2 dA(\mathbf{x}) = \int_{x=0}^{1}\int_{y=0}^{1} (x^2 + y^2)\ \ dx\, dy.$$

The key thing here is that if you think of the $\|\mathbf{x}\|^2$ term as married to the $dA$ term, and that these together form a new measure, we can call that measure $\nu$. This would allow us to write $\nu(S)$ instead of the whole integral. If this strikes you as just a bunch of notation and bookkeeping, you are right. But it does allow us to write down equations that are either compact or expanded depending on our preference.

### 14.1.1 Measures and Averages

Measures really start paying off when taking averages of a function. You can only take an average with respect to a particular measure, and you would like to select a measure that is "natural" for the application or domain. Once a measure is chosen, the average of a function $f$ over a region $S$ with respect to measure $\mu$ is

$$\text{average}(f) \equiv \frac{\int_{x\in S} f(\mathbf{x})\ d\mu(\mathbf{x})}{\int_{x\in S} d\mu(\mathbf{x})}.$$

For example, the average of the function $f(x,y) = x^2$ over $[0,2]^2$ with respect to the area measure is

$$\text{average}(f) \equiv \frac{\int_{x=0}^{2}\int_{y=0}^{2} x^2\ dx\, dy}{\int_{x=0}^{2}\int_{y=0}^{2}\ dx\, dy} = \frac{4}{3}.$$

This machinery helps solve seemingly hard problems where choosing the measure is the tricky part. Such problems often arise in *integral geometry*, a field that studies measures on geometric entities, such as lines and planes. For example, one might want to know the average length of a line through $[0,1]^2$. That is, by definition,

$$\text{average(length)} = \frac{\int_{\text{lines } L \text{ through } [0,1]^2} \text{length}(L) d\mu(L)}{\int_{\text{lines } L \text{ through } [0,1]^2} d\mu(L)}.$$

All that is left, once we know that, is choosing the appropriate $\mu$ for the application. This is dealt with for lines in the next section.

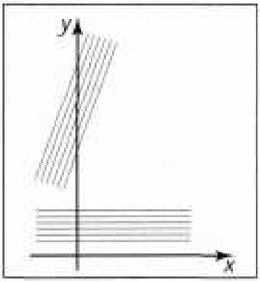


**Figure 14.1.** These two bundles of lines should have the same measure. They have different intersection lengths with the *y*-axis so using *db* would be a poor choice for a differential measure.

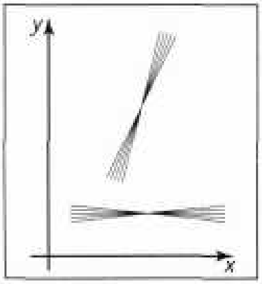


**Figure 14.2.** These two bundles of lines should have the same measure. Since they have different values for change in slope, using *dm* would be a poor choice for a differential measure.

### 14.1.2 Example: Measures on the Lines in the 2D Plane

What measure $\mu$ is "natural"?

If you parameterize the lines as $y = mx + b$, you might think of a given line as a point $(m, b)$ in "slope-intercept" space. An easy measure to use would be $dm\, db$, but this would not be a "good" measure in that not all equal size "bundles" of lines would have the same measure. More precisely, the measure would not be invariant with respect to change of coordinate system. For example, if you took all lines through the square $[0,1]^2$, the measure of lines through it would not be the same as the measure through a unit square rotated forty-five degrees. What we would really like is a "fair" measure that does not change with rotation or translation of a set of lines. This idea is illustrated in Figures 14.1 and 14.2.

To develop a natural measure on the lines, we should first start thinking of them as points in a dual space. This is a simple concept: the line $y = mx + b$ can be specified as the point $(m, b)$ in a slope-intercept space. This concept is illustrated in Figure 14.3. It is more straightforward to develop a measure in $(\phi, b)$ space. In that space $b$ is the $y$-intercept, while $\phi$ is the angle the line makes with the $x$-axis, as shown in Figure 14.4. Here, the differential measure $d\phi\, db$ almost works, but it would not be fair due to the effect shown in Figure 14.1. To account for the larger span $b$ that a constant width bundle of lines makes, we must add a cosine factor:

$$d\mu = \cos\phi \; d\phi \; db.$$

It can be shown that this measure, up to a constant, is the only one that is invariant with respect to rotation and translation.

This measure can be converted into an appropriate measure for other parameterizations of the line. For example, the appropriate measure for $(m, b)$ space is

$$d\mu = \frac{dm\; db}{(1+m^2)^{\frac{3}{2}}}.$$

For the space of lines parameterized in $(u, v)$ space,

$$ux + vy + 1 = 0,$$

the appropriate measure is

$$d\mu = \frac{du\ dv}{(u^2 + v^2)^{\frac{3}{2}}}.$$

For lines parameterized in terms of $(a, b)$, the $x$-intercept and $y$-intercept, the measure is

$$d\mu = \frac{ab\ da\ db}{(a^2 + b^2)^{\frac{3}{2}}}.$$

Note that any of those spaces are equally valid ways to specify lines, and which is best depends upon the circumstances. However, one might wonder whether there exists a coordinate system where the measure of a set of lines is just an area in the dual space. In fact, there is such a coordinate system, and it is delightfully simple; it is the *normal coordinates* which specify a line in terms of the normal distance from the origin to the line, and the angle the normal of the line makes with respect to the $x$-axis (Figure 14.5). The implicit equation for such lines is

$$x \cos\theta + y \sin\theta - p = 0.$$

And, indeed, the measure in that space is

$$d\mu = dp\ d\theta.$$

We shall use these measures to choose fair random lines in a later section.

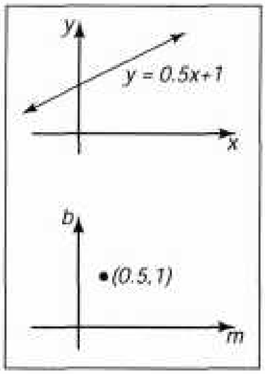


**Figure 14.3.** The set of points on the line $y = mx + b$ in $(x, y)$ space can also be represented by a single point in $(m, b)$ space so the top line and the bottom point represent the same geometric entity: a 2D line.

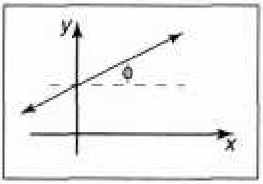


**Figure 14.4.** In angle-intercept space we parameterize the line by angle $\phi \in [-\pi/2, \pi/2)$ rather than slope.

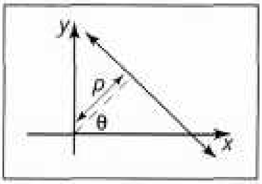


**Figure 14.5.** The normal coordinates of a line use the normal distance to the origin and an angle to specify a line.

### 14.1.3 Example: Measure of Lines in 3D

In 3D there are many ways to parameterize lines. Perhaps, the simplest way is to use their intersection with a particular plane along with some specification of their orientation. For example, we could chart the intersection with the $xy$ plane along with the spherical coordinates of its orientation. Thus, each line would be specified as a $(x, y, \theta, \phi)$ quadruple. This shows that lines in 3D are 4D entities, i.e., they can be described as points in a 4D space.

The differential measure of a line should not vary with $(x, y)$, but bundles of lines with equal cross section should have equal measure. Thus, a fair differential measure is

$$d\mu = dx\,dy\ \sin\theta\,d\theta\,d\phi.$$

Another way to parameterize lines is to chart the intersection with two parallel planes. For example, if the line intersects the plane $z = 0$ at $(x = u, y = v)$ and the plane $z = 1$ at $(x = s, y = t)$, then the line can be described by the quadruple $(u, v, s, t)$. Note, that like the previous parameterization, this one is degenerate for lines parallel to the $xy$ plane. The differential measure is more complicated for this parameterization although it can be approximated as

$$d\mu \approx du\, dv\, a\, ds\, dt,$$

for bundles of lines nearly parallel to the $z$-axis. This is the measure often implicitly used in image-based rendering (Chapter 25).

For sets of lines that intersect a sphere, we can use the parameterization of the two points where the line intersects the sphere. If these are in spherical coordinates, then the point can be described by the quadruple $(\theta_1, \phi_1, \theta_2, \phi_2)$ and the measure is just the differential area associated with each point:

$$d\mu = \sin\theta_1\, d\theta_1\, d\phi_1\, \sin\theta_2\, d\theta_2\, d\phi_2.$$

This implies that picking two uniform random endpoints on the sphere results in a line with uniform density. This observation was used to compute form-factors by Mateu Sbert in his dissertation (Sbert, 1997).

Note that sometimes we want to parameterize directed lines, and sometimes we want the order of the endpoints not to matter. This is a bookkeeping detail that is especially important for rendering applications where the amount of light flowing along a line is different in the two directions along the line.

## 14.2 Continuous Probability

Many graphics algorithms use probability to construct random samples to solve integration and averaging problems. This is the domain of applied continuous probability which has basic connections to measure theory.

### 14.2.1 One-Dimensional Continuous Probability Density Functions

Loosely speaking, a *continuous random variable* $x$ is a scalar or vector quantity that "randomly" takes on some value from the real line $\mathbb{R} = (-\infty, +\infty)$. The behavior of $x$ is entirely described by the distribution of values it takes. This distribution of values can be quantitatively described by the *probability density function* (pdf), $p$, associated with $x$ (the relationship is denoted $x \sim p$). The probability that $x$ assumes a particular value in some interval $[a, b]$ is given by the following integral:

$$\text{Probability}(x \in [a, b]) = \int_a^b p(x)dx. \tag{14.1}$$

Loosely speaking, the probability density function $p$ describes the relative likelihood of a random variable taking a certain value; if $p(x_1) = 6.0$ and $p(x_2) = 3.0$, then a random variable with density $p$ is twice as likely to have a value "near" $x_1$ than it it to have a value near $x_2$. The density $p$ has two characteristics:

$$p(x) \geq 0 \quad \text{(probability is non-negative)}, \tag{14.2}$$

$$\int_{-\infty}^{+\infty} p(x)dx = 1 \quad (\text{Probability}(x \in \mathbb{R}) = 1). \tag{14.3}$$

As an example, the *canonical* random variable $\xi$ takes on values between zero (inclusive) and one (non-inclusive) with uniform probability (here *uniform* simply means each value for $\xi$ is equally likely). This implies that the probability density function $q$ for $\xi$ is

$$q(\xi) = \begin{cases} 1 & \text{if } 0 \leq \xi < 1, \\ 0 & \text{otherwise,} \end{cases}$$

The space over which $\xi$ is defined is simply the interval $[0, 1)$. The probability that $\xi$ takes on a value in a certain interval $[a, b] \in [0, 1)$ is

$$\text{Probability}(a \leq \xi \leq b) = \int_a^b 1\, dx \ = \ b - a.$$

### 14.2.2 One-Dimensional Expected Value

The average value that a real function $f$ of a one-dimensional random variable with underlying pdf $p$ will take on is called its *expected value*, $E(f(x))$ (sometimes written $Ef(x)$):

$$E(f(x)) = \int f(x)p(x)dx.$$

The expected value of a one-dimensional random variable can be calculated by setting $f(x) = x$. The expected value has a surprising and useful property: the expected value of the sum of two random variables is the sum of the expected values of those variables:

$$E(x+y) = E(x) + E(y),$$

for random variables $x$ and $y$. Because functions of random variables are themselves random variables, this linearity of expectation applies to them as well:

$$E(f(x) + g(y)) = E(f(x)) + E(g(y)).$$

An obvious question to ask is whether this property holds if the random variables being summed are correlated (variables that are not correlated are called *independent*). This linearity property in fact does hold *whether or not* the variables are independent! This summation property is vital for most Monte Carlo applications.

### 14.2.3 Multi-Dimensional Random Variables

The discussion of random variables and their expected values extends naturally to multi-dimensional spaces. Most graphics problems will be in such higher-dimensional spaces. For example, many lighting problems are phrased on the surface of the hemisphere. Fortunately, if we define a measure $\mu$ on the space the random variables occupy, everything is very similar to the one-dimensional case. Suppose the space $S$ has associated measure $\mu$; for example $S$ is the surface of a sphere and $\mu$ measures area. We can define a pdf $p : S \mapsto \mathbb{R}$, and if $x$ is a random variable with $x \sim p$, then the probability that $x$ will take on a value in some region $S_i \subset S$ is given by the integral

$$\text{Probability}(x \in S_i) = \int_{S_i} p(x) d\mu.$$

Here *Probability* (*event* ) is the probability that *event* is true, so the integral is the probability that $x$ takes on a value in the region $S_i$.

In graphics, $S$ is often an area ($d\mu = dA = dxdy$) or a set of directions (points on a unit sphere: $d\mu = d\omega = \sin\theta \, d\theta \, d\phi$). As an example, a two-dimensional random variable $\alpha$ is a uniformly distributed random variable on a disk of radius $R$. Here *uniformly* means uniform with respect to area, e.g., the way a bad dart player's hits would be distributed on a dart board. Since it is uniform, we know that $p(\alpha)$ is some constant. From the fact that the area of the disk is $\pi r^2$ and that the total probability is one, we can deduce that

$$p(\alpha) = \frac{1}{\pi R^2}.$$

This means that the probability that $\alpha$ is in a certain subset $S_1$ of the disk is just

$$\text{Probability}(\alpha \in S_1) = \int_{S_1} \frac{1}{\pi R^2} dA.$$

This is all very abstract. To actually use this information, we need the integral in a form we can evaluate. Suppose $S_i$ is the portion of the disk closer to the center than the perimeter. If we convert to polar coordinates, then $\alpha$ is represented as a $(r, \phi)$ pair, and $S_1$ is the region where $r < R/2$. Note, that just because $\alpha$ is uniform, it does not imply that $\phi$ or $r$ are necessarily uniform (in fact, $\phi$ is uniform, and $r$ is not uniform). The differential area $dA$ is just $r\, dr\, d\phi$. Thus,

$$\text{Probability}\left(r < \frac{R}{2}\right) = \int_0^{2\pi} \int_0^{\frac{R}{2}} \frac{1}{\pi R^2} r\, dr\, d\phi = 0.25.$$

The formula for expected value of a real function applies to the multi-dimensional case:

$$E(f(x)) = \int_S f(x)p(x)d\mu,$$

where $x \in S$ and $f : S \mapsto \mathbb{R}$, and $p : S \mapsto \mathbb{R}$. For example, on the unit square $S = [0,1] \times [0,1]$ and $p(x,y) = 4xy$, the expected value of the $x$ coordinate for $(x,y) \sim p$ is

$$\begin{aligned} E(x) &= \int_S f(x,y)p(x,y)dA \\ &= \int_0^1 \int_0^1 4x^2y\, dx\, dy \\ &= \frac{2}{3}. \end{aligned}$$

Note that here $f(x,y) = x$.

### 14.2.4 Variance

The *variance*, $V(x)$, of a one-dimensional random variable is, by definition, the expected value of the square of the difference between $x$ and $E(x)$:

$$V(x) \equiv E([x - E(x)]^2).$$

Some algebraic manipulation gives the non-obvious expression:

$$V(x) = E(x^2) - [E(x)]^2.$$

The expression $E([x - E(x)]^2)$ is more useful for thinking intuitively about variance, while the algebraically equivalent expression $E(x^2) - [E(x)]^2$ is usually convenient for calculations. The variance of a sum of random variables is the sum of the variances *if the variables are independent.* This summation property of variance is one of the reasons it is frequently used in analysis of probabilistic models. The square root of the variance is called the *standard deviation*, $\sigma$, which gives some indication of expected absolute deviation from the expected value.

### 14.2.5 Estimated Means

Many problems involve sums of independent random variables $x_i$, where the variables share a common density $p$. Such variables are said to be *independent identically distributed* (iid) random variables. When the sum is divided by the number of variables, we get an estimate of $E(x)$:

$$E(x) \approx \frac{1}{N} \sum_{i=1}^{N} x_i.$$

As $N$ increases, the variance of this estimate decreases. We want $N$ to be large enough so that we have confidence that the estimate is "close enough." However, there are no sure things in Monte Carlo; we just gain statistical confidence that our estimate is good. To be sure, we would have to have $N = \infty$. This confidence is expressed by the *Law of Large Numbers*:

$$\text{Probability}\left[E(x) = \lim_{N \to \infty} \frac{1}{N} \sum_{i=1}^{N} x_i\right] = 1.$$

## 14.3 Monte Carlo Integration

In this section, the basic Monte Carlo solution methods for definite integrals are outlined. These techniques are then straightforwardly applied to certain integral problems. All of the basic material of this section is also covered in several of the classic Monte Carlo texts. (See the Notes section at the end of this chapter.)

As discussed earlier, given a function $f : S \mapsto \mathbb{R}$ and a random variable $x \sim p$, we can approximate the expected value of $f(x)$ by a sum:

$$E(f(x)) = \int_{x \in S} f(x)p(x)d\mu \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i). \tag{14.4}$$

Because the expected value can be expressed as an integral, the integral is also approximated by the sum. The form of Equation 14.4 is a bit awkward; we would usually like to approximate an integral of a single function $g$ rather than a product $fp$. We can accomplish this by substituting $g = fp$ as the integrand:

$$\int_{x \in S} g(x) d\mu \approx \frac{1}{N} \sum_{i=1}^{N} \frac{g(x_i)}{p(x_i)}. \tag{14.5}$$

For this formula to be valid, $p$ must be positive when $g$ is nonzero.

So to get a good estimate, we want as many samples as possible, and we want the $g/p$ to have a low variance ($g$ and $p$ should have a similar shape). Choosing $p$ intelligently is called *importance sampling*, because if $p$ is large where $g$ is large, there will be more samples in important regions. Equation 14.4 also shows the fundamental problem with Monte Carlo integration: *diminishing return*. Because the variance of the estimate is proportional to $1/N$, the standard deviation is proportional to $1/\sqrt{N}$. Since the error in the estimate behaves similarly to the standard deviation, we will need to quadruple $N$ to halve the error.

Another way to reduce variance is to partition $S$, the domain of the integral, into several smaller domains $S_i$, and evaluate the integral as a sum of integrals over the $S_i$. This is called *stratified sampling*, the technique that jittering employs in pixel sampling (Chapter 10). Normally only one sample is taken in each $S_i$ (with density $p_i$), and in this case the variance of the estimate is:

$$var\left(\sum_{i=1}^{N} \frac{g(x_i)}{p_i(x_i)}\right) = \sum_{i=1}^{N} var\left(\frac{g(x_i)}{p_i(x_i)}\right). \tag{14.6}$$

It can be shown that the variance of stratified sampling is never higher than unstratified if all strata have equal measure:

$$\int_{S_i} p(x) d\mu = \frac{1}{N} \int_{S} p(x) d\mu.$$

The most common example of stratified sampling in graphics is jittering for pixel sampling as discussed in Section 10.11.

As an example of the Monte Carlo solution of an integral $I$, set $g(x)$ equal to $x$ over the interval $(0, 4)$:

$$I = \int_{0}^{4} x \, dx = 8. \tag{14.7}$$

The impact of the shape of the function $p$ on the variance of the $N$ sample estimates is shown in Table 14.1. Note that the variance is reduced when the shape of $p$ is similar to the shape of $g$. The variance drops to zero if $p = g/I$, but

| Method | Sampling function | Variance | Samples needed for standard error of 0.008 |
|---|---|---|---|
| importance | $(6-x)/(16)$ | $56.8N^{-1}$ | 887,500 |
| importance | $1/4$ | $21.3N^{-1}$ | 332,812 |
| importance | $(x+2)/16$ | $6.3N^{-1}$ | 98,437 |
| importance | $x/8$ | $0$ | 1 |
| stratified | 1/4 | $21.3N^{-3}$ | 70 |

**Table 14.1.** Variance for Monte Carlo estimate of $\int_0^4 x\,dx$.

$I$ is not usually known or we would not have to resort to Monte Carlo. One important principle illustrated in Table 14.1 is that stratified sampling is often *far* superior to importance sampling (Mitchell, 1996). Although the variance for this stratification on $I$ is inversely proportional to the cube of the number of samples, there is no general result for the behavior of variance under stratification. There are some functions for which stratification does no good. One example is a white noise function, where the variance is constant for all regions. On the other hand, most functions will benefit from stratified sampling, because the variance in each subcell will usually be smaller than the variance of the entire domain.

### 14.3.1 Quasi–Monte Carlo Integration

A popular method for quadrature is to replace the random points in Monte Carlo integration with *quasi-random* points. Such points are deterministic, but are in some sense uniform. For example, on the unit square $[0,1]^2$, a set of $N$ quasi-random points should have the following property on a region of area $A$ within the square:

$$\text{number of points in the region} \approx AN.$$

For example, a set of regular samples in a lattice has this property.

Quasi-random points can improve performance in many integration applications. Sometimes care must be taken to make sure that they do not introduce aliasing. It is especially nice that, in any application where calls are made to random or stratified points in $[0,1]^d$, one can substitute $d$-dimensional quasi-random points with no other changes.

The key intuition motivating quasi–Monte Carlo integration is that when estimating the average value of an integrand, any set of sample points will do, provided they are "fair."

## 14.4 Choosing Random Points

We often want to generate sets of random or pseudorandom points on the unit square for applications such as distribution ray tracing. There are several methods for doing this, e.g., jittering (see Section 10.11). These methods give us a set of $N$ reasonably equidistributed points on the unit square $[0,1]^2$ : $(u_1, v_1)$ through $(u_N, v_N)$.

Sometimes, our sampling space may not be square (e.g., a circular lens), or may not be uniform (e.g, a filter function centered on a pixel). It would be nice if we could write a mathematical transformation that would take our equidistributed points $(u_i, v_i)$ as input and output a set of points in our desired sampling space with our desired density. For example, to sample a camera lens, the transformation would take $(u_i, v_i)$ and output $(r_i, \phi_i)$ such that the new points are approximately equidistributed on the disk of the lens. While we might be tempted to use the transform

$$\phi_i = 2\pi u_i,$$
$$r_i = v_i R,$$

it has a serious problem. While the points do cover the lens, they do so non-uniformly (Figure 14.6). What we need in this case is a transformation that takes equal-area regions to equal-area regions—one that takes uniform sampling distributions on the square to uniform distributions on the new domain.

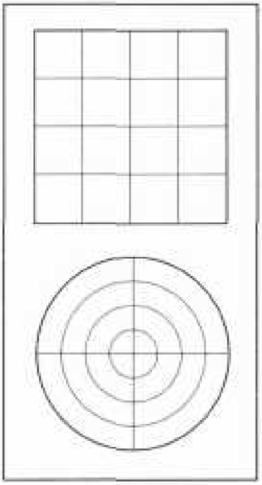


**Figure 14.6.** The transform that takes the horizontal and vertical dimensions uniformly to (r, $\phi$) does not preserve relative area; not all of the resulting areas are the same.

There are several ways to generate such non-uniform points or uniform points on non-rectangular domains, and the following sections review the three most often used: function inversion, rejection, and Metropolis.

### 14.4.1 Function Inversion

If the density $f(x)$ is one-dimensional and defined over the interval $x \in [x_{min}, x_{max}]$, then we can generate random numbers $\alpha_i$ that have density $f$ from a set of uniform random numbers $\xi_i$, where $\xi_i \in [0,1]$. To do this, we need the cumulative probability distribution function $P(x)$:

$$\text{Probability}(\alpha < x) = P(x) = \int_{x_{min}}^{x} f(x')d\mu.$$

To get $\alpha_i$, we simply transform $\xi_i$:

$$\alpha_i = P^{-1}(\xi_i),$$

where $P^{-1}$ is the inverse of $P$. If $P$ is not analytically invertible, then numerical methods will suffice, because an inverse exists for all valid probability distribution functions.

Note that analytically inverting a function is more confusing than it should be due to notation. For example, if we have the function

$$y = x^2,$$

for $x > 0$, then the inverse function is expressed in terms of $y$ as a function of $x$:

$$x = \sqrt{y}.$$

When the function is analytically invertible, it is almost always that simple. However, things are a little more opaque with the standard notation:

$$\begin{aligned} f(x) &= x^2, \\ f^{-1}(x) &= \sqrt{x}. \end{aligned}$$

Here $x$ is just a dummy variable. You may find it easier to use the less standard notation:

$$\begin{aligned} y &= x^2, \\ x &= \sqrt{y}, \end{aligned}$$

while keeping in mind that these are inverse functions of each other.

For example, to choose random points $x_i$ that have density

$$p(x) = \frac{3x^2}{2}$$

on $[-1, 1]$, we see that

$$P(x) = \frac{x^3 + 1}{2},$$

and

$$P^{-1}(x) = \sqrt[3]{2x - 1},$$

so we can "warp" a set of canonical random numbers $(\xi_1, \cdots, \xi_N)$ to the properly distributed numbers

$$(x_1, \cdots, x_N) = (\sqrt[3]{2\xi_1 - 1}, \cdots, \sqrt[3]{2\xi_N - 1}).$$

Of course, this same warping function can be used to transform "uniform" jittered samples into nicely distributed samples with the desired density.

If we have a random variable $\alpha = (\alpha_x, \alpha_y)$ with two-dimensional density $(x, y)$ defined on $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$, then we need the two-dimensional distribution function:

$$\text{Probability}(\alpha_x < x \text{ and } \alpha_y < y) = F(x, y) = \int_{y_{min}}^{y} \int_{x_{min}}^{x} f(x', y') d\mu(x', y').$$

We first choose an $x_i$ using the marginal distribution $F(x, y_{max})$ and then choose $y_i$ according to $F(x_i, y)/F(x_i, y_{max})$. If $f(x, y)$ is separable (expressible as $g(x)h(y)$), then the one-dimensional techniques can be used on each dimension.

Returning to our earlier example, suppose we are sampling uniformly from the disk of radius $R$, so $p(r, \phi) = 1/(\pi R^2)$. The two-dimensional distribution function is

$$\text{Probability}(r < r_0 \;\text{ and }\; \phi < \phi_0) = F(r_0, \phi_0) = \int_0^{\phi_0} \int_0^{r_0} \frac{r dr d\phi}{\pi R^2} = \frac{\phi r^2}{2\pi R^2}.$$

This means that a canonical pair $(\xi_1, \xi_2)$ can be transformed to a uniform random point on the disk:

$$\phi = 2\pi\xi_1,$$
$$r = R\sqrt{\xi_2}.$$

This mapping is shown in Figure 14.7.

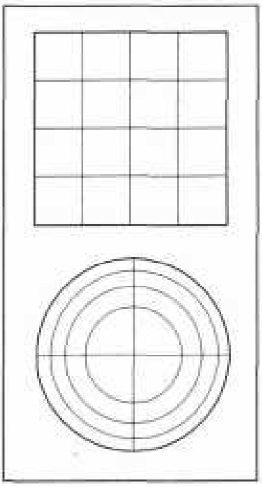


Figure 14.7. A mapping that takes equal area regions in the unit square to equal area regions in the disk.

To choose reflected ray directions for some realistic rendering applications, we choose points on the unit hemisphere according to the density:

$$p(\theta, \phi) = \frac{n+1}{2\pi} \cos^n \theta.$$

Where $n$ is a Phong-like exponent, $\theta$ is the angle from the surface normal and $\theta \in [0, \pi/2]$ (is on the upper hemisphere) and $\phi$ is the azimuthal angle ($\phi \in [0, 2\pi]$). The cumulative distribution function is

$$P(\theta, \phi) = \int_0^{\phi} \int_0^{\theta} p(\theta', \phi') \sin\theta' d\theta' d\phi'. \quad (14.8)$$

The $\sin\theta'$ term arises because, on the sphere, $d\omega = \cos\theta d\theta d\phi$. When the marginal densities are found, $p$ (as expected) is separable, and we find that a $(\xi_1, \xi_2)$ pair of canonical random numbers can be transformed to a direction by

$$\theta = \arccos\left((1 - \xi_1)^{\frac{1}{n+1}}\right),$$
$$\phi = 2\pi\xi_2.$$

Again, a nice thing about this is that a set of jittered points on the unit square can be easily transformed to a set of jittered points on the hemisphere with the desired distribution. Note that if $n$ is set to 1, we have a diffuse distribution, as is often needed.

Often we must map the point on the sphere into an appropriate direction with respect to a *uvw* basis. To do this, we can first convert the angles to a unit vector $\vec{a}$:

$$\mathbf{a} = (\cos\phi \sin\theta,\ \sin\phi \sin\theta,\ \cos\theta)$$

As an efficiency improvement, we can avoid taking trigonometric functions of inverse trigonometric functions (e.g., $\cos(\arccos\theta)$). For example, when $n = 1$ (a diffuse distribution), the vector $\mathbf{a}$ simplifies to

$$\mathbf{a} = \left(\cos(2\pi\xi_1)\sqrt{\xi_2}, \sin(2\pi\xi_1)\sqrt{\xi_2}, \sqrt{1-\xi_2}\right)$$

### 14.4.2 Rejection

A *rejection* method chooses points according to some simple distribution and rejects some of them that are in a more complex distribution. There are several scenarios where rejection is used, and we show some of these by example.

Suppose we want uniform random points within the unit circle. We can first choose uniform random points $(x, y) \in [-1, 1]^2$ and reject those outside the circle. If the function $r()$ returns a canonical random number, then the procedure is:

```
done = false
while (not done) do
   $x = -1 + 2r()$
   $y = -1 + 2r()$
   if ($x^2 + y^2 < 1$) then
      done = true
```

If we want a random number $x \sim p$ and we know that $p : [a, b] \mapsto \mathbb{R}$, and that for all $x$, $p(x) < m$, then we can generate random points in the rectangle $[a, b] \times [0, m]$ and take those where $y < p(x)$:

```
done = false
while (not done) do
   $x = a + r()(b - a)$
   $y = r()m$
   if ($y < p(x)$) then
      done = true
```

This same idea can be applied to take random points on the surface of a sphere. To pick a random unit vector with uniform directional distribution, we first pick a random point in the unit sphere and then treat that point as a direction vector by taking the unit vector in the same direction:

```
done = false
while (not done) do
    $x = -1 + 2r()$
    $y = -1 + 2r()$
    $z = -1 + 2r()$
    if $((l = \sqrt{x^2 + y^2 + z^2}) < 1)$ then
        done = true
$x = x/l$
$y = y/l$
$z = z/l$
```

Although the rejection method is usually simple to code, it is rarely compatible with stratification. For this reason, it tends to converge more slowly and should thus be used mainly for debugging, or in particularly difficult circumstances.

### 14.4.3 Metropolis

The *Metropolis* method uses random *mutations* to produce a set of samples with a desired density. This concept is used extensively in the *Metropolis Light Transport* algorithm referenced in the chapter notes. Suppose we have a random point $x_0$ in a domain $S$. Further, suppose for any point $x$, we have a way to generate random $y \sim p_x$. We use the marginal notation $p_x(y) \equiv p(x \rightarrow y)$ to denote this density function. Now, suppose we let $x_1$ be a random point in $S$ selected with underlying density $p(x_0 \rightarrow x_1)$. We generate $x_2$ with density $p(x_1 \rightarrow x_0)$ and so on. In the limit, where we generate an infinite number of samples, it can be proved that the samples will have some underlying density determined by $p$ regardless of the initial point $x_0$.

Now, suppose we want to choose $p$ such that the underlying density of samples to which we converge is proportional to a function $f(x)$ where $f$ is a non-negative function with domain $S$. Further, suppose we can evaluate $f$, but we have little or no additional knowledge about its properties (such functions are common in graphics). Also, suppose we have the ability to make "transitions" from $x_i$ to $x_{i+1}$ with underlying density function $t(x_i \rightarrow x_{i+1})$. To add flexibility, further suppose we add the potentially non-zero probability that $x_i$ transitions to itself, i.e., $x_{i+1} = x_i$. We phrase this as generating a potential candidate $y \sim t(x_i \rightarrow y)$ and "accepting" this candidate (i.e., $x_{i+1} = y$) with probability $a(x_i \rightarrow y)$ and rejecting it (i.e., $x_{i+1} = x_i$) with probability $1 - a(x_i \rightarrow y)$. Note that the sequence $x_0, x_1, x_2, \ldots$ will be a random set, but there will be some correlation among samples. They will still be suitable for Monte Carlo integration or density estimation, but analyzing the variance of those estimates is much more challenging.

Now, suppose we are given a transition function $t(x \rightarrow y)$ and a function $f(x)$ of which we want to mimic the distribution, can we use $a(y \rightarrow x)$ such that the points are distributed in the shape of $f$? Or more precisely,

$$\{x_0, x_1, x_2, \ldots\} \sim \frac{f}{\int_s f}.$$

It turns out this can be forced by making sure the $x_i$ are *stationary* in some strong sense. If you visualize a huge collection of sample points $x$, you want the "flow" between two points to be the same in each direction. If we assume the density of points near $x$ and $y$ are proportional to $f(x)$ and $f(y)$, respectively, then the flow in the two directions should be the same:

$$\begin{aligned} \text{flow}(x \rightarrow y) &= kf(x)t(x \rightarrow y)a(x \rightarrow y), \\ \text{flow}(y \rightarrow x) &= kf(y)t(y \rightarrow x)a(y \rightarrow x), \end{aligned}$$

where $k$ is some positive constant. Setting these two flows constant gives a constraint on $a$:

$$\frac{a(y \rightarrow x)}{a(x \rightarrow y)} = \frac{f(x)t(x \rightarrow y)}{f(y)t(y \rightarrow x)}.$$

Thus, if either $a(y \rightarrow x)$ or $a(x \rightarrow y)$ is known, so is the other. Making them larger improves the chance of acceptance, so the usual technique is to set the larger of the two to 1.

A difficulty in using the Metropolis sample generation technique is that it is hard to estimate how many points are needed before the set of points is "good." Things are accelerated if the first $n$ points are discarded, although choosing $n$ wisely is non-trivial.

### 14.4.4 Example: Choosing Random Lines in the Square

As an example of the full process of designing a sampling strategy, consider the problem of finding random lines that intersect the unit square $[0, 1]^2$. We want this process to be fair; that is, we would like the lines to be uniformly distributed within the square. Intuitively, we can see that there is some subtlety to this problem; there are "more" lines at an oblique angle than in horizontal or vertical directions. This is because the cross section of the square is not uniform.

Our first goal is to find a function-inversion method, if one exists, and then to fall back on rejection or Metropolis if that fails. This is because we would like to have stratified samples in line space. We try using normal coordinates first, because the problem of choosing random lines in the square is just the problem of finding uniform random points in whatever part of $(r, \theta)$ space corresponds to lines in the square.

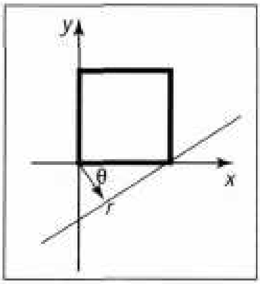


**Figure 14.8.** The largest distance r corresponds to a line hitting the square for $\theta \in [-\pi/2, 0]$. Because the square has sidelength one, $r = \cos\theta$.

Consider the region where $-\pi/2 < \theta < 0$. What values of $r$ correspond to lines that hit the square? For those angles, $r < \cos\theta$ are all the lines that hit the square as shown in Figure 14.8. Similar reasoning in the other four quadrants finds the region in $(r, \theta)$ space that must be sampled, as shown in Figure 14.9. The equation of the boundary of that region $r_{\max}(\theta)$is

$$r_{\max}(\theta) = \begin{cases} 0 & \text{if } \theta \in [-\pi, -\frac{\pi}{2}], \\ \cos\theta & \text{if } \theta \in [-\frac{\pi}{2}, 0], \\ \cos\theta + \sin\theta & \text{if } \theta \in [0, \frac{\pi}{2}], \\ \sin\theta & \text{if } \theta \in [\frac{\pi}{2}, \pi]. \end{cases}$$

Because the region under $r_{\max}(\theta)$ is a simple function bounded below by $r = 0$, we can sample it by first choosing $\theta$ according to the density function:

$$p(\theta) = \frac{r_{\max}(\theta)}{\int_{-\pi}^{\pi} r_{\max}(\theta) d\theta}.$$

The denominator here is 4. Now, we can compute the cumulative probability distribution function:

$$P(\theta) = \begin{cases} 0 & \text{if } \theta \in [-\pi, -\frac{\pi}{2}], \\ (1 + \sin\theta)/4 & \text{if } \theta \in [-\frac{\pi}{2}, 0], \\ (2 + \sin\theta - cos\theta)/4 & \text{if } \theta \in [0, \frac{\pi}{2}], \\ (3 - \cos\theta)/4 & \text{if } \theta \in [\frac{\pi}{2}, \pi]. \end{cases}$$

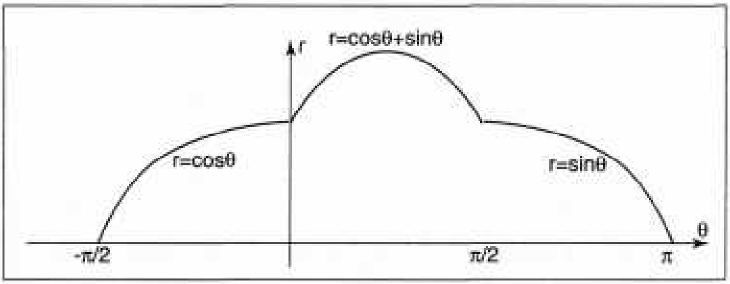


**Figure 14.9.** The maximum radius for lines hitting the unit square $[0,1]^2$ as a function of $\theta$.

We can invert this by manipulating $\xi_1 = P(\theta)$ into the form $\theta = g(\xi_1)$. This yields

$$\theta = \begin{cases} \arcsin(4\xi_1 - 1) & \text{if } \xi_1 < \frac{1}{4}, \\ \left(\arcsin\left((4\xi_1 - 2)^2\right)\right)/2 & \text{if } \xi_1 \in [\frac{1}{4}, \frac{3}{4}], \\ \arccos(3 - 4\xi_1) & \text{if } \xi_1 > \frac{3}{4}. \end{cases}$$

Once we have $\theta$, then $r$ is simply:

$$r = \xi_2 r_{\max}(\theta).$$

As discussed earlier, there are many parameterizations of the line, and each has an associated "fair" measure. We can generate random lines in any of these spaces as well. For example, in slope-intercept space, the region that hits the square is shown in Figure 14.10. By similar reasoning to the normal space, the density function for the slope is

$$p(m) = \frac{1 + |m|}{4}$$

with respect to the differential measure

$$d\mu = \frac{dm}{(1 + m^2)^{\frac{3}{2}}}.$$

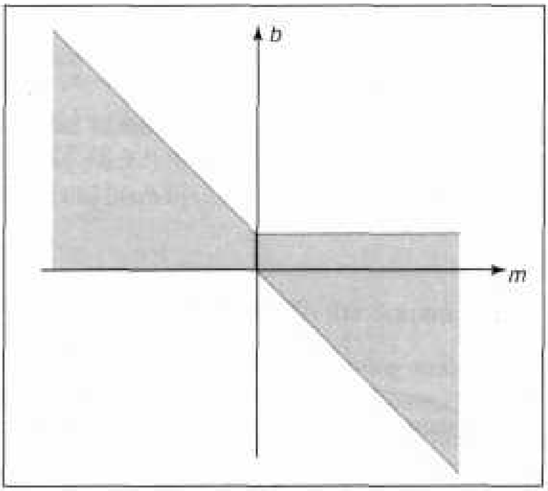


**Figure 14.10.** The region of $(m,b)$ space that contains lines that intersect the unit square $[0,1]^2$.

This gives rise to the cumulative distribution function:

$$P(m) = \begin{cases} \frac{1}{2} - \frac{m-1}{2\sqrt{1+m^2}} & \text{if } m < 0, \\ 1 + \frac{m-1}{2\sqrt{1+m^2}} & \text{if } m \geq 0. \end{cases}$$

These can be inverted by solving two quadratic equations. Given an $m$ generated using $\xi_1$, we then have

$$b = \begin{cases} -m + 2(m+1)\xi_2 & \text{if } \xi < \frac{1}{2}. \\ (1-m)(2\xi_2 - 1) & \text{otherwise.} \end{cases}$$

This is not a better way than using normal coordinates; it is just an alternative way.

## Frequently Asked Questions

- This chapter discussed probability but not statistics. What is the distinction?

Probability is the study of how likely an event is. Statistics infers characteristics of large, but finite, populations of random variables. In that sense, statistics could be viewed as a specific type of applied probability.

- Is Metropolis sampling the same as the Metropolis Light Transport Algorithm?

No. The *Metropolis Light Transport* (Veach & Guibas, 1997) algorithm uses Metropolis sampling as part of its procedure, but it is specifically for rendering, and it has other steps as well.

## Notes

The classic reference for geometric probability is *Geometric Probability* (Solomon, 1978). Another method for picking random edges in a square is given in *Random-Edge Discrepancy of Supersampling Patterns* (Dobkin & Mitchell, 1993). More information on quasi-Monte Carlo methods for graphics can be found in *Efficient Multidimensional Sampling* (Kollig & Keller, 2002). Three classic and very readable books on Monte Carlo methods are *Monte Carlo Methods* (Hammersley & Handscomb, 1964), *Monte Carlo Methods, Basics* (Kalos & Whitlock, 1986), and *The Monte Carlo Method* (Sobel, Stone, & Messer, 1975).

## Exercises

1. What is the average value of the function $xyz$ in the unit cube $(x, y, z) \in [0, 1]^3$?

2. What is the average value of $r$ on the unit-radius disk: $(r, \phi) \in [0, 1] \times [0, 2\pi)$?

3. Show that the uniform mapping of canonical random points $(\xi_1, \xi_2)$ to the barycentric coordinates of any triangle is: $\beta = 1 - \sqrt{1 - \xi_1}$, and $\gamma = (1 - u)\xi_2$.

4. What is the average length of a line inside the unit square? Verify your answer by generating ten million random lines in the unit square and averaging their lengths.

5. What is the average length of a line inside the unit cube? Verify your answer by generating ten million random lines in the unit cube and averaging their lengths.

6. Show from the definition of variance that $V(x) = E(x^2) - [E(x)]^2$.

Michael Gleicher

# 15

# Curves

## 15.1 Curves

Intuitively, think of a *curve* as something you can draw with a pen. The curve is the set of points that the pen traces over an interval of time. While we usually think of a pen writing on paper (e.g., a curve that is in a 2D space), the pen could move in 3D to generate a *space curve*, or you could imagine the pen moving in some other kind of space.

Mathematically, definitions of curve can be seen in at least two ways:

1. The continuous image of some interval in an $n$-dimensional space.

2. A continuous map from a one-dimensional space to an $n$-dimensional space.

Both of these definitions start with the idea of an interval range (the time over which the pen traces the curve). However, there is a significant difference: in the first definition, the curve is the set of points the pen traces (the image), while in the second definition, the curve is the mapping between time and that set of points. For this chapter, we use the first definition.

A curve is an infinitely large set of points. The points in a curve have the property that any point has two neighbors, except for a small number of points that have one neighbor (these are the endpoints). Some curves have no endpoints, either because they are infinite (like a line) or they are *closed* (loop around and connect to themselves).

Because the "pen" of the curve is thin (infinitesimally), it is difficult to create filled regions. While space-filling curves are possible (by having them fold over themselves infinitely many times), we do not consider such mathematical oddities here. Generally, we think of curves as the outlines of things, not the "insides."

The problem that we need to address is how to specify a curve—to give a name or representation to a curve so that we can represent it on a computer. For some curves, the problem of naming them is easy since they have known shapes: line segments, circles, elliptical arcs, etc. A general curve that does not have a "named" shape is sometimes called a *free-form* curve. Because a free-form curve can take on just about any shape, they are much harder to specify.

There are three main ways to specify curves mathematically:

**Implicit** curve representations define the set of points on a curve by giving a procedure that can test to see if a point in on the curve. Usually, an implicit curve representation is defined by an *implicit function* of the form

$$f(x, y) = 0,$$

so that the curve is the set of points for which this equation is true. Note that the implicit function $f$ is a scalar function (it returns a single real number).

**Parametric** curve representations provide a mapping from a *free parameter* to the set of points on the curve. That is, this free parameter provides an index to the points on the curve. The parametric form of a curve is a function that assigns positions to values of the free parameter. Intuitively, if you think of a curve as something you can draw with a pen on a piece of paper, the free parameter is time, ranging over the interval from the time that we began drawing the curve to the time that we finish. The *parametric function* of this curve tells us where the pen is at any instant in time:

$$(x, y) = \mathbf{f}(t).$$

Note that the parametric function is a vector-valued function. This example is a 2D curve, so the output of the function is a 2-vector; in 3D it would be a 3-vector.

**Generative or procedural** curve representations provide procedures that can generate the points on the curve that do not fall into the first two categories. Examples of generative curve descriptions include subdivision schemes and fractals.

Remember that a curve is a set of points. These representations give us ways to specify those sets. Any curve has many possible representations. For this reason, mathematicians typically are careful to distinguish between a curve and its representations. In computer graphics we are often sloppy, since we usually only refer to the representation, not the actual curve itself. So when someone says "an implicit curve," they are either referring to the curve that is represented by some implicit function or to the implicit function that is one of the representations of some curve. Such distinctions are not usually important, unless we need to consider different representations of the same curve. We will consider different curve representations in this chapter, so we will be more careful. When we use a term like "polynomial curve," we will mean the curve that can be represented by the polynomial.

By the definition given at the beginning of the chapter, for something to be a curve it must have a parametric representation. However, many curves have other representations. For example, a circle in 2D with its center at the origin and radius equal to 1 can be written in implicit form as

$$f(x, y) = x^2 + y^2 - 1 = 0,$$

or in parametric form as

$$(x, y) = \mathbf{f}(t) = (\cos t, \sin t), \quad t \in [0, 2\pi).$$

The parametric form need not be the most convenient representation for a given curve. In fact, it is possible to have curves with simple implicit or generative representations for which it is difficult to find a parametric representation.

Different representations of curves have advantages and disadvantages. For example, parametric curves are much easier to draw, because we can sample the free parameter. Generally, parametric forms are the most commonly used in computer graphics since they are easier to work with. Our focus will be on parametric representations of curves.

### 15.1.1 Parameterizations and Re-Parameterizations

A *parametric curve* refers to the curve that is given by a specific parametric function over some particular interval. To be more precise, a parametric curve has a given function that is a mapping from an interval of the parameters. It is often convenient to have the parameter run over the unit interval from 0 to 1. When the free parameter varies over the unit interval, we often denote the parameter as $u$.

If we view the parametric curve to be a line drawn with a pen, we can consider $u = 0$ as the time when the pen is first set down on the paper and the unit of time to be the amount of time it takes to draw the curve ($u = 1$ is the end of the curve).

The curve can be specified by a function that maps time (in these unit coordinates) to positions. Basically, the specification of the curve is a function that can answer the question, "Where is the pen at time $u$?"

If we are given a function $\mathbf{f}(t)$ that specifies a curve over interval $[a, b]$, we can easily define a new function $\mathbf{f}_2(u)$ that specifies the same curve over the unit interval. We can first define

$$g(u) = a + (b - a)u,$$

and then

$$\mathbf{f}_2(u) = \mathbf{f}(g(u)).$$

The two functions, $\mathbf{f}$ and $\mathbf{f}_2$ both represent the same curve; however, they provide different *parameterizations* of the curve. The process of creating a new parameterization for an existing curve is called *re-parameterization*, and the mapping from old parameters to the new ones ($g$, in this example) is called the *re-parameterization function.*

If we have defined a curve by some parameterization, infinitely many others exist (because we can always re-parameterize). Being able to have multiple parameterizations of a curve is useful, because it allows us to create parameterizations that are convenient. However, it can also be problematic, because it makes it difficult to compare two functions to see if they represent the same curve.

The essence of this problem is more general: the existence of the free parameter (or the element of time) adds an invisible, potentially unknown element to our representation of the curves. When we look at the curve after it is drawn, we don't necessarily know the timing. The pen might have moved at a constant speed over the entire time interval, or it might have started slowly and sped up. For example, while $u = 0.5$ is halfway through the parameter space, it may not be half-way along the curve if the motion of the pen starts slowly and speeds up at the end. Consider the following representations of a very simple curve:

$$\begin{aligned}(x, y) &= \mathbf{f}(u) = && (u, u),\\ (x, y) &= \mathbf{f}(u) = && (u^2, u^2),\\ (x, y) &= \mathbf{f}(u) = && (u^5, u^5).\end{aligned}$$

All three functions represent the same curve on the unit interval; however when $u$ is not 0 or 1, $\mathbf{f}(u)$ refers to a different point depending on the representation of the curve.

If we are given a parameterization of a curve, we can use it directly as our specification of the curve, or we can develop a more convenient parameterization. Usually, the *natural parameterization* is created in a way that is convenient (or natural) for specifying the curve, so we don't have to know about how the speed changes along the curve.

If we know that the pen moves at a constant velocity, then the values of the free parameters have more meaning. Halfway through parameter space is halfway along the curve. Rather than measuring time, the parameter can be thought to measure length along the curve. Such parameterizations are called *arc-length* parameterizations because they define curves by functions that map from the distance along the curve (known as the arc length) to positions. We often use the variable $s$ to denote an arc length parameter.

Technically, a parameterization is an arc-length parameterization if the magnitude of its *tangent* (that is, the derivative of the parameterization with respect to the parameter) has constant magnitude. Expressed as an equation,

$$\left|\frac{d\mathbf{f}(s)}{ds}\right|^2 = c.$$

Computing the length along a curve can be tricky. In general, it is defined by the integral of the magnitude of the derivative (intuitively, the magnitude of the derivative is the velocity of the pen as it moves along the curve). So, given a value for the parameter $v$, you can compute $s$ (the arc-length distance along the curve from the point $\mathbf{f}(0)$ to the point $\mathbf{f}(v)$) as

$$s = \int_0^v \left|\frac{d\mathbf{f}(t)}{dt}\right|^2 dt, \tag{15.1}$$

where $\mathbf{f}(t)$ is a function that defines the curve with a natural parameterization.

Using the arc-length parameterization requires being able to solve Equation 15.1 for $t$, given $s$. For many of the kinds of curves we examine, it cannot be done in a closed-form (simple) manner and must be done numerically.

Generally, we use the variable $u$ to denote free parameters that range over the unit interval, $s$ to denote arc-length free parameters, and $t$ to represent parameters that aren't one of the other two.

### 15.1.2 Piecewise Parametric Representations

For some curves, defining a parametric function that represents their shape is easy. For example, lines, circles, and ellipses all have simple functions that define the points they contain in terms of a parameter. For many curves, finding a function that specifies their shape can be hard. The main strategy that we use to create complex curves is divide-and-conquer: we break the curve into a number of simpler smaller pieces, each of which has a simple description.

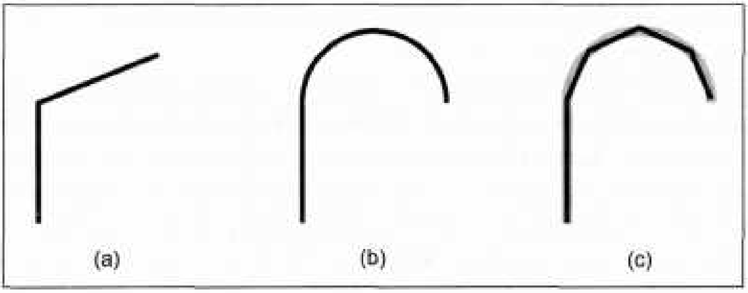


**Figure 15.1.** (a) A curve that can be easily represented as two lines; (b) a curve that can be easily represented as a line and a circular arc; (c) a curve approximating curve (b) with five line segments

For example, consider the curves in Figure 15.1. The first two curves are easily specified in terms of two pieces. In the case of the curve in Figure 15.1(b), we need two different kinds of pieces: a line segment and a circle.

To create a parametric representation of a compound curve (like the curve in Figure 15.1(b)), we need to have our parametric function switch between the functions that represent the pieces. If we define our parametric functions over the range $0 \leq u \leq 1$, then the curve in Figures 15.1(a) or (b) might be defined as

$$\mathbf{f}(u) = \begin{cases} \mathbf{f}_1(2u) & \text{if } u \leq 0.5, \\ \mathbf{f}_2(2u-1) & \text{if } u > 0.5, \end{cases} \tag{15.2}$$

where $\mathbf{f}_1$ is a parameterization of the first piece, $\mathbf{f}_2$ is a parameterization of the second piece, and both of these functions are defined over the unit interval.

We need to be careful in defining the functions $\mathbf{f}_1$ and $\mathbf{f}_2$ to make sure that the pieces of the curve fit together. If $\mathbf{f}_1(1) \neq \mathbf{f}_2(0)$, then our curve pieces will not connect and will not form a single continuous curve.

To represent the curve in Figure 15.1(b), we needed to use two different types of pieces: a line segment and a circular arc. For simplicity's sake, we may prefer to use a single type of piece. If we try to represent the curve in Figure 15.1(b) with only one type of piece (line segments), we cannot exactly recreate the curve (unless we use an infinite number of pieces). While the new curve made of line segments (as in Figure 15.1(c)) may not be exactly the same shape as in Figure 15.1(b), it might be close enough for our use. In such a case, we might prefer the simplicity of using the simpler line segment pieces to having a curve that more accurately represents the shape.

Also, notice that as we use an increasing number of pieces, we can get a better approximation. In the limit (using an infinite number of pieces), we can exactly represent the original shape.

One advantage to using a piecewise representation is that it allows us to make a tradeoff between

1. how well our represented curve approximates the real shape we are trying to represent;
2. how complicated the pieces that we use are;
3. how many pieces we use.

So, if we are trying to represent a complicated shape, we might decide that a crude approximation is acceptable and use a small number of simple pieces. To improve the approximation, we can choose between using more pieces and using more complicated pieces.

In computer graphics practice, we tend to prefer using relatively simple curve pieces (either line segments, arcs, or polynomial segments).

### 15.1.3 Splines

Before computers, when draftsmen wanted to draw a smooth curve, one tool they employed was a stiff piece of metal that they would bend into the desired shape for tracing. Because the metal would bend, not fold, it would have a smooth shape. The stiffness meant that the metal would bend as little as possible to make the desired shape. This stiff piece of metal was called a *spline*.

Mathematicians found that they could represent the curves created by a draftman's spline with piecewise polynomial functions. Initially, they used the term spline to mean a smooth, piecewise polynomial function. More recently, the term spline has been used to describe any piecewise polynomial function. We prefer this latter definition.

For us, a *spline* is a piecewise polynomial function. Such functions are very useful for representing curves.

## 15.2 Curve Properties

To describe a curve, we need to give some facts about its properties. For "named" curves, the properties are usually specific according to the type of curve. For example, to describe a circle, we might provide its radius and the position of its center. For an ellipse, we might also provide the orientation of its major axis and the ratio of the lengths of the axes. For free-form curves however, we need to have a more general set of properties to describe individual curves.

Some properties of curves are attributed to only a single location on the curve, while other properties require knowledge of the whole curve. For an intuition of the difference, imagine that the curve is a train track. If you are standing on the track on a foggy day you can tell that the track is straight or curved and whether or not you are at an end point. These are *local* properties. You cannot tell whether or not the track is a closed curve, or crosses itself, or how long it is. We call this type of property, a *global* property.

The study of local properties of geometric objects (curves and surfaces) is known as *differential geometry.* Technically, to be a differential property, there are some mathematical restrictions about the properties (roughly speaking, in the train-track analogy, you would not be able to have a GPS or a compass). Rather than worry about this distinction, we will use the term *local* property rather than differential property.

Local properties are important tools for describing curves because they do not require knowledge about the whole curve. Local properties include

- continuity,
- position at a specific place on the curve,
- direction at a specific place on the curve,
- curvature (and other derivatives).

Often, we want to specify that a curve includes a particular point. A curve is said to *interpolate* a point if that point is part of the curve. A function $f$ interpolates a value $v$ if there is some value of the parameter $u$ for which $f(t) = v$. We call the place of interpolation, that is the value of $t$, the *site*.

### 15.2.1 Continuity

It will be very important to understand the local properties of a curve where two parametric pieces come together. If a curve is defined using an equation like Equation 15.2, then we need to be careful about how the pieces are defined. If $\mathbf{f}_1(1) \neq \mathbf{f}_2(0)$, then the curve will be "broken"—we would not be able to draw the curve in a continuous stroke of a pen. We call the condition that the curve pieces fit together *continuity* conditions because if they hold, the curve can be drawn as a continuous piece. Because our definition of "curve" at the beginning of the chapter requires a curve to be continuous, technically a "broken curve" is not a curve.

In addition to the positions, we can also check that the derivatives of the pieces match correctly. If $\mathbf{f}_1'(1) \neq \mathbf{f}_2'(0)$, then the combined curve will have an abrupt change in its first derivative at the switching point; the first derivative will not be continuous. In general, we say that a curve is $C^n$ continuous if all of its derivatives up to $n$ match across pieces. We denote the position itself as the zeroth derivative, so that the $C^0$ continuity condition means that the positions of the curve are continuous, and $C^1$ continuity means that positions and first derivatives are continuous. The definition of curve requires the curve to be $C^0$.

An illustration of some continuity conditions is shown in Figure 15.2. A discontinuity in the first derivative (the curve is $C^0$ but not $C^1$) is usually noticeable because it displays a sharp corner. A discontinuity in the second derivative is sometimes visually noticeable. Discontinuities in higher derivatives might matter, depending on the application. For example, if the curve represents a motion, an abrupt change in the second derivative is noticeable, so third derivative continuity is often useful. If the curve is going to have a fluid flowing over it (for example, if it is the shape for an airplane wing or boat hull), a discontinuity in the fourth or fifth derivative might cause turbulence.

The type of continuity we have just introduced ($C^n$) is commonly referred to as *parametric continuity* as it depends on the parameterization of the two curve pieces. If the "speed" of each piece is different, then they will not be continuous. For cases where we care about the shape of the curve, and not its parameterization, we define *geometric continuity* that requires that the derivatives of the curve pieces match when the curves are parameterized equivalently (for example, using an arc-length parameterization). Intuitively, this means that the corresponding

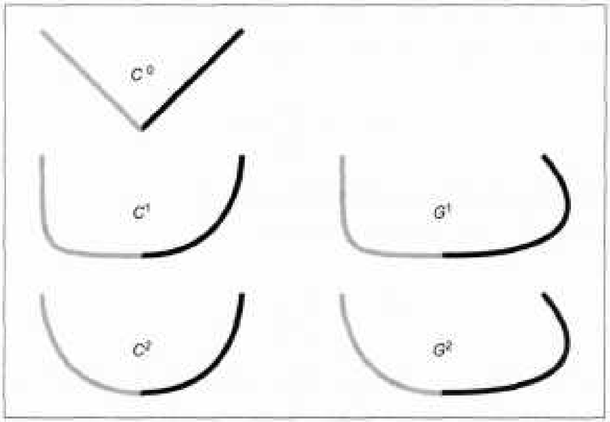


**Figure 15.2.** An illustration of various types of continuity between two curve segments.

derivatives must have the same direction, even if they have different magnitudes. So, if the $C^1$ continuity condition is

$$\mathbf{f}_1'(1) = \mathbf{f}_2'(0),$$

the $G^1$ continuity condition would be

$$\mathbf{f}_1'(1) = k\,\mathbf{f}_2'(0),$$

for some value of scalar $k$. Generally, geometric continuity is less restrictive than parametric continuity. A $C^n$ curve is also $G^n$ except when the parametric derivatives vanish.

## 15.3 Polynomial Pieces

The most widely used representations of curves in computer graphics is done by piecing together basic elements that are defined by polynomials and called polynomial pieces. For example, a line element is given by a linear polynomial. In Section 15.3.1, we give a formal definition and explain how to put pieces of polynomial together.

### 15.3.1 Polynomial Notation

Polynomials are functions of the form

$$f(t) = a_0 + a_1 t + a_2 t^2 + \ldots + a_n t^n. \tag{15.3}$$

The $a_i$ are called the *coefficients*. and $n$ is called the degree of the polynomial if $a_n \neq 0$. We also write Equation 15.3 in the form

$$\mathbf{f}(t) = \sum_{i=0}^{n} \mathbf{a}_i t^i. \tag{15.4}$$

We call this the *canonical* form of the polynomial.

We can generalize the canonical form to

$$\mathbf{f}(t) = \sum_{i=0}^{n} \mathbf{c}_i b_i(t), \tag{15.5}$$

where $b_i(t)$ is a polynomial. We can choose these polynomials in a convenient form for different applications, and we call them *basis functions* or *blending functions* (see Section 15.3.5). In Equation 15.4, the $t^i$ are the $b_i(t)$ of Equation 15.5.

If the set of basis functions is chosen correctly, any polynomial of degree $n + 1$ can be represented by an appropriate choice of $\mathbf{c}$.

The canonical form does not always have convenient coefficients. For practical purposes, throughout this chapter, we will find sets of basis functions such that the coefficients are convenient ways to control the curves represented by the polynomial functions.

To specify a curve embedded in two dimensions, one can either specify two polynomials in $t$: one for how $x$ varies with $t$ and one for how $y$ varies with $t$; or specify a single polynomial where each of the $\mathbf{a}_i$ is a 2D point. An analogous situation exists for any curve in an $n$-dimensional space.

### 15.3.2 A Line Segment

To introduce the concepts of piecewise polynomial curve representations, we will discuss line segments. In practice, line segments are so simple that the mathematical derivations will seem excessive. However, by understanding this simple case, things will be easier when we move on to more complicated polynomials.

Consider a line segment that connects point $\mathbf{p}_0$ to $\mathbf{p}_1$. We could write the parametric function over the unit domain for this line segment as

$$\mathbf{f}(u) = (1 - u)\mathbf{p}_0 + u\mathbf{p}_1. \tag{15.6}$$

By writing this in vector form, we have hidden the dimensionality of the points and the fact that we are dealing with each dimension separately. For example, were we working in 2D, we could have created separate equations:

$$\begin{aligned} f_x(u) = &\quad (1 - u)x_0 + ux_1, \\ f_y(u) = &\quad (1 - u)y_0 + uy_1. \end{aligned}$$

The line that we specify is determined by the two end points, but from now on we will stick to vector notation since it is cleaner. We will call the vector of control parameters, $\mathbf{p}$, the *control points*, and each element of $\mathbf{p}$, a *control point*.

While describing a line segment by the positions of its endpoints is obvious and usually convenient, there are other ways to describe a line segment. For example,

1. the position of the center of the line segment, the orientation, and the length;

2. the position of one endpoint and the position of the second point relative to the first;

3. the position of the middle of the line segment and one endpoint.

It is obvious that given one kind of a description of a line segment, we can switch to another one.

A different way to describe a line segment is using the canonical form of the polynomial (as discussed in Section 15.3.1),

$$\mathbf{f}(u) = \mathbf{a}_0 + u\mathbf{a}_1. \tag{15.7}$$

Any line segment can be represented either by specifying $\mathbf{a}_0$ and $\mathbf{a}_1$ or the endpoints ($\mathbf{p}_0$ and $\mathbf{p}_1$). It is usually more convenient to specify the endpoints, because we can compute the other parameters from the endpoints.

To write the canonical form as a vector expression, we define a vector $\mathbf{u}$ that is a vector of the powers of $u$:

$$\mathbf{u} = \left[1\; u\; u^2\; u^3\; \ldots\; u^n\right],$$

so that Equation 15.4 can be written as

$$\mathbf{f}(u) = \mathbf{u} \cdot \mathbf{a}. \tag{15.8}$$

This vector notation will make transforming between different forms of the curve easier.

Equation 15.8 describes a curve segment by the set of polynomial coefficients for the simple form of the polynomial. We call such a representation the *canonical* form. We will denote the parameters of the canonical form by $\mathbf{a}$.

While it is mathematically simple, the canonical form is not always the most convenient way to specify curves. For example, we might prefer to specify a line segment by the positions of its endpoints. If we want to define $\mathbf{p}_0$ to be the beginning of the segment (where the segment is when $u = 0$) and $\mathbf{p}_1$ to be the end of the line segment (where the line segment is at $u = 1$), we can write

$$\begin{aligned} \mathbf{p}_0 &= \mathbf{f}(0) &= [1\;0] \cdot [\mathbf{a}_0\; \mathbf{a}_1], \\ \mathbf{p}_1 &= \mathbf{f}(1) &= [1\;1] \cdot [\mathbf{a}_0\; \mathbf{a}_1]. \end{aligned} \tag{15.9}$$

We can solve these equations for $\mathbf{a}_0$ and $\mathbf{a}_1$:

$$\begin{aligned} \mathbf{a}_0 &= \mathbf{p}_0, \\ \mathbf{a}_1 &= \mathbf{p}_1 - \mathbf{p}_0. \end{aligned}$$

### Matrix Form for Polynomials

While this first example was easy enough to solve, for more complicated examples it will be easier to write Equation 15.9 in the form

$$\begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{a}_0 \\ \mathbf{a}_1 \end{bmatrix}.$$

Alternatively, we can write

$$\mathbf{p} = \mathbf{C}\,\mathbf{a}, \tag{15.10}$$

where we call $\mathbf{C}$, the *constraint matrix.*[1] If having vectors of points bothers you, you can consider each dimension independently (so that $\mathbf{p}$ is $[x_0\ x_1]$ or $[y_0\ y_1]$) and $\mathbf{a}$ is handled correspondingly).

We can solve Equation 15.10 for $\mathbf{a}$ by finding the inverse of $\mathbf{C}$. This inverse matrix which we will denote by $\mathbf{B}$ is called the *basis* matrix. The basis matrix is very handy since it tells us how to convert between the convenient parameters $\mathbf{p}$ and the canonical form $\mathbf{a}$, and, therefore, gives us an easy way to evaluate the curve

$$\mathbf{f}(u) = \mathbf{u}\,\mathbf{B}\,\mathbf{p}.$$

We can find a basis matrix for whatever form of the curve that we want, providing that there are no non-linearities in the definition of the parameters. Examples of non-linearly defined parameters include the length and angle of the line segment.

Now, suppose we want to parameterize the line segment so that $\mathbf{p}_0$ is the half-way point ($u = 0.5$), and $\mathbf{p}_1$ is the ending point ($u = 1$). To derive the basis matrix for this parameterization, we set

$$\begin{aligned} \mathbf{p}_0 = \quad & \mathbf{f}(0.5) = & 1\,\mathbf{a}_0 + 0.5\,\mathbf{a}_1, \\ \mathbf{p}_1 = \quad & \mathbf{f}(1) = & 1\,\mathbf{a}_0 + 1\,\mathbf{a}_1. \end{aligned}$$

So

$$\mathbf{C} = \begin{bmatrix} 1 & .5 \\ 1 & 1 \end{bmatrix},$$

and therefore

$$\mathbf{B} = \mathbf{C}^{-1} = \begin{bmatrix} 2 & -1 \\ -2 & 2 \end{bmatrix}.$$

### 15.3.3 Beyond Line Segments

Line segments are so simple that finding a basis matrix is trivial. However, it was good practice for curves of higher degree. First, let's consider quadratics (curves of degree two). The advantage of the canonical form (Equation 15.4) is that it works for these more complicated curves, just by letting $n$ be a larger number.

[1] We assume the form of a vector (row or column) is obvious from the context, and we will skip all of the transpose symbols for vectors.

A quadratic (a degree-two polynomial) has three coefficients, $\mathbf{a}_0$, $\mathbf{a}_1$, and $\mathbf{a}_2$. These coefficients are not convenient for describing the shape of the curve. However, we can use the same basis matrix method to devise more convenient parameters. If we know the value of $u$, Equation 15.4 becomes a linear equation in the parameters, and the linear algebra from the last section still works.

Suppose that we wanted to describe our curves by the position of the beginning ($u = 0$), middle[2] ($u = 0.5$), and end ($u = 1$). Entering the appropriate values into Equation 15.4:

$$\begin{array}{llllll} \mathbf{p}_0 & = \mathbf{f}(0) & = \mathbf{a}_0 + 0^1 & \mathbf{a}_1 & + 0^2 & \mathbf{a}_2, \\ \mathbf{p}_1 & = \mathbf{f}(0.5) & = \mathbf{a}_0 + 0.5^1 & \mathbf{a}_1 & + 0.5^2 & \mathbf{a}_2, \\ \mathbf{p}_2 & = \mathbf{f}(1) & = \mathbf{a}_0 + 1^1 & \mathbf{a}_1 & + 1^2 & \mathbf{a}_2. \end{array}$$

So the constraint matrix is

$$\mathbf{C} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & .5 & .25 \\ 1 & 1 & 1 \end{bmatrix},$$

and the basis matrix is

$$\mathbf{B} = \mathbf{C}^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -3 & 4 & -1 \\ 2 & -4 & 2 \end{bmatrix}.$$

There is an additional type of constraint (or parameter) that is sometimes convenient to specify: the derivative of the curve (with respect to its free parameter) at a particular value. Intuitively, the derivatives tell us how the curve is changing, so that the first derivative tells us what direction the curve is going, the second derivative tells us how quickly the curve is changing direction, etc. We will see examples of why it is useful to specify derivatives later.

For the quadratic,

$$\mathbf{f}(u) = \mathbf{a}_0 + \mathbf{a}_1 u + \mathbf{a}_2 u^2,$$

the derivatives are simple:

$$\mathbf{f}'(u) = \frac{d\mathbf{f}}{du} = \mathbf{a}_1 + 2\mathbf{a}_2 u,$$

and

$$\mathbf{f}''(u) = \frac{d^2\mathbf{f}}{du^2} = \frac{d\mathbf{f}'}{du} = 2\mathbf{a}_2.$$

[2] Notice that this is the middle of the parameter space, which might not be the middle of the curve itself.

Or, more generally,

$$\begin{aligned} \mathbf{f}'(u) &= \sum_{i=1}^{n} i u^{i-1} \mathbf{a}_i, \\ \mathbf{f}''(u) &= \sum_{i=2}^{n} i(i-1) u^{i-2} \mathbf{a}_i. \end{aligned}$$

For example, consider a case where we want to specify a quadratic curve segment by the position, first, and second derivative at its middle ($u = 0.5$).

$$\begin{array}{llllllll} \mathbf{p}_0 & = \mathbf{f}(0.5) & = \mathbf{a}_0 + & 0.5^1 & \mathbf{a}_1 + & & 0.5^2 & \mathbf{a}_2, \\ \mathbf{p}_1 & = \mathbf{f}'(0.5) & = & & \mathbf{a}_1 + & 2 & 0.5 & \mathbf{a}_2, \\ \mathbf{p}_2 & = \mathbf{f}''(0.5) & = & & & 2 & & \mathbf{a}_2. \end{array}$$

The constraint matrix is

$$\mathbf{C} = \begin{bmatrix} 1 & .5 & .25 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{bmatrix},$$

and the basis matrix is

$$\mathbf{B} = \mathbf{C}^{-1} = \begin{bmatrix} 1 & -.5 & .125 \\ 0 & 1 & -.5 \\ 0 & 0 & .5 \end{bmatrix}.$$

### 15.3.4 Basis Matrices for Cubics

Cubic polynomials are popular in graphics (See Section 15.5). The derivations for the various forms of cubics are just like the derivations we've seen already in this section. We will work through one more example for practice.

A very useful form of a cubic polynomial is the *Hermite* form, where we specify the position and first derivative at the beginning and end, that is,

$$\begin{array}{lllllllll} \mathbf{p}_0 = & \mathbf{f}(0) & = \mathbf{a}_0 + & 0^1\,\mathbf{a}_1 & + & 0^2\,\mathbf{a}_2 + & & 0^3\,\mathbf{a}_3, \\ \mathbf{p}_1 = & \mathbf{f}'(0) & = & \mathbf{a}_1 & +2 & 0^1\,\mathbf{a}_2 + & 3 & 0^2\,\mathbf{a}_3, \\ \mathbf{p}_2 = & \mathbf{f}(1) & = \mathbf{a}_0 + & 1^1\,\mathbf{a}_1 & + & 1^2\,\mathbf{a}_2 + & & 1^3\,\mathbf{a}_3, \\ \mathbf{p}_3 = & \mathbf{f}'(1) & = & \mathbf{a}_1 & +2 & 1^1\,\mathbf{a}_2 + & 3 & 1^2\,\mathbf{a}_3. \end{array}$$

Thus, the constraint matrix is

$$\mathbf{C} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \end{bmatrix},$$

and the basis matrix is

$$\mathbf{B} = \mathbf{C}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3 & -2 & 3 & -1 \\ 2 & 1 & -2 & 1 \end{bmatrix}.$$

We will discuss Hermite cubic splines in Section 15.5.2.

### 15.3.5 Blending Functions

If we know the basis matrix, $\mathbf{B}$, we can multiply it by the parameter vector, $\mathbf{u}$, to get a vector of functions

$$\mathbf{b}(u) = \mathbf{u}\,\mathbf{B}.$$

Notice that we denote this vector by $\mathbf{b}(u)$ to emphasize the fact that its value depends on the free parameter $u$. We call the elements of $\mathbf{b}(u)$ the *blending functions*, because they specify how to blend the values of the control point vector together:

$$\mathbf{f}(u) = \sum_{i=0}^{n} \mathbf{b}_i(u)\mathbf{p}_i. \tag{15.11}$$

It is important to note that for a chosen value of $u$, Equation 15.11 is a *linear* equation specifying a *linear blend* (or weighted average) of the control points. This is true no matter what degree polynomials are "hidden" inside of the $\mathbf{b}_i$ functions.

Blending functions provide a nice abstraction for describing curves. Any type of curve can be represented as a linear combination of its control points, where those weights are computed as some arbitrary functions of the free parameter.

### 15.3.6 Interpolating Polynomials

In general, a polynomial of degree $n$ can interpolate a set of $n + 1$ values. If we are given a vector $\mathbf{p} = (p_0, \ldots, p_n)$ of points to interpolate and a vector

$\mathbf{t} = (t_0, \ldots, t_n)$ of increasing parameter values, $t_i \neq t_j$, we can use the methods described in the previous sections to determine an $n + 1 \times n + 1$ basis matrix that gives us a function $f(t)$ such that $f(t_i) = p_i$. For any given vector $\mathbf{t}$, we need to set up and solve an $n = 1 \times n + 1$ linear system. This provides us with a set of $n + 1$ basis functions that perform interpolation:

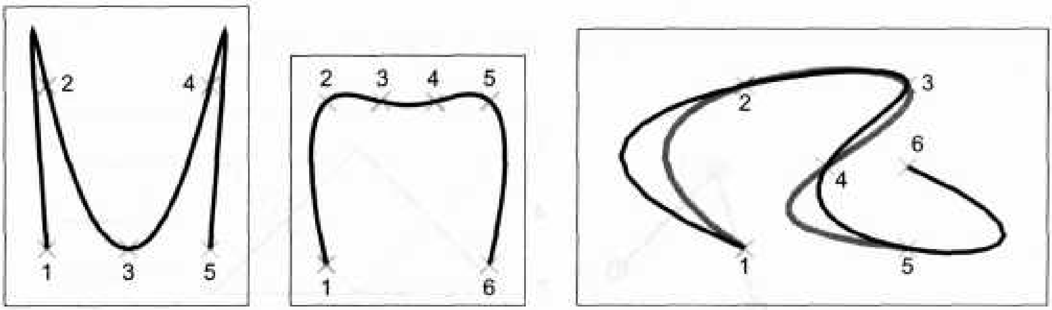
$$\mathbf{f}(t) = \sum_{i=0}^{n} \mathbf{p}_i b_i(t).$$

These interpolating basis functions can be derived in other ways. One particularly elegant way to define them is the *Lagrange form:*

$$b_i = \prod_{j=0, j \neq i}^{n} \frac{x - t_j}{t_i - t_j}. \tag{15.12}$$

There are more computationally efficient ways to express the interpolating basis functions than the Lagrange form (see De Boor (1978) for details).

Interpolating polynomials provide a mechanism for defining curves that interpolate a set of points. Figure 15.3 shows some examples. While it is possible to create a single polynomial to interpolate any number of points, we rarely use high-order polynomials to represent curves in computer graphics. Instead, interpolating splines (piecewise polynomial functions) are preferred. Some reasons for this are considered in Section 15.5.3.



(a) Interpolating polynomial through five points

(b) Interpolating polynomial through six points

(c) Interpolating polynomial through five and six points

**Figure 15.3.** Interpolating polynomials through multiple points. Notice the extra wiggles and over-shooting between points. In (c), when the sixth point is added, it completely changes the shape of the curve due to the non-local nature of interpolating polynomials.

## 15.4 Putting Pieces Together

Now that we've seen how to make individual pieces of polynomial curves, we can consider how to put these pieces together.

### 15.4.1 Knots

The basic idea of a piecewise parametric function is that each piece is only used over some parameter range. For example, if we want to define a function that has two piecewise linear segments that connect three points (as shown in Figure 15.4(a)), we might define

$$\mathbf{f}(u) = \begin{cases} \mathbf{f}_1(2u) & \text{if } 0 \le u < \frac{1}{2}, \\ \mathbf{f}_2(2u-1) & \text{if } \frac{1}{2} \le u < 1, \end{cases} \tag{15.13}$$

where $\mathbf{f}_1$ and $\mathbf{f}_2$ are functions for each of the two line segments. Notice that we have re-scaled the parameter for each of the pieces to facilitate writing their equations as

$$\mathbf{f}_1(u) = (1-u)\mathbf{p}_1 + u\mathbf{p}_2.$$

For each polynomial in our piecewise function, there is a site (or parameter value) where it starts and ends. Sites where a piece function begins or ends are called *knots*. For the example in Equation 15.13, the values of the knots are $0$, $0.5$, and $1$.

We may also write piecewise polynomial functions as the sum of basis functions, each scaled by a coefficient. For example, we can re-write the two line segments of Equation 15.13 as

$$\mathbf{f}(u) = \mathbf{p}_1 b_1(u) + \mathbf{p}_2 b_2(u) + \mathbf{p}_3 b_3(u), \tag{15.14}$$

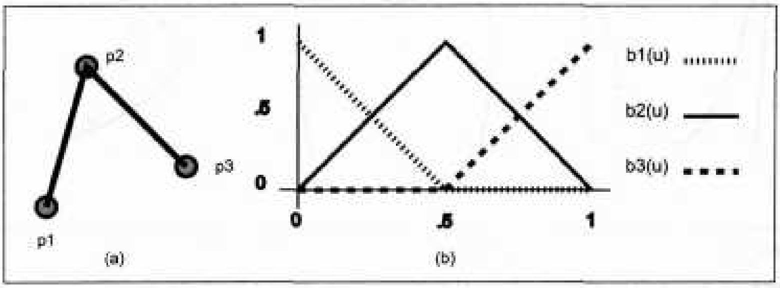


**Figure 15.4.** (a) Two line segments connect three points; (b) the blending functions for each of the points are graphed at right.

where the function $b_1(u)$ is defined as

$$b_1(u) = \begin{cases} 1 - 2u & \text{if } 0 \leq u < \frac{1}{2}, \\ 0 & \text{otherwise,} \end{cases}$$

and $b_2$ and $b_3$ are defined similarly. These functions are plotted in Figure 15.4(b).

The knots of a polynomial function are the combination of the knots of all of the pieces that are used to create it. The *knot vector* is a vector that stores all of the knot values in ascending order.

Notice that in this section we have used two different mechanisms for combining polynomial pieces: using independent polynomial pieces for different ranges of the parameter and blending together piecewise polynomial functions.

### 15.4.2 Using Independent Pieces

In Section 15.3, we defined pieces of polynomials over the unit parameter range. If we want to assemble these pieces, we need to convert from the parameter of the overall function to the value of the parameter for the piece. The simplest way to do this is to define the overall curve over the parameter range $[0, n]$ where $n$ is the number of segments. Depending on the value of the parameter, we can shift it to the required range.

### 15.4.3 Putting Segments Together

If we want to make a single curve from two line segments, we need to make sure that the end of the first line segment is at the same location as the beginning of the next. There are three ways to connect the two segments (in order of simplicity):

1. Represent the line segment as its two endpoints, and then use the same point for both. We call this a *shared-point* scheme.

2. Copy the value of the end of the first segment to the beginning of the second segment every time that the parameters of the first segment change. We call this a *dependency* scheme.

3. Write an explicit equation for the connection, and enforce it through numerical methods as the other parameters are changed.

While the simpler schemes are preferable since they require less work, they also place more restrictions on the way the line segments are parameterized. For example, if we want to use the center of the line segment as a parameter (so that the user can specify it directly), we will use the beginning of each line segment and the center of the line segment as their parameters. This will force us to use the dependency scheme.

Notice that if we use a shared point or dependency scheme, the total number of control points is less than $n * m$, where $n$ is the number of segments and $m$ is the number of control points for each segment; many of the control points of the independent pieces will be computed as functions of other pieces. Notice that if we use either the shared-point scheme for lines (each segment uses its two

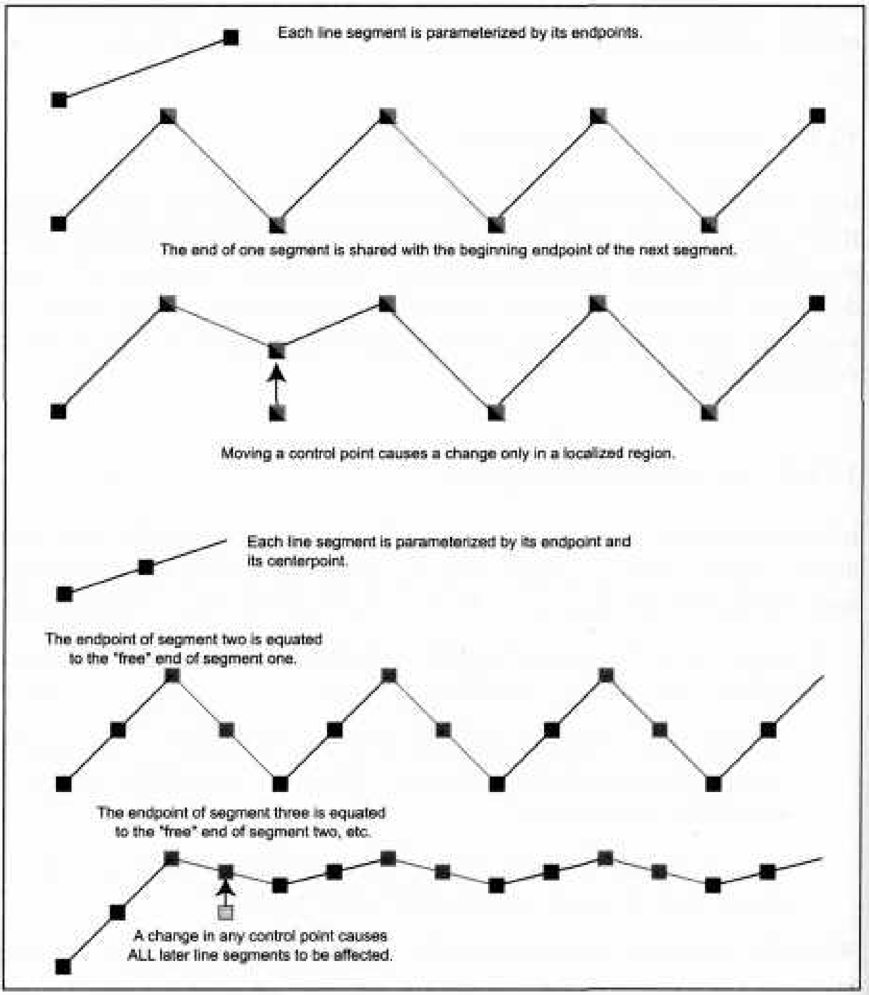

Each line segment is parameterized by its endpoints.
The end of one segment is shared with the beginning endpoint of the next segment.
Moving a control point causes a change only in a localized region.
Each line segment is parameterized by its endpoint and its centerpoint.
The endpoint of segment two is equated to the "free" end of segment one.
The endpoint of segment three is equated to the "free" end of segment two, etc.
A change in any control point causes ALL later line segments to be affected.

**Figure 15.5.** A chain of line segments with local control and one with non-local control.

endpoints as parameters and shares interior points with its neighbors), or if we use the dependency scheme (such as the example one with the first endpoint and midpoint), we end up with $n + 1$ controls for an $n$-segment curve.

Dependency schemes have a more serious problem. A change in one place in the curve can propagate through the entire curve. This is called a lack of *locality.* Locality means that if you move a point on a curve it will only effect a local region. The local region might be big, but it will be finite. If a curve's controls do not have locality, changing a control point may effect points infinitely far away.

To see locality, and the lack thereof, in action, consider two chains of line segments, as shown in Figure 15.5. One chain has its pieces parameterized by their endpoints and uses point-sharing to maintain continuity. The other has its pieces parameterized by an endpoint and midpoint and uses dependency propagation to keep the segments together. The two segment chains can represent the same curves: they are both a set of $n$ connected line segments. However, because of locality issues, the endpoint-shared form is likely to be more convenient for the user. Consider changing the position of the first control point in each chain. For the endpoint-shared version, only the first segment will change, while all of the segments will be affected in the midpoint version, as in Figure 15.5. In fact, for any point moved in the endpoint-shared version, at most two line segments will change. In the midpoint version, all segments after the control point that is moved will change, even if the chain is infinitely long.

In this example, the dependency propagation scheme was the one that did not have local control. This is not always true. There are direct sharing schemes that are not local and propagation schemes that are local.

We emphasize that locality is a convenience of control issue. While it is inconvenient to have the entire curve change every time, the same changes can be made to the curve. It simply requires moving several points in unison.

## 15.5 Cubics

In graphics, when we represent curves using piecewise polynomials we usually use either line segments or cubic polynomials for the pieces. There are a number of reasons why cubics are popular in computer graphics:

- Piecewise cubic polynomials allow for $C^2$ continuity, which is generally sufficient for most visual tasks. The $C^1$ smoothness that quadratics offer is often insufficient. The greater smoothness offered by higher-order polynomials is rarely important.

- Cubic curves provide the minimum-curvature interpolants to a set of points. That is, if you have a set of $n+3$ points and define the "smoothest" curve that passes through them (that is the curve that has the minimum curvature over its length), this curve can be represented as a piecewise cubic with $n$ segments.
- Cubic polynomials have a nice symmetry where position and derivative can be specified at the beginning and end.
- Cubic polynomials have a nice tradeoff between the numerical issues in computation and the smoothness.

Notice that we do not have to use cubics. They just tend to be a good tradeoff between the amount of smoothness and complexity. Different applications may have different tradeoffs. We focus on cubics since they are the most commonly used.

The canonical form of a cubic polynomial is

$$\mathbf{f}(u) = \mathbf{a}_0 + \mathbf{a}_1\, u + \mathbf{a}_2\, u^2 + \mathbf{a}_3\, u^3.$$

As we discussed in Section 15.3, these canonical form coefficients are not a convenient way to describe a cubic segment.

We seek forms of cubic polynomials for which the coefficients are a convenient way to control the resulting curve represented by the cubic. One of the main conveniences will be to provide ways to insure the connectedness of the pieces and the continuity between the segments.

Each cubic polynomial piece requires four coefficients or control points. That means for a piecewise polynomial with $n$ pieces, we may require up to $4n$ control points if no sharing between segments is done or dependencies used. More often, some part of each segment is either shared or depends on an adjacent segment, so the total number of control points is much lower. Also, note that a control point might be a position or a derivative of the curve.

Unfortunately, there is no single "best" representation for a piecewise cubic. It is not possible to have a piecewise polynomial curve representation that has all of the following desirable properties:

1. each piece of the curve is a cubic;
2. the curve interpolates the control points;
3. the curve has local control;
4. the curve has $C^2$ continuity.

We can have any three of these properties, but not all four; there are representations that have any combination of three. In this book, we will discuss cubic B-splines that do not interpolate their control points (but have local control and are $C^2$); Cardinal splines and Catmull-Rom splines that interpolate their control points and offer local control, but are not $C^2$; and natural cubics that interpolate and are $C^2$, but do not have local control.

The continuity properties of cubics refer to the continuity between the segments (at the knot points). The cubic pieces themselves have infinite continuity in their derivatives (the way we have been talking about continuity so far). Note that if you have a lot of control points (or knots), the curve can be wiggly, which might not seem "smooth."

### 15.5.1 Natural Cubics

With a piecewise cubic curve, it is possible to create a $C^2$ curve. To do this, we need to specify the position and first and second derivative at the beginning of each segment (so that we can make sure that it is the same as at the end of the previous segment). Notice, that each curve segment receives three out of its four parameters from the previous curve in the chain. These $C^2$ continuous chains of cubics are sometimes referred to as *natural* cubic splines.

For one segment of the natural cubic, we need to parameterize the cubic by the positions of its endpoints and the first and second derivative at the beginning point. The control points are therefore

$$\begin{array}{lllllllll}
\mathbf{p}_0 = & \mathbf{f}(0) & = \mathbf{a}_0 & + & 0^1\mathbf{a}_1 & + & 0^2\,\mathbf{a}_2 & + & 0^3\,\mathbf{a}_3, \\
\mathbf{p}_1 = & \mathbf{f}'(0) & = & & 1^1\mathbf{a}_1 & +2 & 0^1\,\mathbf{a}_2 & +3 & 0^2\,\mathbf{a}_3, \\
\mathbf{p}_2 = & \mathbf{f}''(0) & = & & & 2 & 1^1\mathbf{a}_2 & +6 & 0^1\,\mathbf{a}_3, \\
\mathbf{p}_3 = & \mathbf{f}(1) & = \mathbf{a}_0 & + & 1^1\,\mathbf{a}_1 & + & 1^2\,\mathbf{a}_2 & + & 1^3\,\mathbf{a}_3.
\end{array}$$

Therefore, the constraint matrix is

$$\mathbf{C} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix},$$

and the basis matrix is

$$\mathbf{B} = \mathbf{C}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & .5 & 0 \\ -1 & -1 & -.5 & 1 \end{bmatrix}.$$

Given a set of $n$ control points, a natural cubic spline has $n-1$ cubic segments. The first segment uses the control points to define its beginning position, ending position, and first and second derivative at the beginning. A dependency scheme copies the position, and first and second derivative of the end of the first segment for use in the second segment.

A disadvantage of natural cubic splines is that they are not local. Any change in any segment may require the entire curve to change (at least the part after the change was made). To make matters worse, natural cubic splines tend to be ill-conditioned: a small change at the beginning of the curve can lead to large changes later. Another issue is that we only have control over the derivatives of the curve at its beginning. Segments after the beginning of the curve determine their derivatives from their beginning point.

### 15.5.2 Hermite Cubics

Hermite cubic polynomials were introduced in Section 15.3.4. A segment of a cubic Hermite spline allows the positions and first derivatives of both of its end points to be specified. A chain of segments can be linked into a $C^1$ spline by using the same values for the position and derivative of the end of one segment and for the beginning of the next.

Given a set of $n$ control points, where every other control point is a derivative value, a cubic Hermite spline contains $(n-2)/2$ cubic segments. The spline interpolates the points, as shown in Figure 15.6, but can guarantee only $C^1$ continuity.

Hermite cubics are convenient because they provide local control over the shape, and provide $C^1$ continuity. However, since the user must specify both positions and derivatives, a special interface for the derivatives must be provided. One possibility is to provide the user with points that represent where the derivative vectors would end if they were "placed" at the position point.

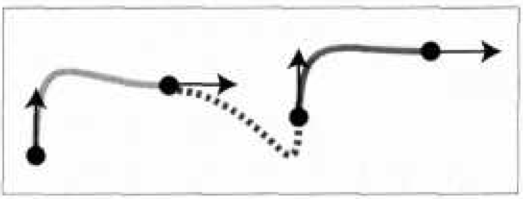


**Figure 15.6.** A Hermite cubic spline made up of three segments.

### 15.5.3 Cardinal Cubics

A *cardinal cubic spline* is a type of $C^1$ interpolating spline made up of cubic polynomial segments. Given a set of $n$ control points, a cardinal cubic spline uses $n-2$ cubic polynomial segments to interpolate all of its points except for the first and last.

Cardinal splines have a parameter called *tension* that controls how "tight" the curve is between the points it interpolates. The tension is a number in the range $[0, 1)$ that controls how the curve bends towards the next control point. For the important special case of $t = 0$, the splines are called *Catmull-Rom* splines.

Each segment of the cardinal spline uses four control points. For segment $i$, the points used are $i$, $i+1$, $i+2$, and $i+3$ as the segments share three points with their neighbors. Each segment begins at its second control point and ends at its third control point. The derivative at the beginning of the curve is determined by the vector between the first and third control points, while the derivative at the end of the curve is given by the vector between the second and forth points, as shown in Figure 15.7.

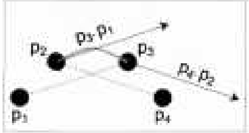


**Figure 15.7.** A segment of a cardinal cubic spline interpolates its second and third control points ($\mathbf{p}_2$ and $\mathbf{p}_3$), and uses its other points to determine the derivatives at the beginning and end.

The tension parameter adjusts how much the derivatives are scaled. Specifically, the derivatives are scaled by $(1-t)/2$. The constraints on the cubic are therefore

$$\begin{aligned} \mathbf{f}(0) &= \mathbf{p}_2, \\ \mathbf{f}(1) &= \mathbf{p}_3, \\ \mathbf{f}'(0) &= \tfrac{1}{2}(1-t)(\mathbf{p}_3 - \mathbf{p}_1), \\ \mathbf{f}'(1) &= \tfrac{1}{2}(1-t)(\mathbf{p}_4 - \mathbf{p}_2). \end{aligned}$$

Solving these equations for the control points (defining $s = (1-t)/2$) gives

$$\begin{array}{rllllllll} \mathbf{p}_0 = & \mathbf{f}(1) - \frac{2}{1-t}\mathbf{f}'(0) & = \mathbf{a}_0 & +(1-\frac{1}{s}) & \mathbf{a}_1 & + & \mathbf{a}_2 & + & \mathbf{a}_3, \\ \mathbf{p}_1 = & \mathbf{f}(0) & = \mathbf{a}_0, & & & & & & \\ \mathbf{p}_2 = & \mathbf{f}(1) & = \mathbf{a}_0 & + & \mathbf{a}_1 & + & \mathbf{a}_2 & + & \mathbf{a}_3, \\ \mathbf{p}_3 = & \mathbf{f}(0) + \frac{1}{s}\mathbf{f}'(1) & = \mathbf{a}_0 & +\frac{1}{s} & \mathbf{a}_1 & +2\frac{1}{s} & \mathbf{a}_2 & +3\frac{1}{s} & \mathbf{a}_3. \end{array}$$

This yields the cardinal matrix

$$\mathbf{B} = \mathbf{C}^{-1} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -s & 0 & s & 0 \\ 2s & s-3 & 3-2s & -s \\ -s & 2-s & s-2 & s \end{bmatrix}.$$

Since the third point of segment $i$ is the second point of segment $i+1$, adjacent segments of the cardinal spline connect. Similarly, the same points are used to specify the first derivative of each segment, providing $C^1$ continuity.

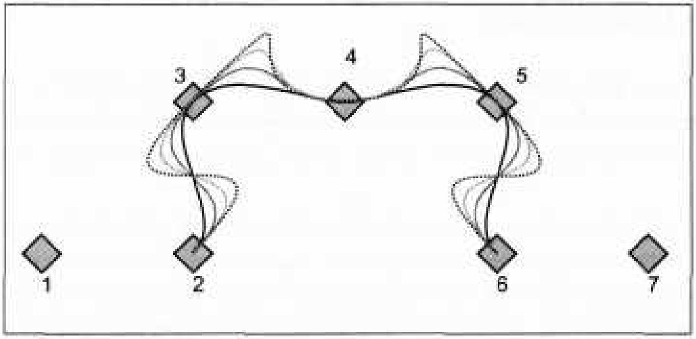


**Figure 15.8.** Cardinal splines through seven control points with varying values of tension parameter $t$.

Cardinal splines are useful, because they provide an easy way to interpolate a set of points with $C^1$ continuity and local control. They are only $C^1$, so they sometimes get "kinks" in them. The tension parameter gives some control over what happens between the interpolated points, as shown in Figure 15.8, where a set of cardinal splines through a set of points is shown. The curves use the same control points, but they use different values for the tension parameters. Note that the first and last control points are not interpolated.

Given a set of $n$ points to interpolate, you might wonder why we might prefer to use a cardinal cubic spline (that is a set of $n - 2$ cubic pieces) rather than a single, order $n$ polynomial as described in Section 15.3.6. Some of the disadvantages of the interpolating polynomial are:

- The interpolating polynomial tends to overshoot the points, as seen in Figure 15.9. This overshooting gets worse as the number of points grows larger. The cardinal splines tend to be well behaved in between the points.
- Control of the interpolating polynomial is not local. Changing a point at the beginning of the spline affects the entire spline. Cardinal splines are local: any place on the spline is affected by its four neighboring points at most.
- Evaluation of the interpolating polynomial is not local. Evaluating a point on the polynomial requires access to all of its points. Evaluating a point on the piecewise cubic requires a fixed small number of computations, no matter how large the total number of points is.

There are a variety of other numerical and technical issues in using interpolating splines as the number of points grows larger. See (De Boor, 2001) for more information.

A cardinal spline has the disadvantage that it does not interpolate the first or last point, which can be easily fixed by adding an extra point at either end of

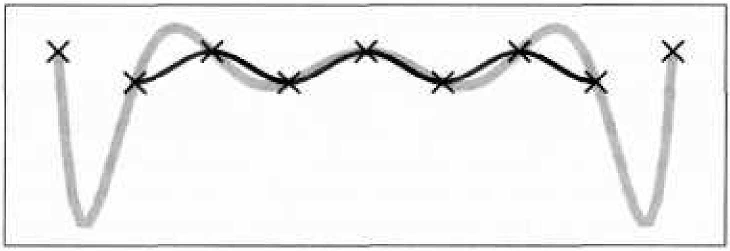


**Figure 15.9.** Splines interpolating nine control points (marked with small crosses). The thick gray line shows an interpolating polynomial. The thin, dark line shows a Catmull-Rom spline. The latter is made of seven cubic segments, which are each shown in alternating gray tones.

the sequence. The cardinal spline also is not as continuous—providing only $C^1$ continuity at the knots.

## 15.6 Approximating Curves

It might seem like the easiest way to control a curve is to specify a set of points for it to interpolate. In practice, however, interpolation schemes often have undesirable properties because they have less continuity and offer no control of what happens between the points. Curve schemes that only approximate the points are often preferred. With an approximating scheme, the control points influence the shape of the curve, but do not specify it exactly. Although we give up the ability to directly specify points for the curve to pass through, we gain better behavior of the curve and local control. Should we need to interpolate a set of points, the positions of the control points can be computed such that the curve passes through these interpolation points.

The two most important types of approximating curves in computer graphics are Bézier curves and B-spline curves.

### 15.6.1 Bézier Curves

Bézier curves are one of the most common representations for free-form curves in computer graphics. The curves are named for Pierre Bézier, one of the people who was instrumental in their development. Bézier curves have an interesting history where they were concurrently developed by several independent groups.

A Bézier curve is a polynomial curve that approximates its control points. The curves can be a polynomial of any degree. A curve of degree $d$ is controlled by

$d + 1$ control points. The curve interpolates its first and last control points, and the shape is directly influenced by the other points.

Often, complex shapes are made by connecting a number of Bézier curves of low degree, and in computer graphics, cubic ($d = 3$) Bézier curves are commonly used for this purpose. Many popular illustration programs, such as Adobe Illustrator, and font representation schemes, such as that used in Postscript, use cubic Bézier curves. Bézier curves are extremely popular in computer graphics because they are easy to control, have a number of useful properties, and there are very efficient algorithms for working with them.

Bézier curves are constructed such that:

- the curve interpolates the first and last control points, with $u = 0$ and 1, respectively;
- the first derivative of the curve at its beginning (end) is determined by the vector between the first and second (next to last and last) control points. The derivatives are given by the vectors between these points scaled by the degree of the curve;
- higher derivatives at the beginning (end) of the curve depend on the points at the beginning (end) of the curve. The $n^{th}$ derivative depends on the first (last) $n + 1$ points.

For example, consider the Bézier curve of degree 3 as in Figure 15.10. The curve has four ($d + 1$) control points. It begins at the first control point ($\mathbf{p}_0$) and ends at the last ($\mathbf{p}_1$). The first derivative at the beginning is proportional to the vector between the first and second control points ($\mathbf{p}_1 - \mathbf{p}_0$). Specifically, $\mathbf{f}'(0) = 3(\mathbf{p}_1 - \mathbf{p}_0)$. Similarly, the first derivative at the end of the curve is given

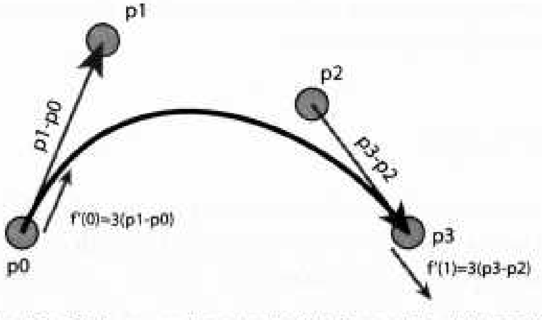


**Figure 15.10.** A cubic Bézier curve is controlled by four points. It interpolates the first and last, and the beginning and final derivatives are three times the vectors between the first two (or last two) points.

by $\mathbf{f}'(1) = 3(\mathbf{p}_3 - \mathbf{p}_2)$. The second derivative at the beginning of the curve can be determined from control points $\mathbf{p}_0$, $\mathbf{p}_1$ and $\mathbf{p}_2$.

Using the facts about Bézier cubics in the preceding paragraph, we can use the methods of Section 15.5 to create a parametric function for them. The definitions of the beginning and end interpolation and derivatives give

$$\begin{aligned}
\mathbf{p}_0 = &\quad \mathbf{f}(0) &&= \mathbf{a}_3 0^3 + \mathbf{a}_2 0^2 + \mathbf{a}_1 0 + \mathbf{a}_0,\\
\mathbf{p}_3 = &\quad \mathbf{f}(1) &&= \mathbf{a}_3 1^3 + \mathbf{a}_2 1^2 + \mathbf{a}_1 1 + \mathbf{a}_0,\\
3(\mathbf{p}_1 - \mathbf{p}_0) = &\quad \mathbf{f}'(0) &&= 3\mathbf{a}_3 0^2 + 2\mathbf{a}_2 0 + \mathbf{a}_1,\\
3(\mathbf{p}_3 - \mathbf{p}_2) = &\quad \mathbf{f}'(1) &&= 3\mathbf{a}_3 1^2 + 2\mathbf{a}_2 1 + \mathbf{a}_1.
\end{aligned}$$

This can be solved for the basis matrix

$$\mathbf{B} = \mathbf{C}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix},$$

and then written as

$$\mathbf{f}(u) = (1 - 3u + 3u^2 - u^3)\mathbf{p}_0 + (3u - 6u^2 + 3u^3)\mathbf{p}_1 + (3u^2 - 3u^3)\mathbf{p}_2 + (u^3)\mathbf{p}_3,$$

or

$$\mathbf{f}(u) = \sum_{i=0}^{d} b_{i,3}\mathbf{p}_i,$$

where the $b_{i,3}$ are the Bézier blending functions of degree 3:

$$\begin{aligned}
b_{0,3} = &\quad (1-u)^3,\\
b_{1,3} = &\quad 3u(1-u)^2,\\
b_{2,3} = &\quad 3u^2(1-u),\\
b_{3,3} = &\quad u^3.
\end{aligned}$$

Fortunately, the blending functions for Bézier curves have a special form that works for all degrees. These functions are known as the *Bernstein basis polynomials* and have the general form

$$b_{k,n}(u) \;=\; C(n,k)\, u^k\, (1-u)^{(n-k)},$$

where $n$ is the order of the Bézier curve, and $k$ is the blending function number between 1 and $n$ (inclusive). $C(n,k)$ are the binomial coefficients:

$$C(n,k) = \frac{n!}{k!\,(n-k)!}.$$

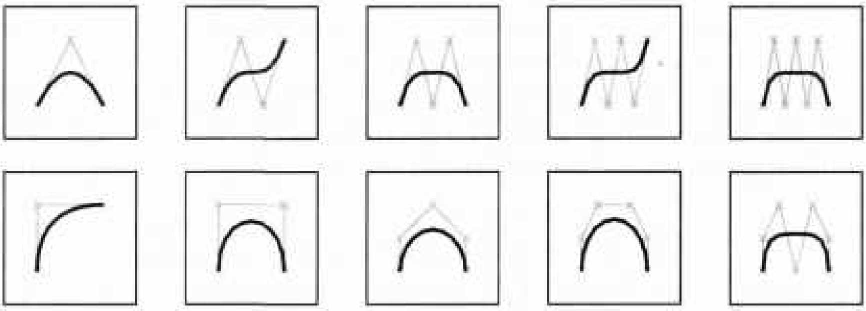


Figure 15.11. Various Bézier segments of degree 2–6. The control points are shown with crosses, and the control polygons (line segments connecting the control points) are also shown.

Given the positions of the control points $\mathbf{p}_k$, the function to evaluate the Bézier curve of order $n$ (with $n + 1$ control points) is

$$\mathbf{p}(u) = \sum_{k=0}^{n} p_k C(n,k)\, u^k\, (1-u)^{(n-k)}.$$

Some Bézier segments are shown in Figure 15.11.

Bézier segments have several useful properties:

- The curve is bounded by the convex hull of the control points.
- Any line intersects the curve no more times than it intersects the set of line segments connecting the control points. This is called the *variation diminishing* property. This property is illustrated in Figure 15.12.
- The curves are symmetric: reversing the order of the control points yields the same curve, with a reversed parameterization.
- The curves are *affine invariant*. This means that translating, scaling, rotating, or skewing the control points is the same as performing those operations on the curve itself.
- There are good simple algorithms for evaluating and subdividing Bézier curves into pieces that are themselves Bézier curves. Because subdivision can be done effectively using the algorithm described later, a divide and conquer approach can be used to create effective algorithms for important tasks such as rendering Bézier curves, approximating them with line segments, and determining the intersection between two curves.

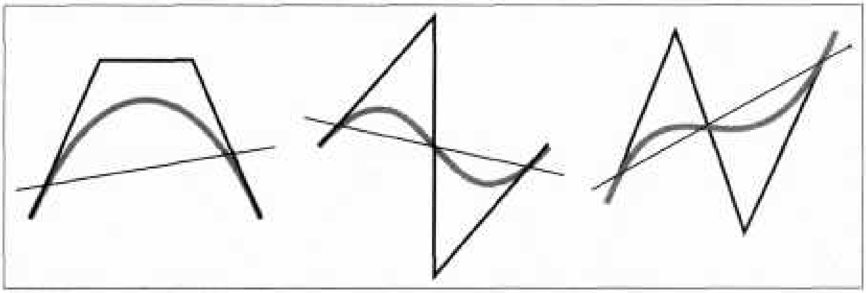


**Figure 15.12.** The *variation diminishing* property of Bézier curves means that the curve does not cross a line more than its control polygon does. Therefore, if the control polygon has no "wiggles," the curve will not have them either. B-splines (Section 15.6.2) also have this property.

When Bézier segments are connected together to make a spline, connectivity between the segments is created by sharing the endpoints. However, continuity of the derivatives must be created by positioning the other control points. This provides the user of a Bézier spline with control over the smoothness. For $G^1$ continuity, the second-to-last point of the first curve and the second point of the second curve must be collinear with the equated endpoints. For $C^1$ continuity, the distances between the points must be equal as well. This is illustrated in Figure 15.13. Higher degrees of continuity can be created by properly positioning more points.

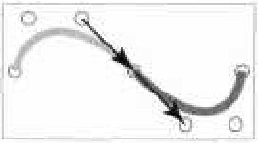


**Figure 15.13.** Two Bézier segments connect to form a $C^1$ spline, because the vector between the last two points of the first segment is equal to the vector between the first two points of the second segment.

### Geometric Intuition for Bezier Curves

Bézier curves can be derived from geometric principles, as well as from the algebraic methods described above. We outline the geometric principles because they provides intuition on how Bézier curves work.

Imagine that we have a set of control points from which we want to create a smooth curve. Simply connecting the points with lines (to form the control polygon) will lead to something that is non-smooth. It will have sharp corners. We could imagine "smoothing" this polygon by cutting off the sharp corners, yielding a new polygon that is smoother, but still not "smooth" in the mathematical sense (since the curve is still a polygon, and therefore only $C^1$. We can repeat this process, each time yielding a smoother polygon, as shown in Figure 15.14. In the limit, that is if we repeated the process infinitely many times, we would obtain a $C^1$ smooth curve.

What we have done with corner cutting is defining a *subdivision* scheme. That is, we have defined curves by a process for breaking a simpler curve into smaller pieces (e.g., subdividing it). The resulting curve is the *limit curve* that is achieved

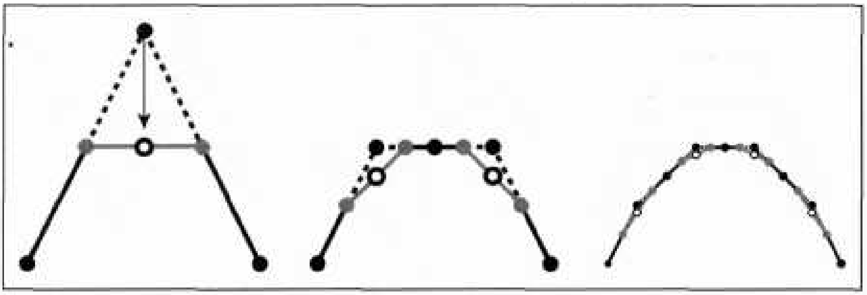


**Figure 15.14.** Subdivision procedure for quadratic Béziers. Each line segment is divided in half and these midpoints are connected (gray points and lines). The interior control point is moved to the midpoint of the new line segment (white circle).

by applying the process infinitely many times. If the subdivision scheme is defined correctly, the result will be a smooth curve, and it will have a parametric form.

Let us consider applying corner cutting to a single corner. Given three points $(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)$, we repeatedly "cut off the corners" as shown in Figure 15.15. At each step, we divide each line segment in half, connect the midpoints, and then move the corner point to the midpoint of the new line segment. Note that in this process, new points are introduced, moved once, and then remain in this position for any remaining iterations. The endpoints never move.

If we compute the "new" position for $\mathbf{p}_2$ as the midpoint of the midpoints, we get the expression

$$\mathbf{p}_2' = \frac{1}{2}(\frac{1}{2}\mathbf{p}_0 + \frac{1}{2}\mathbf{p}_1) + \frac{1}{2}(\frac{1}{2}\mathbf{p}_1 + \frac{1}{2}\mathbf{p}_2).$$

The construction actually works for other proportions of distance along each segment. If we let $u$ be the distance between the beginning and the end of each

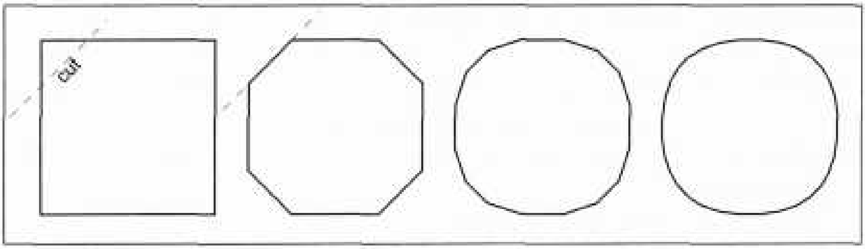


**Figure 15.15.** By repeatedly cutting the corners off a polygon, we approach a smooth curve.

segment where we place the middle point, we can re-write this expression as

$$\mathbf{p}(u) = (1-u)((1-u)\mathbf{p}_0 + u\mathbf{p}_1) + u((1-u)\mathbf{p}_1 + u\mathbf{p}_2).$$

Regrouping terms gives the quadratic Bézier function:

$$\mathbf{B}_2(u) = (1-u)^2\mathbf{p}_0 + 2u(1-u)\mathbf{p}_1 + u^2\mathbf{p}_2.$$

### The De Casteljau Algorithm

One nice feature of Bézier curves is that there is a very simple and general method for computing and subdividing them. The method, called the *de Casteljau algorithm,* uses a sequence of linear interpolations to compute the positions along the Bézier curve of arbitrary order. It is the generalization of the subdivision scheme described in the previous section.

The de Casteljau algorithm begins by connecting every adjacent set of points with lines, and finding the point on these lines that is the $u$ interpolation, giving a set of $n-1$ points. These points are then connected with straight lines, those lines are interpolated (again by $u$), giving a set of $n-2$ points. This process is repeated until there is one point. An illustration of this process is shown in Figure 15.16.

The process of computing a point on a Bézier segment also provides a method for dividing the segment at the point. The intermediate points computed during the de Casteljau algorithm form the new control points of the new, smaller segments, as shown in Figure 15.17.

The existence of a good algorithm for dividing Bézier curves makes divide-and-conquer algorithms possible. For example, when drawing a Bézier curve segment, it is easy to check if the curve is close to being a straight line because it is bounded by its convex hull. If the control points of the curve are all close to being co-linear, the curve can be drawn as a straight line. Otherwise, the curve can be

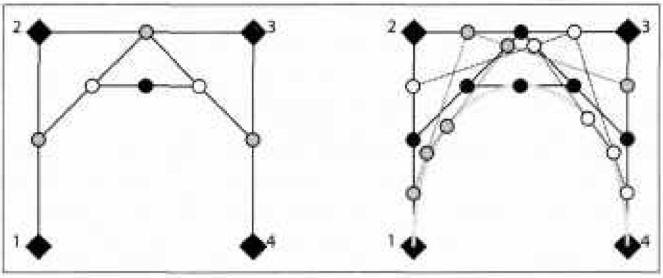


**Figure 15.16.** *An illustration of the de Casteljau algorithm for a cubic Bézier. The left-hand image shows the construction for $u = 0.5$. The right-hand image shows the construction for 0.25, 0.5, and 0.75.*

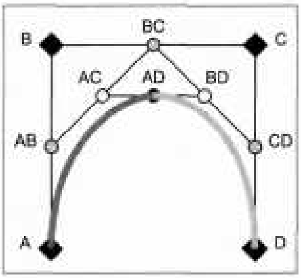


**Figure 15.17.** The de Casteljau algorithm is used to subdivide a cubic Bézier segment. The initial points (black diamonds A, B, C, and D) are linearly interpolated to yield gray circles (AB, BC, CD), which are linearly interpolated to yield white circles (AC, BD), which are linearly interpolated to give the point on the cubic AD. This process also has subdivided the Bézier segment with control points A,B,C,D into two Bézier segments with control points A, AB, AC, AD and AD, BD, CD, D.

divided into smaller pieces, and the process can be repeated. Similar algorithms can be used for determining the intersection between two curves. Because of the existence of such algorithms, other curve representations are often converted to Bézier form for processing.

### 15.6.2 B-Splines

B-splines provide a method for approximating a set of $n$ points with a curve made up of polynomials of degree $d$ that gives $C^{(d-1)}$ continuity. Unlike the Bézier splines of the previous section, B-splines allow curves to be generated for any desired degree of continuity (almost up to the number of points). Because of this, B-splines are a preferred way to specify very smooth curves (high degrees of continuity) in computer graphics. If we want a $C^2$ or higher curve through an arbitrary number of points, B-splines are probably the right method.

We can represent a curve using a linear combination of B-spline basis functions. Since these basis functions are themselves splines, we call them basis splines or B-splines for short. Each B-spline or basis function is made up of a set of $d + 1$ polynomials each of degree $d$. The methods of B-splines provide general procedures for defining these functions.

The term B-spline specifically refers to one of the basis functions, not the function created by the linear combination of a set of B-splines. However, there is inconsistency in how the term is used in computer graphics. Commonly, a "B-spline curve" is used to mean a curve represented by the linear combination of B-splines.

The idea of representing a polynomial as the linear combination of other polynomials has been discussed in Section 15.3.1 and 15.3.5. Representing a spline as a linear combination of other splines was shown in Section 15.4.1. In fact, the example given is a simple case of a B-spline.

The general notation for representing a function as a linear combination of other functions is

$$\mathbf{f}(t) = \sum_{i=1}^{n} \mathbf{p}_i b_i(t), \tag{15.15}$$

where the $\mathbf{p}_i$ are the coefficients and the $b_i$ are the basis functions. If the coefficients are points (e.g. 2 or 3 vectors), we refer to them as control points. The key to making such a method work is to define the $b_i$ appropriately. B-splines provide a very general way to do this.

A set of B-splines can be defined for a number of coefficients $n$ and a parameter value $k$.[3] The value of $k$ is one more than the degree of the polynomials used to make the B-splines ($k = d + 1$.)

B-Splines are important because they provide a very general method for creating functions (that will be useful for representing curves) that have a number of useful properties. A curve with $n$ points made with B-splines with parameter value $k$ is:

- $C^{(k-2)}$ continuous;
- made of polynomials of degree $k - 1$;
- has local control. Any site on the curve only depends on $k$ of the control points;
- is bounded by the convex hull of the points;
- exhibits the variation diminishing property illustrated in Figure 15.12.

A curve created using B-splines does not necessarily interpolate its control points.

We will introduce B-splines by first looking at a specific, simple case to introduce the concepts. We will then generalize the methods and show why they are interesting. Because the method for computing B-splines is very general, we delay introducing it until we have shown what these generalizations are.

[3] The B-spline parameter is actually the *order* of the polynomials used in the B-splines. While this terminology is not uniform in the literature, the use of the B-spline parameter $k$ as a value one greater than the polynomial degree is widely used, although some texts (see the chapter notes) write all of the equations in terms of polynomial degree.

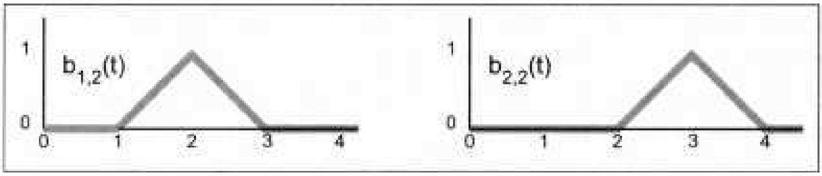


**Figure 15.18.** B-Splines with $d = 2$.

### Uniform Linear B-Splines

Consider a set of basis functions of the following form:

$$b_{i,2}(t) = \begin{cases} t - i & \text{if } i \leq t < i + 1, \\ 2 - t + i & \text{if } i + 1 \leq t \leq i + 2, \\ 0 & \text{otherwise.} \end{cases} \tag{15.16}$$

Each of these functions looks like a little triangular "hat" between $i$ and $i + 2$ with its peak at $i + 1$. Each is a piecewise polynomial, with knots at $i$, $i + 1$, and $i + 2$. Two of them are graphed in Figure 15.18.

Each of these functions $b_{i,2}$ is a first degree (linear) B-spline. Because we will consider B-splines of other parameter values later, we denote these with the 2 in the subscript.

Notice that we have chosen to put the lower edge of the B-spline (its first knot) at $i$. Therefore the first knot of the first B-spline ($i = 1$) is at 1. Iteration over the B-splines or elements of the coefficient vector is from 1 to $n$ (see Equation 15.15). When B-splines are implemented, as well as in many other discussions of them, they often are numbered from 0 to $n - 1$.

We can create a function from a set of $n$ control points using Equation 15.15, with these functions used for the $b_i$ to create an "overall function" that was influenced by the coefficients. If we were to use these ($k = 2$) B-splines to define the overall function, we would define a piecewise polynomial function that linearly interpolates the coefficients $\mathbf{p}_i$ between $t = k$ and $t = n + 1$. Note that while ($k = 2$) B-splines interpolate all of their coefficients, B-splines of higher degree do this under some specific conditions that we will discuss in Section 15.6.3.

Some properties of B-splines can be seen in this simple case. We will write these in the general form using $k$, the parameter, and $n$ for the number of coefficients or control points.

- Each B-spline has $k + 1$ knots.

- Each B-spline is zero before its first knot and after its last knot.

- The overall spline has local control because each coefficient is only multiplied by one B-spline, and this B-spline is non-zero only between $k + 1$ knots.

- The overall spline has $n + k$ knots.

- Each B-spline is $C^{(k-2)}$ continuous, therefore the overall spline is $C^{(k-2)}$ continuous.

- The set of B-splines sums to 1 for all parameter values between knots $k$ and $n + 1$. This range is where there are $k$ B-splines that are non-zero. Summing to 1 is important because it means that the B-splines are shift invariant: translating the control points will translate the entire curve.

- Between each of its knots, the B-Spline is a single polynomial of degree $d = k - 1$. Therefore, the overall curve (that sums these together) can also be expressed as a single, degree $d$ polynomial between any adjacent knots.

In this example, we have chosen the knots to be uniformly spaced. We will consider B-splines with non-uniform spacing later. When the knot spacing is uniform, each of the B-splines are identical except for being shifted. B-splines with uniform knot spacing are sometimes called *uniform B-splines* or *periodic B-splines*.

### Uniform Quadratic B-Splines

The properties of B-Splines listed in the previous section were intentionally written for arbitrary $n$ and $k$. A general procedure for constructing the B-splines will be provided later, but first, lets consider another specific case with $k = 3$.

The B-spline $b_{2,3}$ is shown in Figure 15.19. It is made of quadratic pieces (degree 2), and has 3 of them. It is $C^1$ continuous and is non-zero only within the 4 knots that it spans. Notice that a quadratic B-spline is made of 3 pieces, one between knot 1 and 2, one between knot 2 and 3, and one between knot 3

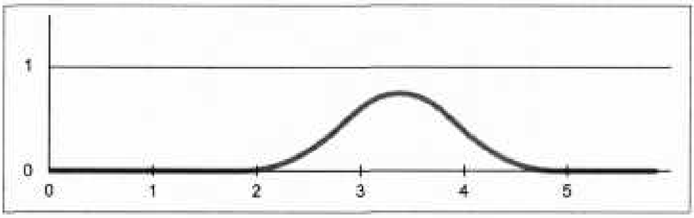


**Figure 15.19.** The B-spline $b_{2,3}$ with uniform knot spacing.

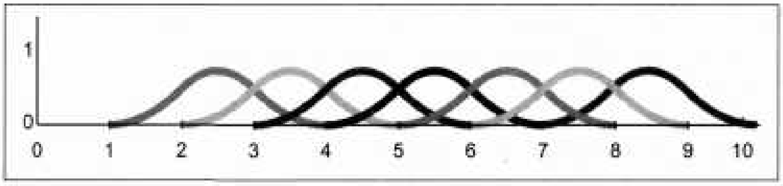


**Figure 15.20.** The set of seven B-splines with $k = 3$ and uniform knot spacing [1, 2, 3, 4, 5, 6, 7, 8, 10].

and 4. In Section 15.6.3 we will see a general procedure for building these functions. For now, we simply examine these functions:

$$b_{i,3}(t) = \begin{cases} \frac{1}{2}u^2 & \text{if } i \le t < i+1 \quad u = t - i, \\ -u^2 + u + \frac{1}{2} & \text{if } i+1 \le t < i+2 \quad u = t - (i+1), \\ \frac{1}{2}(1-u)^2 & \text{if } i+2 \le t < i+3 \quad u = t - (i+2), \\ 0 & \text{otherwise.} \end{cases} \tag{15.17}$$

In order to make the expressions simpler, we wrote the function for each part as if it applied over the range 0 to 1.

If we evaluate the overall function made from summing together the B-splines, at any time only $k$ (3 in this case) of them are non-zero. One of them will be in the first part of Equation 15.17, one will be in the second part, and one will be in the third part. Therefore, we can think of any piece of the overall function as being made up of a degree $d = k - 1$ polynomial that depends on $k$ coefficients. For the $k = 3$ case, we can write

$$\mathbf{f}(u) = \frac{1}{2}(1-u)^2\mathbf{p}_i + (-u^2 + u + \frac{1}{2})\mathbf{p}_{i+1} + \frac{1}{2}u^2\mathbf{p}_{i+2}$$

where $u = t - i$. This defines the piece of the overall function when $i \le t < i+1$.

If we have a set of $n$ points, we can use the B-splines to create a curve. If we have seven points, we will need a set of seven B-splines. A set of seven B-splines

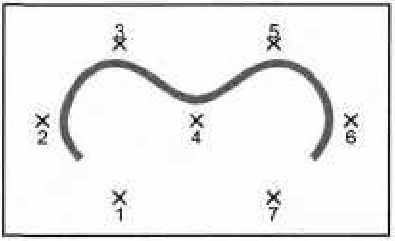


**Figure 15.21.** Curve made from seven quadratic ($k$=3) B-splines, using seven control points.

for $k = 3$ is shown in Figure 15.20. Notice that there are $n + k$ (10) knots, that the sum of the B-splines is 1 over the range $k$ to $n + 1$ (knots 3 through 8). A curve specified using these B-splines and a set of points is shown in Figure 15.21.

## Uniform Cubic B-Splines

Because cubic polynomials are so popular in computer graphics, the special case of B-splines with $k = 4$ is sufficiently important that we consider it before discussing the general case. A B-spline of third degree is defined by 4 cubic polynomial pieces. The general process by which these pieces are determined is described later, but the result is

$$b_{i,4}(t) = \begin{cases} \frac{1}{6}u^3 & \text{if } i \le t < i+1 \quad u = t - i, \\ \frac{1}{6}(-3u^3 + 3u^2 + 3u + 1) & \text{if } i+1 \le t < i+2 \quad u = t - (i+1), \\ \frac{1}{6}(3u^3 - 6u^2 + 4) & \text{if } i+2 \le t < i+3 \quad u = t - (i+2), \\ \frac{1}{6}(-u^3 + 3u^2 - 3u + 1) & \text{if } i+3 \le t < i+4 \quad u = t - (i+3), \\ 0 & \text{otherwise.} \end{cases} \tag{15.18}$$

This degree 3 B-spline is graphed for $i = 1$ in Figure 15.22.

We can write the function for the overall curve between knots $i - 3$ and $i + 4$ as a function of the parameter $u$ between 0 and 1 and the four control points that influence it:

$$\begin{aligned} \mathbf{f}(u) = {} & \frac{1}{6}(-u^3 + 3u^2 - 3u + 1)\mathbf{p}_i + \frac{1}{6}(3u^3 - 6u^2 + 4)\mathbf{p}_{i+1} \\ & + \frac{1}{6}(-3u^3 + 3u^2 + 3u + 1)\mathbf{p}_{i+2} + \frac{1}{6}u^3\mathbf{p}_{i+3}. \end{aligned}$$

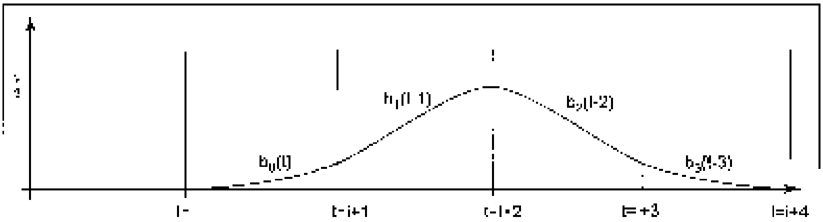


Figure 15.22. The cubic ($k = 4$) B-spline with uniform knots.

This can be re-written using the matrix notation of the previous sections, giving a basis matrix for cubic B-splines of

$$\mathbf{M_b} = \frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}.$$

Unlike the matrices that were derived from constraints in Section 15.5, this matrix is created from the polynomials that are determined by the general B-spline procedure defined in the next section.

### 15.6.3 Non-Uniform B-Splines

One nice feature of B-splines is that they can be defined for any $k > 1$. So if we need a smoother curve, we can simply increase the value of $k$. This is illustrated in Figure 15.1.

So far, we have said that B-splines generalize to any $k > 1$ and any $n \geq d$. There is one last generalization to introduce before we show how to actually compute these B-splines. B-splines are defined for any non-decreasing knot vector.

For a given $n$ and $k$, the set of B-splines (and the function created by their linear combination) has $n + k$ knots. We can write the value of these knots as a vector, that we will denote as $\mathbf{t}$. For the uniform B-splines, the knot vector is $[1, 2, 3, \ldots, n + k]$. However, B-splines can be generated for any knot vector of length $n + k$, providing the values are non-decreasing (e.g., $t_{i+1} \geq t_i$).

There are two main reasons why non-uniform knot spacing is useful: it gives us control over what parameter range of the overall function each coefficient af-

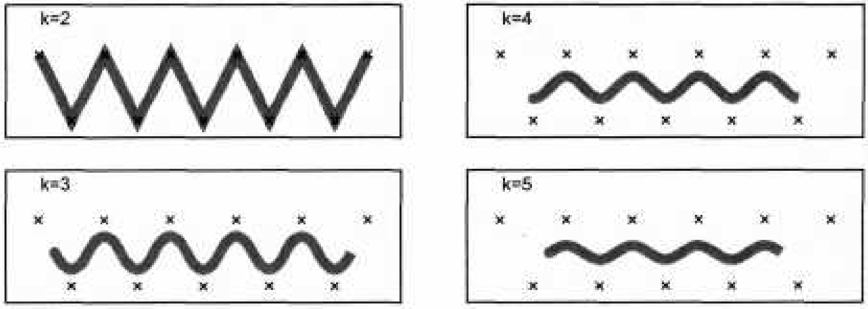


**Figure 15.1.** B-spline curves using the same uniform set of knots and the same control points, for various values of *k*. Note that as *k* increases, the valid parameter range for the curve shrinks.

fects, and it allows us to repeat knots (e.g., create knots with no spacing in between) in order to create functions with different properties around these points. The latter will be considered later in this section.

The ability to specify knot values for B-splines is similar to being able to specify the interpolation sites for interpolating spline curves. It allows us to associate curve features with parameter values. By specifying a non-uniform knot vector, we specify what parameter range each coefficient of a B-spline curve affects. Remember that B-spline $i$ is non-zero only between knot $i$ and knot $i+k$. Therefore, the coefficient associated with it only affects the curve between these parameter values.

One place where control over knot values is particularly useful is in inserting or deleting knots near the beginning of a sequence. To illustrate this, consider a curve defined using linear B-splines ($k = 2$) as discussed in Section 15.6.2. For $n = 4$, the uniform knot vector is $[1, 2, 3, 4, 5, 6]$. This curve is controlled by a set of four points and spans the parameter range $t = 2$ to $t = 5$. The "end" of the curve ($t = 5$) interpolates the last control point. If we insert a new point in the middle of the point set, we would need a longer knot vector. The locality properties of the B-splines prevent this insertion from affecting the values of the curve at the ends. The longer curve would still interpolate its last control point at its end. However, if we chose to keep the uniform knot spacing, the new knot vector would be $[1, 2, 3, 4, 5, 6, 7]$. The end of the curve would be at $t = 6$, and the parameter value at which the last control point is interpolated will be a different parameter value than before the insertion. With non-uniform knot spacing, we can use the knot vector $[1, 2, 3, 3.5, 4, 5, 6]$ so that the ends of the curve are unaffected by the change. The abilities to have non-uniform knot spacing makes the locality property of B-splines an algebraic property, as well as a geometric one.

We now introduce the general method for defining B-splines. Given values for the number of coefficients $n$, the B-spline parameter $k$, and the knot vector $\mathbf{t}$ (which has length $n + k$), the following recursive equations define the B-splines:

$$b_{i,1,\mathbf{t}}(t) = \begin{cases} 1 & \text{if } \mathbf{t}_i \leq t < \mathbf{t}_{i+1}, \\ 0 & \text{otherwise.} \end{cases} \tag{15.19}$$

$$b_{i,k,\mathbf{t}}(t) = \frac{t-\mathbf{t}_i}{\mathbf{t}_{i+k-1}-\mathbf{t}_i} b_{i,k-1}(t) + \frac{\mathbf{t}_{i+k}-t}{\mathbf{t}_{i+k}-\mathbf{t}_{i+1}} b_{i+1,k-1}(t). \tag{15.20}$$

This equation is know as the *Cox-de Boor recurrence*. It may be used to compute specific values for specific B-splines. However, it is more often applied algebraically to derive equations such as Equation 15.17 or 15.18.

As an example, consider how we would have derived Equation 15.17. Using a uniform knot vector $[1, 2, 3, \ldots]$, $t_i = i$, and the value $k = 3$ in Equation 15.20

yields

$$\begin{aligned} b_{i,3}(t) &= \frac{t-i}{(i+2)-i}b_{i,2} + \frac{(i+3)-t}{(i+3)-(i+1)}b_{i+1,2} \qquad (15.21) \\ &= \frac{1}{2}(t-i)b_{i,2} + \frac{1}{2}(i+3-t)b_{i+1,2}. \end{aligned}$$

Continuing the recurrence, we must evaluate the recursive expressions:

$$\begin{aligned} b_{i,2}(t) &= \frac{t-i}{(i+2-1)-i}b_{i,1} + \frac{(i+2)-t}{(i+2)-(i+1)}b_{i+1,1} \\ &= (t-i)b_{i,1} + (i+2-t)b_{i+1,1} \\ b_{i+1,2}(t) &= \frac{t-(i+1)}{((i+1)+2-1)-(i+1)}b_{i+1,1} \\ &\quad + \frac{((i+1)+2)-t}{((i+1)+2)-((i+1)+1)}b_{(i+1)+1,1} \\ &= (t-i+1)b_{i+1,1} + (i+3-t)b_{i+2,1}. \end{aligned}$$

Inserting these results into Equation 15.22 gives:

$$\begin{aligned} b_{i,3}(t) = \frac{1}{2}(t-i)((t-i)b_{i,1} + (i+2-t)b_{i+1,1}) \\ + \frac{1}{2}(i+3-t)(t-i+1)b_{i+1,1} + (i+3-t)b_{i+2,1}. \end{aligned}$$

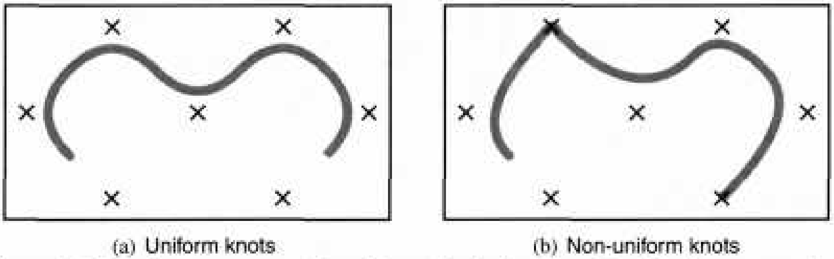
To see that this expression is equivalent to Equation 15.17, we note that each of the $(k = 1)$ B-splines is like a switch, turning on only for a particular parameter range. For instance, $b_{i,1}$ is only non-zero between $i$ and $i+1$. So, if $i \leq t < i+1$, only the first of the $(k = 1)$ B-splines in the expression is non-zero, so

$$b_{i,3}(t) = \frac{1}{2}(t-i)^2 \text{ if } i \leq t < i+1.$$

Similar manipulations give the other parts of Equation 15.17.

### Repeated Knots and B-Spline Interpolation

While B-splines have many nice properties, functions defined using them generally do not interpolate the coefficients. This can be inconvenient if we are using them to define a curve that we want to interpolate a specific point. We give a brief overview of how to interpolate a specific point using B-splines here. A more complete discussion can be found in the books listed in the chapter notes.


(a) Uniform knots

(b) Non-uniform knots

**Figure 15.23.** A curve parameterized by quadratic B-splines ($k$ = 3) with seven control points. On the left, uniform knots vector [1,2,3,4,5,6,7,8,9,10] is used. On the right, the non-uniform knot spacing [1,2,3,4,4,6,7,8,8,10] is used. The duplication of the 4th and 8th knot means that all interior knots of the 3rd and 7th B-spline are equal, so the curve interpolates the control point associated with those points.

One way to cause B-splines to interpolate their coefficients is to repeat knots. If all of the interior knots for a particular B-spline have the same value, then the overall function will interpolate this B-spline's coefficient. An example of this is shown in Figure 15.23.

Interpolation by repeated knots comes at a high cost: it removes the smoothness of the B-spline and the resulting overall function and represented curve. However, at the beginning and end of the spline, where continuity is not an issue, knot repetition is useful for creating *endpoint interpolating B-splines.* While the first (or last) knot's value is not important for interpolation, for simplicity, we make the first (or last) $k$ knots have the same value to achieve interpolation.

Endpoint interpolating quadratic B-splines are shown in Figure 15.24. The first two and last two B-splines are different than the uniform ones. Their expressions can be derived through the use of the Cox-de Boor recurrence:

$$b_{1,3,[0,0,0,1,2,\ldots]}(t) = \begin{cases} (1-t)^2 & \text{if } 0 \le t < 1, \\ 0 & \text{otherwise.} \end{cases}$$

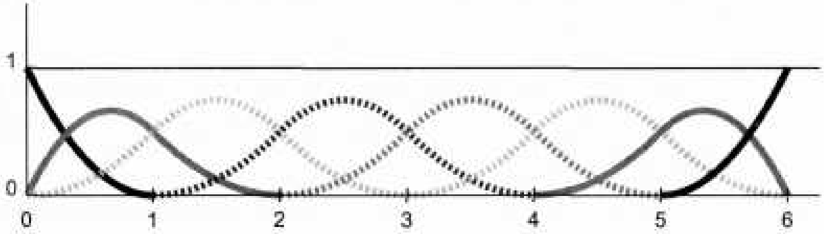


**Figure 15.24.** Endpoint-interpolating quadratic ($k$ =3) B-splines, for $n$ = 8. The knot vector is [0,0,0,1,2,3,4,5,6,6,6]. The first and last two B-splines are aperiodic, while the middle four (shown as dotted lines) are periodic and identical to the ones in Figure 15.20.

$$b_{2,3,[0,0,0,1,2,\ldots]}(t) = \begin{cases} 2u - \frac{3}{2}u^2 & \text{if } 0 \le t < 1 \quad u = t, \\ \frac{1}{2}(1-u)^2 & \text{if } 1 \le t < 2 \quad u = t-1, \\ 0 & \text{otherwise.} \end{cases}$$

### 15.6.4 NURBS

Despite all of the generality B-splines provide, there are some functions that cannot be exactly represented using them. In particular, B-splines cannot represent conic sections. To represent such curves, a ratio of two polynomials is used. Non-uniform B-splines are used to represent both the numerator and the denominator. The most general form of these are Non-Uniform Rational B-Splines, or NURBS for short.

NURBS associate a scalar weight $h_i$ with every control point $\mathbf{p_i}$ and use the same B-splines for both:

$$\mathbf{f}(u) = \frac{\sum_{i=1}^{n} h_i \mathbf{p}_i b_{i,k,\mathbf{t}}}{\sum_{i=1}^{n} h_i b_{i,k,\mathbf{t}}},$$

where $b_{i,k,\mathbf{t}}$ are the B-splines with parameter $k$ and knot vector $\mathbf{t}$.

NURBS are very widely used to represent curves and surfaces in geometric modeling because of the amazing versatility they provide, in addition to the useful properties of B-splines.

## 15.7 Summary

In this chapter, we have discussed a number of representations for free-form curves. The most important ones for computer graphics are:

- Cardinal splines use a set of cubic pieces to interpolate control points. They are generally preferred to interpolating polynomials because they are local and easier to evaluate.

- Bézier curves approximate their control points and have many useful properties and associated algorithms. For this reason, they are popular in graphics applications.

- B-spline curves represent the curve as a linear combination of B-spline functions. They are general and have many useful properties such as being bounded by their convex hull and being variation diminishing. B-splines are often used when smooth curves are desired.

## Notes

The problem of representing shapes mathematically is an entire field unto itself, generally known as Geometric Modeling. Representing curves is just the beginning and is generally a precursor to modeling surfaces and solids. A more thorough discussion of curves can be found in most geometric modeling texts, see for example *Geometric Modeling* (Mortenson, 1985) for a text that is accessible to computer graphics students. Many geometric modeling books specifically focus on smooth curves and surfaces. Texts such as *An Introduction to Splines for Use in Computer Graphics* (Bartels, Beatty, & Barsky, 1987), *Curves and Surfaces for CAGD: A Practical Guide* (Farin, 2002) and *Geometric Modeling with Splines: An Introduction* (E. Cohen, Riesenfeld, & Elber, 2001) provide considerable detail about curve and surface representations. Other books focus on the mathematics of splines; *A Practical Guide to Splines* (De Boor, 2001) is a standard reference.

The history of the development of curve and surface representations is complex, see the chapter by Farin in *Handbook of Computer Aided Geometric Design* (Farin, Hoschek, & Kim, 2002) or the book on the subject *An Introduction to NURBS: With Historical Perspective* (D. F. Rogers, 2000) for a discussion. Many ideas were independently developed by multiple groups who approached the problems from different disciplines. Because of this, it can be difficult to attribute ideas to a single person or to point at the "original" sources. It has also led to a diversity of notation, terminology, and ways of introducing the concepts in the literature.

### 15.7.1 Exercises

For Exercises 1–4, find the constraint matrix, the basis matrix, and the basis functions. To invert the matrices you can use a program such as MATLAB or OCTAVE (a free MATLAB-like system).

1. A line segment: parameterized with $\mathbf{p}_0$ located 25% of the way along the segment ($u = 0.25$), and $\mathbf{p}_1$ located 75% of the way along the segment.

2. A quadratic: parameterized with $\mathbf{p}_0$ as the position of the beginning point ($u = 0$), $\mathbf{p}_1$, the first derivative at the beginning point, and $\mathbf{p}_2$, the second derivative at the beginning point.

3. A cubic: its control points are equally spaced ($\mathbf{p}_0$ has $u = 0$, $\mathbf{p}_1$ has $u = 1/3$, $\mathbf{p}_2$ has $u = 2/3$, and $\mathbf{p}_3$ has $u = 1$).

4. A quintic: (a degree five polynomial, so the matrices will be $6 \times 6$) where $\mathbf{p}_0$ is the beginning position, $\mathbf{p}_1$ is the beginning derivative, $\mathbf{p}_2$ is the middle ($u = 0.5$) position, $\mathbf{p}_3$ is the first derivative at the middle, $\mathbf{p}_4$ is the position at the end, and $\mathbf{p}_5$ is the first derivative at the end.

5. The Lagrange Form (Equation 15.12) can be used to represent the interpolating cubic of Exercise 3. Use it at several different parameter values to confirm that it does produce the same results as the basis functions derived in Exercise 3.

6. Devise an arc-length parameterization for the curve represented by the parametric function
$$f(u) = (u, u^2).$$

7. Given the four control points of a segment of a Hermite spline, compute the control points of an equivalent Bézier segment.

8. Use the de Castijeau algorithm to evaluate the position of the cubic Bézier curve with its control points at (0,0), (0,1), (1,1) and (1,0) for parameter values $u = 0.5$ and $u = 0.75$. Drawing a sketch will help you do this.

9. Use the Cox / de Boor recurrence to derive Equation 15.16.

Michael Ashikhmin

16

# Computer Animation

*Animation* is derived from the Latin *anima* and means the act, process, or result of imparting life, interest, spirit, motion, or activity. Motion is a defining property of life and much of the true art of animation is about how to tell a story, show emotion, or even express subtle details of human character through motion. A computer is a secondary tool for achieving these goals—it is a tool which a skillful animator can use to help get the result he wants faster and without concentrating on technicalities in which he is not interested. Animation without computers, which is now often called "traditional" animation, has a long and rich history of its own which is continuously being written by hundreds of people still active in this art. As in any established field, some time-tested rules have been crystallized which give general high-level guidance to how certain things should be done and what should be avoided. These principles of traditional animation apply equally to computer animation, and we will discuss some of them below.

The computer, however, is more than just a tool. In addition to making the animator's main task less tedious, computers also add some truly unique abilities that were simply not available or were extremely difficult to obtain before. Modern modeling tools allow the relatively easy creation of detailed three-dimensional models, rendering algorithms can produce an impressive range of appearances, from fully photorealistic to highly stylized, powerful numerical simulation algorithms can help to produce desired physics-based motion for particularly hard to animate objects, and motion capture systems give the ability to record and use real-life motion. These developments led to an exploding use of computer animation techniques in motion pictures and commercials, automotive design and architecture, medicine and scientific research among many other areas. Completely new domains and applications have also appeared including fully computer-animated feature films, virtual/augmented reality systems and, of course, computer games.

Other chapters of this book cover many of the developments mentioned above (for example, geometric modeling and rendering) more directly. Here, we will provide an overview only of techniques and algorithms directly used to create and manipulate motion. In particular, we will loosely distinguish and briefly describe four main computer animation approaches:

- **Keyframing** gives the most direct control to the animator who provides necessary data at some moments in time and the computer fills in the rest.
- **Procedural** animation involves specially designed, often empirical, mathematical functions and procedures whose output resembles some particular motion.
- **Physics-based** techniques solve differential equation of motion.
- **Motion capture** uses special equipment or techniques to record real-world motion and then transfers this motion into that of computer models.

We do not touch upon the artistic side of the field at all here. In general, we can not possibly do more here than just scratch the surface of the fascinating subject of creating motion with a computer. We hope that readers truly interested in the subject will continue their journey well beyond the material of this chapter.

## 16.1 Principles of Animation

In his seminal 1987 SIGGRAPH paper (Lasseter, 1987), John Lasseter brought key principles developed as early as the 1930's by traditional animators of Walt Disney studios to the attention of the then-fledgling computer animation community. Twelve principles were mentioned: *squash and stretch; timing; anticipation; follow through and overlapping action; slow-in and slow-out; staging; arcs; secondary action; straight-ahead and pose-to-pose action; exaggeration; solid drawing skill; appeal*. Almost two decades later, these time-tested rules, which can make a difference between a natural and entertaining animation and a mechanistic-looking and boring one, are as important as ever. For computer animation, in addition, it is very important to *balance* control and flexibility given to the animator with the full advantage of the computer's abilities. Although these principles are widely known, many factors affect how much attention is being paid to these rules in practice. While a character animator working on a feature film might spend many hours trying to follow some of these suggestions (for example, tweaking his timing to be just right), many game designers tend to believe that their time is better spent elsewhere.

### 16.1.1 Timing

*Timing*, or the speed of action, is at the heart of any animation. How fast things happen affects the meaning of action, emotional state, and even perceived weight of objects involved. Depending on its speed, the same action, a turn of a character's head from left to right, can mean anything from a reaction to being hit by a heavy object to slowly seeking a book on a bookshelf or stretching a neck muscle. It is very important to set timing appropriate for the specific action at hand. Action should occupy enough time to be noticed while avoiding too slow and potentially boring motions. For computer animation projects involving recorded sound, the sound provides a natural timing anchor to be followed. In fact, in most productions, the actor's voice is recorded first and the complete animation is then synchronized to this recording. Since large and heavy objects tend to move slower than small and light ones (with less acceleration, to be more precise), timing can be used to provide significant information about the weight of an object.

### 16.1.2 Action Layout

At any moment during an animation, it should be clear to the viewer what idea (action, mood, expression) is being presented. Good *staging*, or high-level planning of the action, should lead a viewer's eye to where the important action is currently concentrated, effectively telling him "look at this, and now, look at this" without using any words. Some familiarity with human perception can help us with this difficult task. Since human visual systems react mostly to relative changes rather than absolute values of stimuli, a sudden motion in a still environment or lack of motion in some part of a busy scene naturally draws attention. The same action presented so that the silhouette of the object is changing can often be much more noticeable compared with a frontal arrangement (see Figure 16.1(a)).

On a slightly lower level, each action can be split into three parts: *anticipation* (preparation for the action), the action itself and *follow-through* (termination of the action). In many cases the action itself is the shortest part and, in some sense, the least interesting. For example, kicking a football might involve extensive preparation on the part of the kicker and long "visual tracking" of the departing

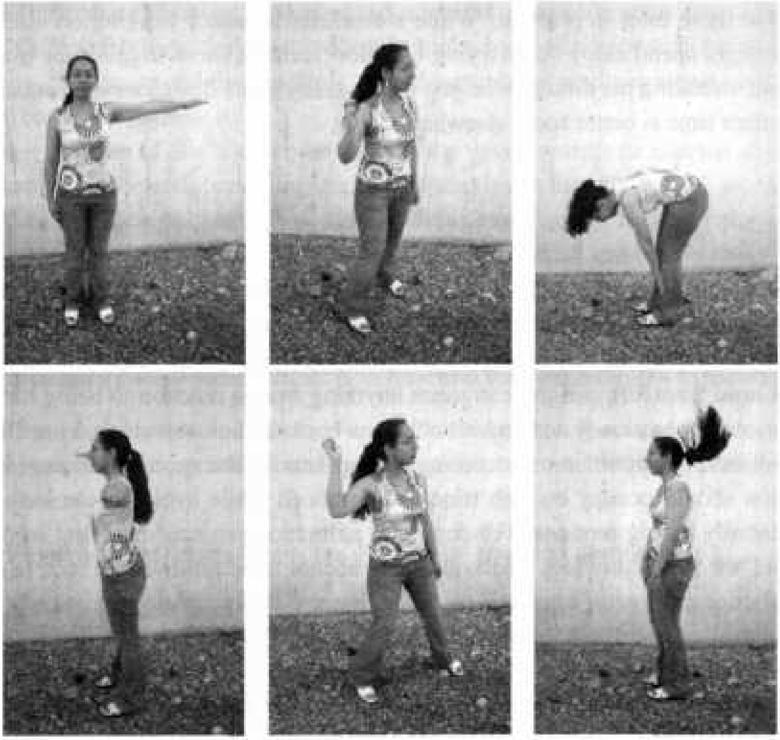


**Figure 16.1.** Action layout: **Left:** Staging action properly is crucial for bringing attention to currently important motion. The act of raising a hand would be prominent on the top but harder to notice on the bottom. A change in nose length, on the contrary, might be completely invisible in the first case. Note that this might be intentionally hidden, for example, to be suddenly revealed later. Neither arrangement is particularly good if both motions should be attended to. **Middle:** The amount of anticipation can tell much about the following action. The action which is about to follow (throwing a ball) is very short but it is clear what is about to happen. The more wound up the character is, the faster the following action is perceived. Without a proper amount of **Right:** The follow-through phase is especially important for secondary appendages (hair) whose motion follows the leading part (head). The motion of the head is very simple, but leads to non-trivial follow-through behavior of the hair itself. It is impossible to create a natural animation without a follow-through phase and overlapping action in this case. *Figure by Peter Shirley and Christina Villarruel.*

ball with ample opportunities to show the stress of the moment, emotional state of the kicker, and even the reaction to the expected result of the action. The action itself (motion of the leg to kick the ball) is rather plain and takes just a fraction of a second in this case.

The goal of anticipation is to prepare the viewer to what is about to happen. This becomes especially important if the action itself is very fast, greatly important, or extremely difficult. Creating a more extensive anticipation for such actions serves to underscore these properties and, in case of fast events, makes sure the action will not be missed (see Figure 16.1(b)).

In real life, the main action often causes one or more other *overlapping actions*. Different appendages or loose parts of the object typically drag behind the main leading section and keep moving for a while in the follow-through part of the main action as shown in Figure 16.1(c). Moreover, the next action often starts before the previous one is completely over. A player might start running while he is still tracking the ball he just kicked. Ignoring such natural flow is generally perceived as if there are pauses between actions and can result in robot-like mechanical motion. While overlapping is necessary to keep the motion natural, *secondary action* is often added by the animator to make motion more interesting and achieve realistic complexity of the animation. It is important not to allow secondary action to dominate the main action.

### 16.1.3 Animation Techniques

Several specific techniques can be used to make motion look more natural. The most important one is probably *squash and stretch* which suggests to change the shape of a moving object in a particular way as it moves. One would generally stretch an object in the direction of motion and squash it when a force is applied to it, as demonstrated in Figure 16.2 for a classic animation of a bouncing ball. It is important to preserve the total volume as this happens to avoid the illusion of growing or shrinking of the object. The greater the speed of motion (or the force), the more stretching (or squashing) is applied. Such deformations are used for several reasons. For very fast motion, an object can move between two sequential frames so quickly that there is no overlap between the object at the time of the current frame and at the time of the previous frame which can lead to strobing (a variant of aliasing). Having the object elongated in the direction of motion can ensure better overlap and helps the eye to fight this unpleasant effect. Stretching/squashing can also be used to show flexibility of the object with more deformation applied for more pliable materials. If the object is intended to appear as rigid, its shape is purposefully left the same when it moves.

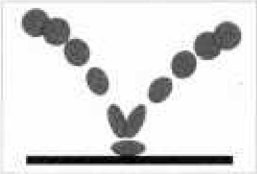


**Figure 16.2.** Classic example of applying the squash and stretch principle. Note that the volume of the bouncing ball should remain roughly the same throughout the animation.

Natural motion rarely happens along straight lines, so this should generally be avoided in animation and *arcs* should be used instead. Similarly, no real-world motion can instantly change its speed—this would require an infinite amount of force to be applied to an object. It is desirable to avoid such situations in animation as well. In particular, the motion should start and end gradually (*slow in and out*). While hand-drawn animation is sometimes done via *straight-ahead action* with an animator starting at the first frame and drawing one frame after another in sequence until the end, *pose-to-pose action*, also known as *keyframing*, is much more suitable for computer animation. In this technique, animation is carefully planned through a series of relatively sparsely spaced key frames with the rest of the animation (in-between frames) filled in only after the keys are set (Figure 16.3). This allows more precise timing and allows the computer to take over the most tedious part of the process—the creation of the in-between frames—using algorithms presented in the next section.

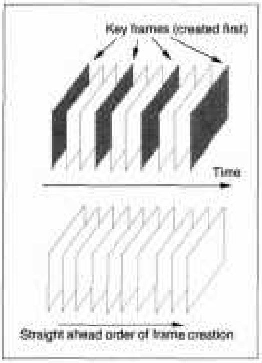

Key frames (created first)
Time
Straight ahead order of frame creation

**Figure 16.3.** Keyframing (top) encourages detailed action planning while straight-ahead action (bottom) leads to a more spontaneous result.

Almost any of the techniques outlined above can be used with some reasonable amount of *exaggeration* to achieve greater artistic effect or underscore some specific property of an action or a character. The ultimate goal is to achieve something the audience will want to see, something which is *appealing*. Extreme complexity or too much symmetry in a character or action tends to be less appealing. To create good results, a traditional animator needs *solid drawing skills*. Analogously, a computer animator should certainly understand computer graphics and have a solid knowledge of the tools he uses.

### 16.1.4 Animator Control vs. Automatic Methods

In traditional animation, the animator has complete control over all aspects of the production process and nothing prevents the final product to be as it was planned in every detail. The price paid for this flexibility is that every frame is created by hand, leading to an extremely time- and labor-consuming enterprise. In computer animation, there is a clear tradeoff between, on the one hand, giving an animator more direct control over the result, but asking him to contribute more work and, on the other hand, relying on more automatic techniques which might require setting just a few input parameters but offer little or no control over some of the properties of the result. A good algorithm should provide sufficient flexibility while asking an animator only the information which is intuitive, easy to provide, and which he himself feels is necessary for achieving the desired effect. While perfect compliance with this requirement is unlikely in practice since it would probably take something close to a mind-reading machine, we do encourage the reader to evaluate any computer-animation technique from the point of view of providing such *balance*.

## 16.2 Keyframing

The term keyframing can be misleading when applied to 3D computer animation since no actual completed frames (i.e., images) are typically involved. At any given moment, a 3D scene being animated is specified by a set of numbers: the

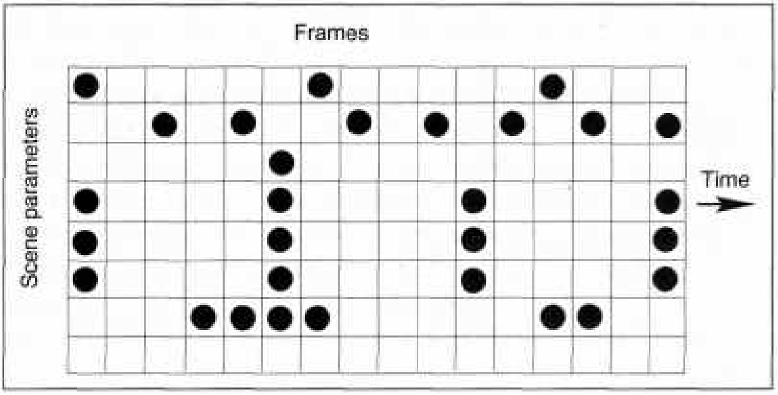


**Figure 16.4.** Different patterns of setting keys (black circles above) can be used simultaneously for the same scene. It is assumed that there are more frames before as well as after this portion.

positions of centers of all objects, their RGB colors, the amount of scaling applied to each object in each axis, modeling transformations between different parts of a complex object, camera position and orientation, light sources intensity, etc. To animate a scene, some subset of these values have to change with time. One can, of course, directly set these values at every frame, but this will not be particularly efficient. Short of that, some number of important moments in time (key frames $t_k$) can be chosen along the timeline of animation for each of the parameters and values of this parameter (key values $f_k$) are set only for these selected frames. We will call a combination $(t_k, f_k)$ of key frame and key value simply a key. Key frames do not have to be the same for different parameters, but it is often logical to set keys at least for some of them simultaneously. For example, key frames chosen for $x$-, $y$- and $z$-coordinates of a specific object might be set at exactly the same frames forming a single position vector key $(t_k, \mathbf{p_k})$. These key frames, however, might be completely different from those chosen for the object's orientation or color. The closer key frames are to each other, the more control the animator has over the result; however the cost of doing more work of setting the keys has to be assessed. It is, therefore, typical to have large spacing between keys in parts of the animation which are relatively simple, concentrating them in intervals where complex action occurs as shown in Figure 16.4.

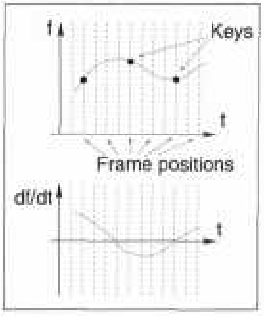


**Figure 16.5.** A continuous curve *f(t)* is fit through the keys provided by the animator even though only values at frame positions are of interest. The derivative of this function gives the speed of parameter change and is at first determined automatically by the fitting procedure.

Once the animator sets the key $(t_k, f_k)$, the system has to compute values of $f$ for all other frames. Although we are ultimately interested only in a discrete set of values, it is convenient to treat this as a classical interpolation problem which fits a continuous *animation curve* $f(t)$ through a provided set of data points (Figure 16.5). Extensive discussion of curve fitting algorithms can be found in Chapter 15, and we will not repeat it here. Since the animator initially provides only the keys and not the derivative (tangent), methods which compute all necessary information directly from keys are preferable for animation. The speed of parameter change along the curve is given by the derivative of the curve with respect to time $df/dt$. Therefore, to avoid sudden jumps in velocity, $C^1$ continuity is typically necessary. A higher degree of continuity is typically not required from animation curves, since the second derivative, which corresponds to acceleration or applied force, can experience very sudden changes in real-world situations (ball hitting a solid wall), and higher derivatives do not directly correspond to any parameters of physical motion. These consideration make Catmull-Rom splines one of the best choices for initial animation curve creation.

Most animation systems give the animator the ability to perform interactive fine editing of this initial curve, including inserting more keys, adjusting existing keys, or modifying automatically computed tangents. Another useful technique which can help to tweak the shape of the curve is called TCB control (TCB stands for tension, continuity and bias). The idea is to introduce three new parameters which can be used to modify the shape of the curve near a key through coordinated adjustment of incoming and outgoing tangents at this point. For keys uniformly spaced in time with distance $\Delta t$ between them, the standard Catmull-Rom expression for incoming $T_i^{in}$ and outgoing $T_i^{out}$ tangents at an internal key $(t_k, f_k)$ can be rewritten as

$$T_k^{in} = T_k^{out} = \frac{1}{2\Delta t}(f_{k+1} - f_k) + \frac{1}{2\Delta t}(f_k - f_{k-1}).$$

Modified tangents of a TCB spline are

$$T_k^{in} = \frac{(1-t)(1-c)(1+b)}{2\Delta t}(f_{k+1} - f_k) + \frac{(1-t)(1+c)(1-b)}{2\Delta t}(f_k - f_{k-1}),$$

$$T_k^{out} = \frac{(1-t)(1+c)(1+b)}{2\Delta t}(f_{k+1} - f_k) + \frac{(1-t)(1-c)(1-b)}{2\Delta t}(f_k - f_{k-1}).$$

The tension parameter $t$ controls the sharpness of the curve near the key by scaling both incoming and outgoing tangents. Larger tangents (lower tension) lead to a flatter curve shape near the key. Bias $b$ allows the animator to selectively increase the weight of a key's neighbors locally pulling the curve closer to a straight line connecting the key with its left ($b$ near 1, "overshooting" the action) or right ($b$ near $-1$, "undershooting" the action) neighbors. A non-zero value of continuity $c$ makes incoming and outgoing tangents different allowing the animator to create kinks in the curve at the key value. Practically useful values of TCB parameters are typically confined to the interval $[-1; 1]$ with defaults $t = c = b = 0$ corresponding to the original Catmull-Rom spline. Examples of possible curve shape adjustments are shown in Figure 16.6.

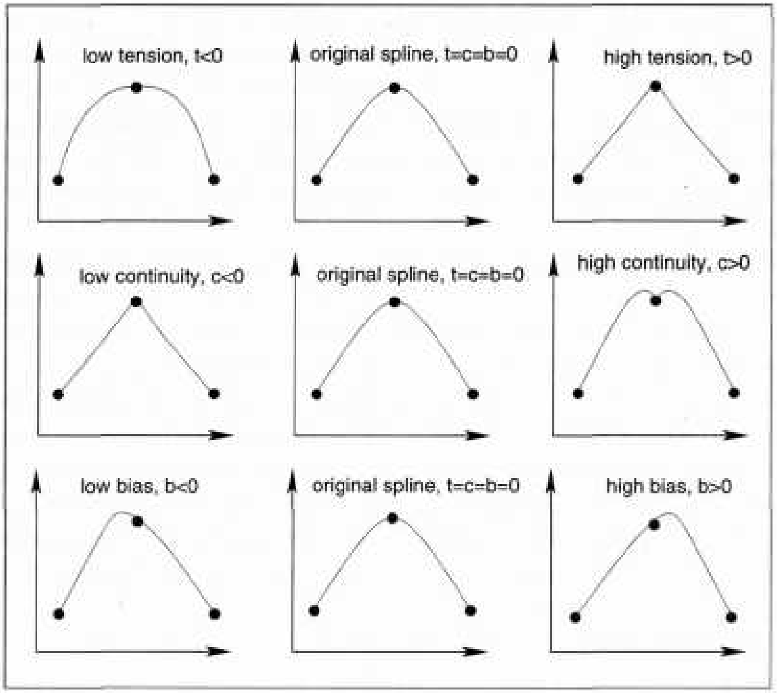


**Figure 16.6.** Editing the default interpolating spline (middle column) using TCB controls. Note that all keys remain at the same positions.

### 16.2.1 Motion Controls

So far, we have described how to control the shape of the animation curve through key positioning and fine tweaking of tangent values at the keys. This, however, is generally not sufficient when one would like to have control both over where the object is moving, i.e., its path, and how fast it moves along this path. Given a set of positions in space as keys, automatic curve-fitting techniques can fit a curve through them, but resulting motion is only constrained by forcing the object to arrive at a specified key position $\mathbf{p_k}$ at the corresponding key frame $t_k$, and nothing is directly said about the speed of motion between the keys. This can create problems. For example, if an object moves along the $x$-axis with velocity 11 meters per second for 1 second and then with 1 meter per second for 9 seconds, it will arrive at position $x = 20$ after 10 seconds thus satisfying animator's keys (0,0) and (10, 20). It is rather unlikely that this jerky motion was actually desired, and uniform motion with speed 2 meters/second is probably closer to what the animator wanted when setting these keys. Although typically not displaying

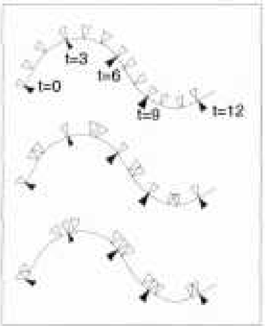


**Figure 16.7.** All three motions are along the same 2D path and satisfy the set of keys at the tips of the black triangles. The tips of the white triangles show object position at $\Delta t = 1$ intervals. Uniform speed of motion between the keys (top) might be closer to what the animator wanted but automatic fitting procedures could result in either of the other two motions.

such extreme behavior, polynomial curves resulting from standard fitting procedures do exhibit non-uniform speed of motion between keys as demonstrated in Figure 16.7. While this can be tolerable (within limits) for some parameters for which the human visual system is not very good at determining non-uniformities in the rate of change (such as color or even rate of rotation), we have to do better for position $\mathbf{p}$ of the object where velocity directly corresponds to everyday experience.

We will first distinguish curve parameterization used during the fitting procedure from that used for animation. When a curve is fit through position keys, we will write the result as a function $\mathbf{p}(u)$ of some parameter $u$. This will describe the geometry of the curve in space. The arc length $s$ is the physical length of the curve. A natural way for the animator to control the motion along the now existing curve is to specify an extra function $s(t)$ which corresponds to how far along the curve the object should be at any given time. To get an actual position in space, we need one more auxiliary function $u(s)$ which computes a parameter value $u$ for given arc length $s$. The complete process of computing an object position for a given time $t$ is then given by composing these functions (see Figure16.8):

$$\mathbf{p}(t) = \mathbf{p}(u(s(t))).$$

Several standard functions can be used as the distance-time function $s(t)$. One of the simplest is the linear function corresponding to constant velocity: $s(t) = vt$ with $v = \text{const}$. Another common example is the motion with constant acceleration $a$ (and initial speed $v_0$) which is described by the parabolic $s(t) = v_0t + at^2/2$. Since velocity is changing gradually here, this function can help to model desirable ease-in and ease-out behavior. More generally, the

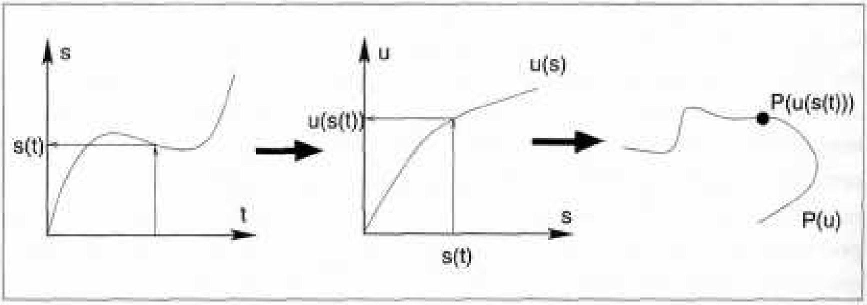
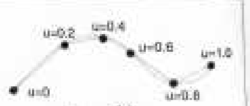


**Figure 16.8.** To get position in space at a given time $t$, one first utilizes user-specified motion control to obtain the distance along the curve $s(t)$ and then compute the corresponding curve parameter value $u(s(t))$. Previously fitted curve $\mathbf{P}(u)$ can now be used to find the position $\mathbf{P}(u(s(t)))$.

slope of $s(t)$ gives the velocity of motion with negative slope corresponding to the motion backwards along the curve. To achieve most flexibility, the ability to interactively edit $s(t)$ is typically provided to the animator by the animation system. The distance-time function is not the only way to control motion. In some cases it might be more convenient for the user to specify a velocity-time function $v(t)$ or even an acceleration-time function $a(t)$. Since these are correspondingly first and second derivatives of $s(t)$, to use these type of controls, the system first recovers the distance-time function by integrating the user input (twice in the case of $a(t)$).

The relationship between the curve parameter $u$ and arc length $s$ is established automatically by the system. In practice, the system first determines arc length dependance on parameter $u$ (i.e., the inverse function $s(u)$). Using this function, for any given $S$ it is possible to solve the equation $s(u) - S = 0$ with unknown $u$ obtaining $u(S)$. For most curves, the function $s(u)$ can not be expressed in closed analytic form and numerical integration is necessary (see Chapter 14). Standard numerical root-finding procedures (such as the Newton-Raphson method, for example) can then be directly used to solve the equation $s(u) - S = 0$ for $u$.



| u | s(u) |
|---|---|
| 0.0 | 0.0 |
| 0.2 | 2.5 |
| 0.4 | 4.0 |
| 0.6 | 5.0 |
| 0.8 | 7.0 |
| 1.0 | 8.5 |

**Figure 16.9.** To create a tabular version of $s(u)$, the curve can be approximated by a number of line segments connecting points on the curve positioned at equal parameter increments. The table is searched to find the $u$-interval for a given $S$. For the curve above, for example, the value of $u$ corresponding to the position of $S = 6.5$ lies between $u = 0.6$ and $u = 0.8$.

An alternative technique is to approximate the curve itself as a set of linear segments between points $\mathbf{p}_i$ computed at some set of sufficiently densely spaced parameter values $u_i$. One then creates a table of approximate arc lengths

$$s(u_i) \approx \sum_{j=1}^{i} ||\mathbf{p_j} - \mathbf{p_{j-1}}|| = s(u_{i-1}) + ||\mathbf{p_i} - \mathbf{p_{i-1}}||.$$

Since $s(u)$ is a non-decreasing function of $u$, one can then find the interval containing the value $S$ by simple searching through the table (see Figure16.9). Linear interpolation of the interval's $u$ end values is then performed to finally find u(S). If greater precision is necessary, a few steps of the Newton-Raphson algorithm with this value as the starting point can be applied.

### 16.2.2 Interpolating Rotation

The techniques presented above can be used to interpolate the keys set for most of the parameters describing the scene. Three-dimensional rotation is one important motion for which more specialized interpolation methods and representations are common. The reason for this is that applying standard techniques to 3D rotations often leads to serious practical problems. Rotation (a change in orientation of an object) is the only motion other than translation which leaves the shape of the object intact. It therefore plays a special role in animating rigid objects.

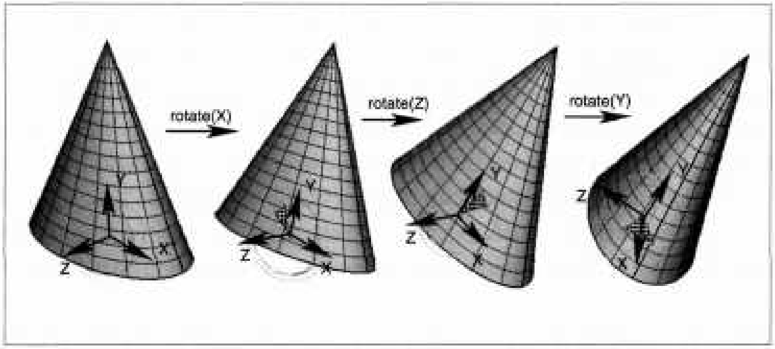


**Figure 16.10.** Three Euler angles can be used to specify arbitrary object orientation through a sequence of three rotations around coordinate axes embedded into the object (axis Y always points to the tip of the cone). Note that each rotation is given in a new coordinate system. Fixed angle representation is very similar but the coordinate axes it uses are fixed in space and do not rotate with the object.

There are several ways to specify the orientation of an object. First, transformation matrices as described in Chapter 6 can be used. Unfortunately, naive (element-by-element) interpolation of rotation matrices does not produce a correct result. For example, the matrix "half-way" between 2D clock- and counterclockwise 90 degree rotation is the null matrix:

$$\frac{1}{2}\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} + \frac{1}{2}\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

The correct result is, of course, the unit matrix corresponding to no rotation. Second, one can specify arbitrary orientation as a sequence of exactly three rotations around coordinate axes chosen in some specific order. These axes can be fixed in space (*fixed-angle* representation) or embedded into the object therefore changing after each rotation (*Euler-angle* representation as shown in Figure 16.10). These three angles of rotation can be animated directly through standard keyframing, but a subtle problem known as gimbal lock arises. Gimbal lock occurs if during rotation one of the three rotation axes is by accident aligned with another, thereby reducing by one the number of available degrees of freedom as shown in Figure 16.11 for a physical device. This effect is more common than one might think—a single 90 degree turn to the right (or left) can potentially put an object into a gimbal lock. Finally, any orientation can be specified by choosing an appropriate axis in space and angle of rotation around this axis. While animating in this representation is relatively straightforward, combining two rotations, i.e., finding the axis and angle corresponding to a sequence of two rotations both represented by axis and angle, is non-trivial. A special mathematical apparatus, *quaternions*

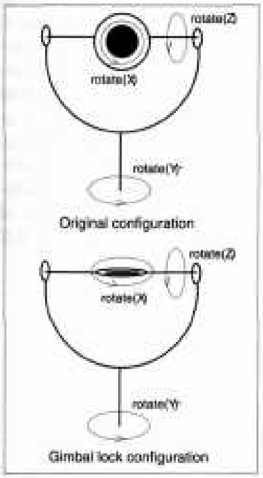


**Figure 16.11.** In this example, gimbal lock occurs when a 90 degree turn around axis Z is made. Both X and Y rotations are now performed around the same axis leading to the loss of one degree of freedom.

has been developed to make this representation suitable both for combining several rotations into a single one and for animation.

Given a 3D vector $\mathbf{v} = (x, y, z)$ and a scalar $s$, a quaternion $q$ is formed by combining the two into a four component object: $q = [s\ x\ y\ z] = [s;\ \mathbf{v}]$. Several new operations are then defined for quaternions. Quaternion addition simply sums scalar and vector parts separately:

$$q_1 + q_2 \equiv [s_1 + s_2;\ \mathbf{v_1} + \mathbf{v_2}].$$

Multiplication by a scalar $a$ gives a new quaternion

$$aq \equiv [as;\ a\mathbf{v}].$$

More complex quaternion multiplication is defined as

$$q_1 \cdot q_2 \equiv [s_1 s_2 - \mathbf{v_1}\mathbf{v_2};\ s_1\mathbf{v_2} + s_2\mathbf{v_1} + \mathbf{v_1} \times \mathbf{v_2}],$$

where × denotes a vector cross product. It is easy to see that, similar to matrices, quaternion multiplication is associative, but not commutative. We will be interested mostly in normalized quaternions—those for which the quaternion norm $|q| = \sqrt{s^2 + \mathbf{v}^2}$ is equal to one. One final definition we need is that of an inverse quaternion:

$$q^{-1} = (1/|q|)[s;\ -\mathbf{v}].$$

To represent a rotation by angle $\phi$ around an axis passing through the origin whose direction is given by the normalized vector $\mathbf{n}$, a normalized quaternion

$$q = [\cos(\phi/2); \sin(\phi/2)\mathbf{n}]$$

is formed. To rotate point $\mathbf{p}$, one turns it into the quaternion $q_p = [0;\ \mathbf{p}]$ and computes the quaternion product

$$q'_p = q \cdot q_p \cdot q^{-1}$$

which is guaranteed to have a zero scalar part and the rotated point as its vector part. Composite rotation is given simply by the product of quaternions representing each of the separate rotation steps. To animate with quaternions, one can treat them as points in a four-dimensional space and set keys directly in this space. To keep quaternions normalized, one should, strictly speaking, restrict interpolation procedures to a unit sphere (a 3D object) in this 4D space. However, a spherical version of even linear interpolation (often called *slerp*) already results in rather unpleasant math. Simple 4D linear interpolation followed by projection onto the unit sphere shown in Figure 16.12 is much simpler and often sufficient in practice. Smoother results can be obtained via repeated application of a linear interpolation procedure using the de Casteljau algorithm.

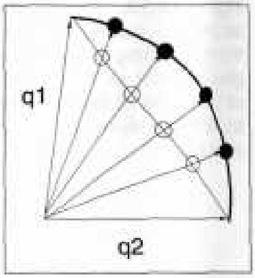


**Figure 16.12.** Interpolating quaternions should be done on the surface of a 3D unit sphere embedded in 4D space. However, much simpler interpolation along a 4D straight line (open circles) followed by re-projection of the results onto the sphere (black circles) is often sufficient.

## 16.3 Deformations

Although techniques for object deformation might be more properly treated as modeling tools, they are traditionally discussed together with animation methods. Probably the simplest example of an operation which changes object shape is a non-uniform scaling. More generally, some function can be applied to local coordinates of all points specifying the object (i.e., vertices of a triangular mesh or control polygon of a spline surface), repositioning these points and creating a new shape: $\mathbf{p}' = f(\mathbf{p}, \gamma)$ where $\gamma$ is a vector of parameters used by the deformation function. Choosing different $f$ (and combining them by applying one after another) can help to create very interesting deformations. Examples of useful simple functions include bend, twist, and taper which are shown in Figure 16.13. Animating shape change is very easy in this case by keyframing the parameters of the deformation function. Disadvantages of this technique include difficulty of choosing the mathematical function for some non-standard deformations and the fact that the resulting deformation is *global* in the sense that the complete object, and not just some part of it, is reshaped.

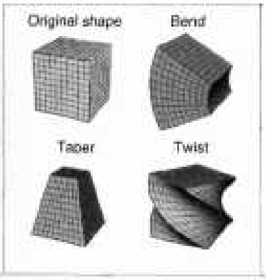


**Figure 16.13.** Popular examples of global deformations. Bending and twist angles as well as the degree of taper can all be animated to achieve dynamic shape change.

To deform an object locally while providing more direct control over the result, one can choose a single vertex, move it to a new location and adjust vertices within some neighborhood to follow the seed vertex. The area affected by the deformation and the specific amount of displacement in different parts of the object are controlled by an attenuation function which decreases with distance (typically computed over the object's surface) to the seed vertex. Seed vertex motion can be keyframed to produce animated shape change.

A more general deformation technique is called free-form deformation (FFD) (Sederberg & Parry, 1986). A local (in most cases rectilinear) coordinate grid is first established to encapsulate the part of the object to be deformed, and coordinates $(s, t, u)$ of all relevant points are computed with respect to this grid. The user then freely reshapes the grid of lattice points $\mathbf{P_{ijk}}$ into a new distorted lattice $\mathbf{P'_{ijk}}$ (Figure 16.14). The object is reconstructed using coordinates computed in the original undistorted grid in the trivariate analog of Bézier interpolants (see Chapter 15) with distorted lattice points $\mathbf{P'_{ijk}}$ serving as control points in this expression:

$$P(s,u,t) = \sum_{i=0}^{L} \binom{i}{L} (1-s)^{L-i} s^i \sum_{j=0}^{M} \binom{j}{M} (1-t)^{M-j} t^j \sum_{k=0}^{N} \binom{k}{N} (1-u)^{N-k} u^k \mathbf{P'_{ijk}},$$

where $L, M, N$ are maximum indices of lattice points in each dimension. In effect, the lattice serves as a low resolution version of the object for the purpose of deformation, allowing for a smooth shape change of an arbitrarily complex object through a relatively small number of intuitive adjustments. FFD lattices can themselves be treated as regular objects by the system and can be transformed, animated, and even further deformed if necessary, leading to corresponding changes in the object to which the lattice is attached. For example, moving a *deformation tool* consisting of the original lattice and distorted lattice representing a bulge across an object results in a bulge moving across the object.

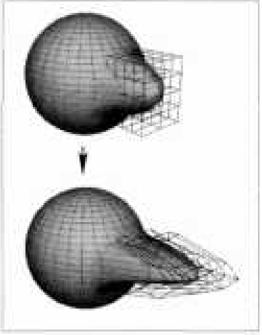


**Figure 16.14.** Adjusting the FFD lattice results in the deformation of the object.

## 16.4 Character Animation

Animation of articulated figures is most often performed through a combination of keyframing and specialized deformation techniques. The character model intended for animation typically consists of at least two main layers as shown in Figure 16.15. The motion of a highly detailed surface representing the outer shell or *skin* of the character is what the viewer will eventually see in the final product. The *skeleton* underneath it is a hierarchical structure (a tree) of joints which provides a kinematic model of the figure and is used exclusively for animation. In some cases, additional intermediate layer(s) roughly corresponding to muscles are inserted between the skeleton and the skin.

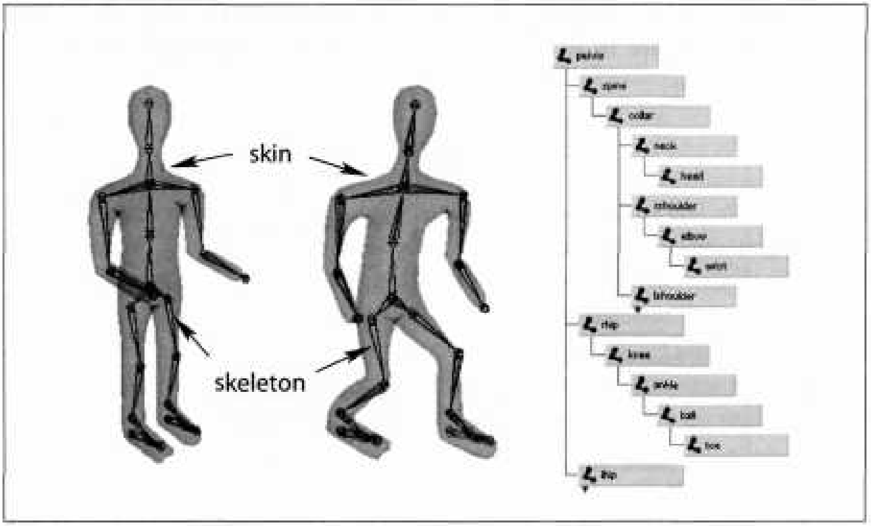


**Figure 16.15.** (Left) A hierarchy of joints, a skeleton, serves as a kinematic abstraction of the character; (middle) repositioning the skeleton deforms a separate skin object attached to it; (right) a tree data structure is used to represent the skeleton. For compactness, the internal structure of several nodes is hidden (they are identical to a corresponding sibling).

Each of the skeleton's joints acts as a parent for the hierarchy below it. The root represents the whole character and is positioned directly in the world coordinate system. If a local transformation matrix which relates a joint to its parent in the hierarchy is available, one can obtain a transformation which relates local space of any joint to the world system (i.e., the system of the root) by simply concatenating transformations along the path from the root to the joint. To evaluate the whole skeleton (i.e., find position and orientation of all joints), a depth-first traversal of the complete tree of joints is performed. A transformation stack is a natural data structure to help with this task. While traversing down the tree, the current composite matrix is pushed on the stack and new one is created by multiplying the current matrix with the one stored at the joint. When backtracking to the parent, this extra transformation should be undone before another branch is visited; this is easily done by simply popping the stack. Although this general and simple technique for evaluating hierarchies is used throughout computer graphics, in animation (and robotics) it is given a special name—*forward kinematics* (FK). While general representations for all transformations can be used, it is common to use specialized sets of parameters, such as link lengths or joint angles, to specify skeletons. To animate with forward kinematics, rotational parameters of all joints are manipulated directly. The technique also allows the animator to change the distance between joints (link lengths), but one should be aware that this corresponds to limb stretching and can often look rather unnatural.

Forward kinematics requires the user to set parameters for all joints involved in the motion (Figure 16.16 (top)). Most of these joints, however, belong to in-

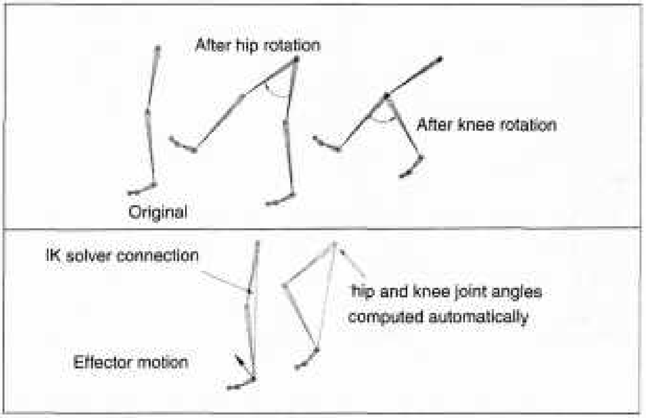


**Figure 16.16.** Forward kinematics (top) requires the animator to put all joints into correct position. In inverse kinematic (bottom), parameters of some internal joints are computed based on desired end effector motion.

ternal nodes of the hierarchy, and their motion is typically not something the animator wants to worry about. In most situations, the animator just wants them to move naturally "on their own," and one is much more interested in specifying the behavior of the end point of a joint chain, which typically corresponds to something performing a specific action, such as an ankle or a tip of a finger. The animator would rather have parameters of all internal joints be determined from the motion of the end effector automatically by the system. *Inverse kinematics* (IK) allows us to do just that (see Figure 16.16(bottom)).

Let $\mathbf{x}$ be the position of the end effector and $\alpha$ be the vector of parameters needed to specify all internal joints along the chain from the root to the final joint. Sometimes the orientation of the final joint is also directly set by the animator, in which case we assume that the corresponding variables are included in the vector $\mathbf{x}$. For simplicity, however, we will write all specific expressions for the vector:

$$\mathbf{x} = (x_1, x_2, x_3)^T.$$

Since each of the variables in $\mathbf{x}$ is a function of $\alpha$, it can be written as a vector equation $\mathbf{x} = \mathbf{F}(\alpha)$. If we change the internal joint parameters by a small amount $\delta\alpha$, a resulting change $\delta\mathbf{x}$ in the position of the end effector can be approximately written as

$$\delta\mathbf{x} = \frac{\partial \mathbf{F}}{\partial \alpha}\delta\alpha, \tag{16.1}$$

where $\frac{\partial \mathbf{F}}{\partial \alpha}$ is the matrix of partial derivatives called the Jacobian:

$$\frac{\partial \mathbf{F}}{\partial \alpha} = \begin{bmatrix} \frac{\partial f_1}{\partial \alpha_1} & \frac{\partial f_1}{\partial \alpha_2} & \cdots & \frac{\partial f_1}{\partial \alpha_n} \\ \frac{\partial f_2}{\partial \alpha_1} & \frac{\partial f_2}{\partial \alpha_2} & \cdots & \frac{\partial f_2}{\partial \alpha_n} \\ \frac{\partial f_3}{\partial \alpha_1} & \frac{\partial f_3}{\partial \alpha_2} & \cdots & \frac{\partial f_3}{\partial \alpha_n} \end{bmatrix}.$$

At each moment in time, we know the desired position of the end effector (set by the animator) and, of course, the effector's current position. Subtracting the two, we will get the desired adjustment $\delta\mathbf{x}$. Elements of the Jacobian matrix are related to changes in a coordinate of the end effector when a particular internal parameter is changed while others remain fixed (see Figure 16.17). These elements can be computed for any given skeleton configuration using geometric relationships. The only remaining unknowns in the system of equations 16.1 are the changes in internal parameters $\alpha$. Once we solve for them, we update $\alpha = \alpha + \delta\alpha$ which gives all the necessary information for the FK procedure to reposition the skeleton.

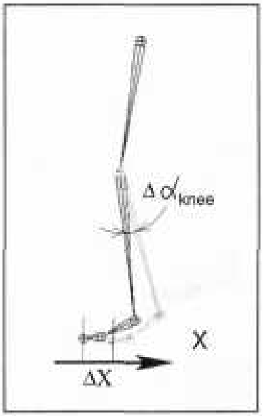


**Figure 16.17.** Partial derivative $\partial x/\partial\alpha_{\text{knee}}$ is given by the limit of $\Delta x/\Delta\alpha_{\text{knee}}$. Effector displacement is computed while all joints, except the knee, are kept fixed.

Unfortunately, the system 16.1 can not usually be solved analytically and, moreover, it is in most cases underconstrained, i.e., the number of unknown internal joint parameters $\alpha$ exceeds the number of variables in vector $\mathbf{x}$. This means that different motions of the skeleton can result in the same motion of the end effector. Some examples are shown on Figure 16.18. Many ways of obtaining specific solution for such systems are available, including those taking into account natural *constraints* needed for some real-life joints (bending a knee only in one direction, for example). One should also remember that the computed Jacobian matrix is valid only for one specific configuration, and it has to be updated as the skeleton moves. The complete IK framework is presented in Figure 16.19. Of course, the root joint for IK does not have to be the root of the whole hierarchy, and multiple IK solvers can be applied to independent parts of the skeleton. For example, one can use separate solvers for right and left feet and yet another one to help animate grasping with the right hand, each with its own root.

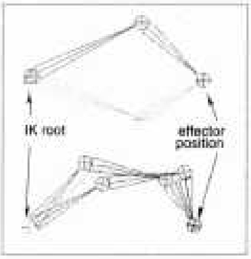


**Figure 16.18.** Multiple configurations of internal joints can result in the same effector position. (Top) disjoint "flipped" solutions; (bottom) *a continuum* of solutions.

A combination of FK and IK approaches is typically used to animate the skeleton. Many common motions (walking or running cycles, grasping, reaching, etc.) exhibit well-known patterns of mutual joint motion making it possible to quickly create naturally looking motion or even use a library of such "clips." The animator then adjusts this generic result according to the physical parameters of the character and also to give it more individuality.

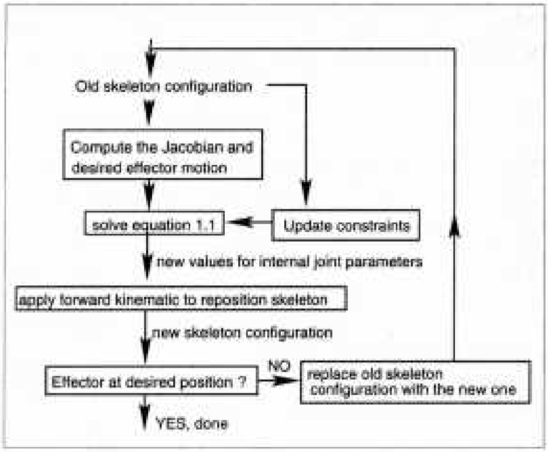

Old skeleton configuration
Compute the Jacobian and desired effector motion
solve equation 1.1
Update constraints
new values for internal joint parameters
apply forward kinematic to reposition skeleton
new skeleton configuration
Effector at desired position ?
NO
replace old skeleton configuration with the new one
YES, done

**Figure 16.19.** A diagram of the inverse kinematic algorithm.

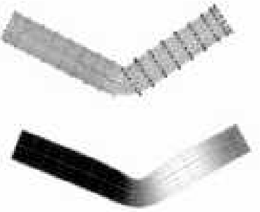


**Figure 16.20.** Top: Rigid skinning assigns skin vertices to a specific joint. Those belonging to the elbow joint are shown in black; Bottom: Soft skinning can blend the influence of several joints. Weights for the elbow joint are shown (lighter = greater weight). Note smoother skin deformation of the inner part of the skin near the joint.

When a skeleton changes its position, it acts as a special type of deformer applied to the skin of the character. The motion is transferred to this surface by assigning each skin vertex one (*rigid skinning*) or more (*smooth skinning*) joints as drivers (see Figure 16.20). In the first case, a skin vertex is simply frozen into the local space of the corresponding joint, which can be the one nearest in space or one chosen directly by the user. The vertex then repeats whatever motion this joint experiences, and its position in world coordinates is determined by standard FK procedure. Although it is simple, rigid skinning makes it difficult to obtain sufficiently smooth skin deformation in areas near the joints or also for more subtle effects resembling breathing or muscle action. Additional specialized deformers called *flexors* can be used for this purpose. In smooth skinning, several joints can influence a skin vertex according to some weight assigned by the animator, providing more detailed control over the results. Displacement vectors, $\mathbf{d}_i$, suggested by different joints affecting a given skin vertex (each again computed with standard FK) are averaged according to their weights $w_i$ to compute the final displacement of the vertex $\mathbf{d} = \sum w_i \mathbf{d}_i$. Normalized weights ($\sum w_i = 1$) are the most common but not fundamentally necessary. Setting smooth skinning weights to achieve the desired effect is not easy and requires significant skill from the animator.

### 16.4.1 Facial Animation

Skeletons are well suited for creating most motions of a character's body, but they are not very convenient for realistic facial animation. The reason is that the skin of a human face is moved by muscles directly attached to it contrary to other parts of the body where the primary objective of the muscles is to move the bones of the skeleton and any skin deformation is a secondary outcome. The result of this facial anatomical arrangement is a very rich set of dynamic facial expressions humans use as one of the main instruments of communication. We are all very well trained to recognize such facial variations and can easily notice any unnatural appearance. This not only puts special demands on the animator but also requires a high-resolution geometric model of the face and, if photorealism is desired, accurate skin reflection properties and textures.

While it is possible to set key poses of the face vertex-by-vertex and interpolate between them or directly simulate the behavior of the underlying muscle structure using physics-based techniques (see Section 16.5 below), more specialized high-level approaches also exist. The static shape of a specific face can be characterized by a relatively small set of so-called *conformational parameters* (overall scale, distance from the eye to the forehead, length of the nose, width of the jaws, etc.) which are used to morph a generic face model into one with individual features. An additional set of *expressive parameters* can be used to describe the dynamic shape of the face for animation. Examples include rigid rotation of the head, how wide the eyes are open, movement of some feature point from its static position, etc. These are chosen so that most of the interesting expressions can be obtained through some combination of parameter adjustments, therefore, allowing a face to be animated via standard keyframing. To achieve a higher level of control, one can use expressive parameters to create a set of expressions corresponding to common emotions (neutral, sadness, happiness, anger, surprise, etc.) and then blend these key poses to obtain a "slightly sad" or "angrily surprised" face. Similar techniques can be used to perform lip-synch animation, but key poses in this case correspond to different phonemes. Instead of using a sequence of static expressions to describe a dynamic one, the Facial Action Coding System (FACS) (Eckman & Friesen, 1978) decomposes dynamic facial expressions directly into a sum of elementary motions called action units (AUs). The set of AUs is based on extensive psychological research and includes such movements as raising the inner brow, wrinkling the nose, stretching lips, etc. Combining AUs can be used to synthesize a necessary expression.

### 16.4.2 Motion Capture

Even with the help of the techniques described above, creating realistic-looking character animation from scratch remains a daunting task. It is therefore only natural that much attention is directed towards techniques which record an actor's motion in the real world and then apply it to computer-generated characters. Two main classes of such *motion capture* (MC) techniques exist: electromagnetic and optical.

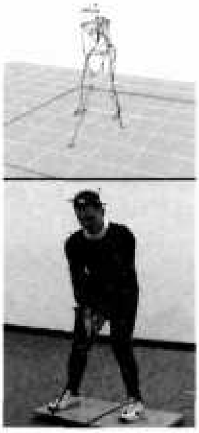


**Figure 16.21.** Optical motion capture: markers attached to a performer's body allow skeletal motion to be extracted. *Image courtesy of Motion Analysis Corp.*

In electromagnetic motion capture, an electromagnetic sensor directly measures its position (and possibly orientation) in 3D often providing the captured results in real time. Disadvantages of this technique include significant equipment cost, possible interference from nearby metal objects, and noticeable size of sensors and batteries which can be an obstacle in performing high-amplitude motions. In optical MC, small colored markers are used instead of active sensors making it a much less intrusive procedure. Figure 16.21 shows the operation of such a system. In the most basic arrangement, the motion is recorded by two calibrated video cameras, and simple triangulation is used to extract the marker's 3D position. More advanced computer vision algorithms used for accurate tracking of multiple markers from video are computationally expensive, so, in most cases, such processing is done offline. Optical tracking is generally less robust than electromagnetic. Occlusion of a given marker in some frames, possible misidentification of markers, and noise in images are just a few of the common problem which have to be addressed. Introducing more cameras observing the motion from different directions improves both accuracy and robustness, but this approach is more expensive and it takes longer to process such data. Optical MC becomes more attractive as available computational power increases and better computer vision algorithms are developed. Because of low impact nature of markers, optical methods are suitable for delicate facial motion capture and can also be used with objects other than humans—for example, animals or even tree branches in the wind.

With several sensors or markers attached to a performer's body, a set of time-dependant 3D positions of some collection of points can be recorded. These tracking locations are commonly chosen near joints, but, of course, they still lie on skin surface and not at points where actual bones meet. Therefore, some additional care and a bit of extra processing is necessary to convert recorded positions into those of the physical skeleton joints. For example, putting two markers on opposite sides of the elbow or ankle allows the system to obtain better joint position by averaging locations of the two markers. Without such extra care, very noticeable artifacts can appear due to offset joint positions as well as inherent noise and insufficient measurement accuracy. Because of physical inaccuracy during motion, for example, character limbs can loose contact with objects they are supposed to touch during walking or grasping, problems like foot-sliding (skating) of the skeleton can occur. Most of these problems can be corrected by using inverse kinematics techniques which can explicitly force the required behavior of the limb's end.

Recovered joint positions can now be directly applied to the skeleton of a computer-generated character. This procedure assumes that the physical dimensions of the character are identical to those of the performer. Retargeting recorded motion to a different character and, more generally, editing MC data, requires significant care to satisfy necessary constraints (such as maintaining feet on the ground or not allowing an elbow to bend backwards) and preserve an overall natural appearance of the modified motion. Generally, the greater the desired change from the original, the less likely it will be possible to maintain the quality of the result. An interesting approach to the problem is to record a large collection of motions and stich together short clips from this library to obtain desired movement. Although this topic is currently a very active research area, limited ability to adjust the recorded motion to the animator's needs remains one of the main disadvantages of motion capture technique.

## 16.5 Physics-Based Animation

The world around us is governed by physical laws many of which can be formalized as sets of partial or, in some simpler cases, ordinary differential equations. One of the original applications of computers was (and remains) solving such equations. It is therefore only natural to attempt to use numerical techniques developed over the several past decades to obtain realistic motion for computer animation.

Because of its relative complexity and significant cost, physics-based animation is most commonly used in situations when other techniques are either unavailable or do not produce sufficiently realistic results. Prime examples include

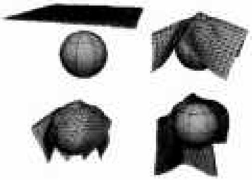


Figure 16.22. Realistic cloth simulation is often performed with physics-based methods. In this example, forces are due to collisions and gravity.

animation of fluids (which includes many gaseous phase phenomena described by the same equations—smoke, clouds, fire, etc.), cloth simulation (an example is shown in Figure 16.22), rigid body motion, and accurate deformation of elastic objects. Governing equations and details of commonly used numerical approaches are different in each of these cases, but many fundamental ideas and difficulties remain applicable across applications. Many methods for numerically solving ODEs and PDEs exist but discussing them in details is far beyond the scope of this book. To give the reader a flavor of physics-based techniques and some of the issues involved, we will briefly mention here only the finite difference approach—one of the conceptually simplest and most popular families of algorithms which has been applied to most, if not all, differential equations encountered in animation.

The key idea of this approach is to replace a differential equation with its discrete analog—a *difference equation*. To do this, the continuous domain of interest is represented by a finite set of points at which the solution will be computed. In the simplest case, these are defined on a uniform rectangular grid as shown in Figure 16.23. Every derivative present in the original ODE or PDE is then replaced by its approximation through function values at grid points. One way of doing this is to subtract the function value at a given point from the function value for its neighboring point on the grid:

$$\frac{df(t)}{dt} \approx \frac{\Delta f}{\Delta t} = \frac{f(t+\Delta t) - f(t)}{\Delta t} \text{ or } \frac{\partial f(x,t)}{\partial x} \approx \frac{\Delta f}{\Delta x} = \frac{f(x+\Delta x,t) - f(x,t)}{\Delta x}. \tag{16.2}$$

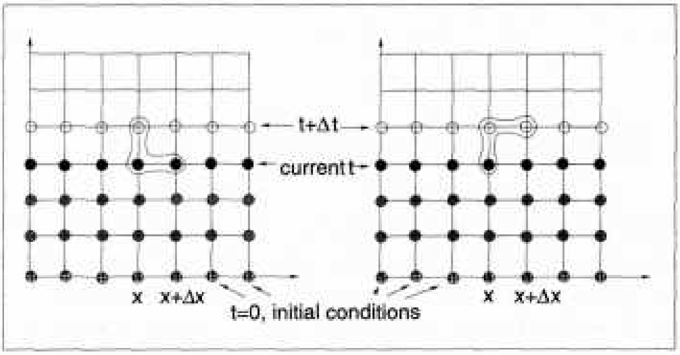


Figure 16.23. Two possible difference schemes for an equation involving derivatives $\partial f/\partial x$ and $\partial f/\partial t$. (Left) An explicit scheme expresses unknown values (open circles) only through known values at the current (black circles) and possibly past (gray circles) time; (Right) Implicit schemes mix known and unknown values in a single equation making it necessary to solve all such equations as a system. For both schemes, information about values on the right boundary is needed to close the process.

These expressions are, of course, not the only way. One can, for example, use $f(t - \Delta t)$ instead of $f(t)$ above and divide by $2\Delta t$. For an equation containing a time derivative, it is now possible to propagate values of an unknown function forward in time in a sequence of $\Delta t$-size steps by solving the system of difference equations (one at each spatial location) for unknown $f(t + \Delta t)$. Some initial conditions, i.e., values of the unknown function at $t = 0$, are necessary to start the process. Other information, such as values on the boundary of the domain, might also be required depending on the specific problem.

The computation of $f(t+\Delta t)$ can be done easily for so called *explicit* schemes when all other values present are taken at the current time and the only unknown in the corresponding difference equation $f(t + \Delta t)$ is expressed through these known values. *Implicit* schemes mix values at current and future times and might use, for example,

$$\frac{f(x + \Delta x, t + \Delta t) - f(x, t + \Delta t)}{\Delta x}$$

as an approximation of $\frac{\partial f}{\partial x}$. In this case one has to solve a system of algebraic equations at each step.

The choice of difference scheme can dramatically affect all aspects of the algorithm. The most obvious among them is *accuracy*. In the limit $\Delta t \rightarrow 0$ or $\Delta x \rightarrow 0$, expressions of the type in Equation 16.2 are exact, but for finite step size some schemes allow better approximation of the derivative than others. *Stability* of a difference scheme is related to how fast numerical errors, which are always present in practice, can grow with time. For stable schemes this growth is bounded, while for unstable ones it is exponential and can quickly overwhelm the solution one seeks (see Figure 16.24). It is important to realize that while some inaccuracy in the solution is tolerable (and, in fact, accuracy demanded in physics and engineering is rarely needed for animation), an unstable result is completely meaningless, and one should avoid using unstable schemes. Generally, explicit schemes are either unstable or can become unstable at larger step sizes while implicit ones are unconditionally stable. Implicit schemes allows greater step size (and, therefore, fewer steps) which is why they are popular despite the need to solve a system of algebraic equations at each step. Explicit schemes are attractive because of their simplicity if their stability conditions can be satisfied. Developing a good difference scheme and corresponding algorithm for a specific problem is not easy, and for most standard situations it is well advised to use an existing method. Ample literature discussing details of these techniques is available.

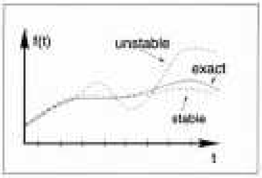


**Figure 16.24.** An unstable solution might follow the exact one initially, but can deviate arbitrarily far from it with time. Accuracy of a stable solution might still be insufficient for a specific application.

One should remember that, in many cases, just computing all necessary terms in the equation is a difficult and time-consuming task on its own. In rigid body or cloth simulation, for example, most of the forces acting on the system are due to collisions among objects. At each step during animation, one therefore has to solve a purely geometric, but very non-trivial, problem of collision detection. In such conditions, schemes which require fewer evaluations of such forces might provide significant computational savings.

Although the result of solving appropriate time-dependant equations gives very realistic motion, this approach has its limitations. First of all, it is very hard to control the result of physics-based animation. Fundamental mathematical properties of these equations state that once the initial conditions are set, the solution is uniquely defined. This does not leave much room for animator input and, if the result is not satisfactory for some reason, one has only a few options. They are mostly limited to adjusting initial condition used, changing physical properties of the system, or even modifying the equations themselves by introducing artificial terms intended to "drive" the solution in the direction the animator wants. Making such changes requires significant skill as well as understanding of the underlying physics and, ideally, numerical methods. Without this knowledge, the realism provided by physics-based animation can be destroyed or severe numerical problems might appear.

## 16.6 Procedural Techniques

Imagine that one could write (and implement on a computer) a mathematical function which outputs precisely the desired motion given some animator guidance. Physics-based techniques outlined above can be treated as a special case of such an approach when the "function" involved is the procedure to solve a particular differential equation and "guidance" is the set of initial and boundary conditions, extra equation terms, etc.

However, if we are only concerned with the final result, we do not have to follow a physics-based approach. For example, a simple constant amplitude wave on the surface of a lake can be directly created by applying the function $f(\mathbf{x}, t) = A\cos(\omega t - \mathbf{k}\mathbf{x} + \phi)$ with constant frequency $\omega$, wave vector $\mathbf{k}$ and phase $\phi$ to get displacement at the 2D point $\mathbf{x}$ at time $t$. A collection of such waves with random phases and appropriately chosen amplitudes, frequencies, and wave vectors can result in a very realistic animation of the surface of water without explicitly solving any fluid dynamics equations. It turns out that other rather simple mathematical functions can also create very interesting patterns or objects. Several such functions, most based on lattice noises, have been described in Chapter 11. Adding time dependance to these functions allows us to animate certain complex phenomena much easier and cheaper than with physics-based techniques while maintaining very high visual quality of the results. If $\mathit{noise}(\mathbf{x})$ is the underlying pattern-generating function, one can create a time-dependant variant of it by moving the argument position through the lattice. The simplest case is motion with constant speed: $\mathit{timenoise}(\mathbf{x}, t) = \mathit{noise}(\mathbf{x} + \mathbf{v}t)$, but more complex motion through the lattice is, of course, also possible and, in fact, more common. One such path, a spiral, is shown in Figure 16.25. Another approach is to animate parameters used to generate the *noise* function. This is especially appropriate if the appearance changes significantly with time—a cloud becoming more turbulent, for example. In this way one can animate the dynamic process of formation of clouds using the function which generates static ones.

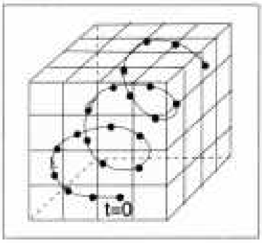


**Figure 16.25.** A path through the cube defining procedural noise is traversed to animate the resulting pattern.

For some procedural techniques, time dependance is a more integral component. The simplest *cellular automata* operate on a 2D rectangular grid where a binary value is stored at each location (cell). To create a time varying pattern, some user-provided rules for modifying these values are repeatedly applied. Rules typically involve some set of conditions on the current value and that of the cell's neighbors. For example, the rules of the popular 2D *Game of Life* cellular automaton invented in 1970 by British mathematician John Conway are:

1. A dead cell (i.e., binary value at a given location is 0) with exactly three live neighbors becomes a live cell (i.e., its value set to 1);

2. A live cell with two or three live neighbors stays alive;

3. In all other cases, a cell dies or remains dead.

Once the rules are applied to all grid locations, a new pattern is created and a new evolution cycle can be started. Three sample snapshots of the live cell distribution at different times are shown in Figure 16.26. More sophisticated automata

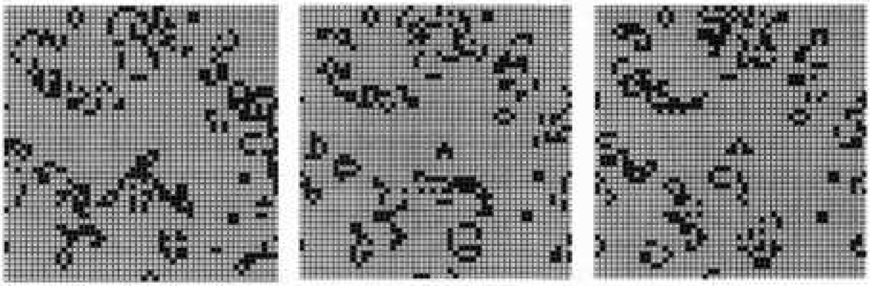


**Figure 16.26.** Several (non-consecutive) stages in the evolution of a *Game of Life* automaton. Live cells are shown in black. Stable objects, oscillators, travelling patterns, and many other interesting constructions can result from the application of very simple rules. *Figure created using a program by Alan Hensel.*

simultaneously operate on several 3D grids of possibly floating point values and can be used for modeling dynamics of clouds and other gaseous phenomena or biological systems for which this apparatus was originally invented (note the terminology). Surprising pattern complexity can arise from just a few well-chosen rules, but how to write such rules to create the desired behavior is often not obvious. This is a common problem with procedural techniques: there is only limited, if any, guidance on how to create new procedures or even adjust parameters of existing ones. Therefore, a lot of tweaking and learning by trial-and-error ("by experience") is usually needed to unlock the full potential of procedural methods.

Another interesting approach which was also originally developed to describe biological objects is the technique called *L-systems* (after the name of their original inventor, Astrid Lindenmayer). This approach is based on *grammars* or sets of recursive rules for rewriting strings of symbols. There are two types of symbols: *terminal symbols* stand for elements of something we want to represent with a grammar. Depending on their meaning, grammars can describe structure of trees and bushes, buildings and whole cities, or programming and natural languages. In animation, L-systems are most popular for representing plants and corresponding terminals are instructions to the geometric modeling system: put a leaf (or a branch) at a current position—we will use the symbol @ and just draw a circle, move current position forward by some number of units (symbol $f$), turn current direction 60 degrees around world Z-axis (symbol $+$), pop (symbol [) or push (symbol ]) current position/orientation, etc. Auxiliary *nonterminal symbols* (denoted by capital letters) have only semantic rather than any direct meaning. They are intended to be eventually rewritten through terminals. We start from the special nonterminal start symbol $S$ and keep applying grammar rules to the current string in parallel, i.e., replace all nonterminals currently present to get the new string, until we end up with a string containing only terminals and no more substitution is therefore possible. This string of modeling instructions is then used to output the actual geometry. For example, a set of rules (productions)

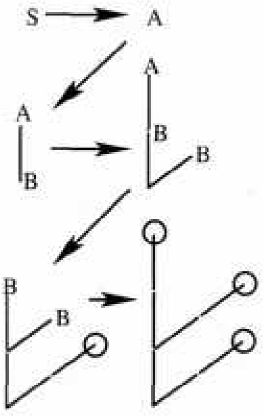


**Figure 16.27.** Consecutive derivation steps using a simple L-system. Capital letters denote non-terminals and illustrate positions at which corresponding non-terminal will be expanded. They are not part of the actual output.

$$
\begin{aligned}
S &\rightarrow A \\
A &\rightarrow [+B]fA \\
A &\rightarrow B \\
B &\rightarrow fB \\
B &\rightarrow f@
\end{aligned}
$$

might result in the following sequence of rewriting steps demonstrated in Figure 16.27

$$
S \longmapsto A \longmapsto [+B]fA \longmapsto [+fB]f[+B]fA \longmapsto
$$
$$
[+ff@]f[+fB]fB \longmapsto [+ff@]f[+ff@]ff@
$$

As shown above, there are typically many different productions for the same nonterminal allowing the generation of many different objects with the same grammar. The choice of which rule to apply can depend on which symbols are located next to the one being replaced (context-sensitivity) or can be performed at random with some assigned probability for each rule (stochastic L-systems). More complex rules can model interaction with the environment, such as pruning to a particular shape, and parameters can be associated with symbols to control geometric commands issued.

L-systems already capture plant topology changes with time: each intermediate string obtained in the rewriting process can be interpreted as a "younger" version of the plant (see Figure 16.27). For more significant changes, different productions can be in effect at different times allowing the structure of the plant to change significantly as it grows. A young tree, for example, produces a lot of new branches while an older one branches only moderately.

Very realistic plant models have been created with L-systems. However, as with most procedural techniques, one needs some experience to meaningfully apply existing L-systems, and writing new grammars to capture some desired effect is certainly not easy.

## 16.7 Groups of Objects

To animate multiple objects one can, of course, simply apply standard techniques outlined above to each of them. This works reasonably well for a moderate number of independent objects whose desired motion is known in advance. However, in many cases, some kind of coordinated action in a dynamic environment is necessary. If only a few objects are involved, the animator can use an artificial intelligence (AI)-based system to automatically determine immediate tasks for each object based on some high-level goal, plan necessary motion, and execute the plan. Many modern games use such *autonomous objects* to create smart monsters or player's collaborators.

Interestingly, as the number of objects in a group grows from just a few to several dozens, hundreds, and thousands, individual members of a group must have only very limited "intelligence" in order for the group as a whole to exhibit what looks like coordinated goal-driven motion. It turns out that this *flocking* is *emergent behavior* which can arise as a result of limited interaction of group members with just a few of their closest neighbors (Reynolds, 1987). Flocking should be familiar to anyone who has observed the fascinatingly synchronized motion of a flock of birds or a school of fish. The technique can also be used to control groups of animals moving over terrain or even a human crowd.

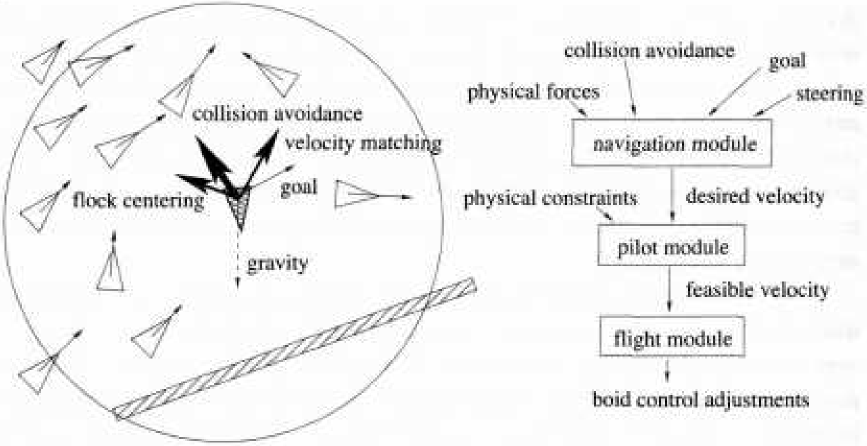


**Figure 16.28.** (Left) Individual flock member (boid) can experience several urges of different importance (shown by line thickness) which have to be negotiated into a single velocity vector. A boid is aware of only its limited neighborhood (circle). (Right) Boid control is commonly implemented as three separate modules.

At any given moment, the motion of a member of a group, often called boid when applied to flocks, is the result of balancing several often contradictory tendencies, each of which suggests its own velocity vector (see Figure 16.28). First, there are external physical forces $F$ acting on the boid, such as gravity or wind. New velocity due to those forces can be computed directly through Newton's law as

$$\mathbf{v}_{new}^{physics} = \mathbf{v}_{old} + \mathbf{F}\Delta t/m.$$

Second, a boid should react to global environment and to the behavior of other group members. Collision avoidance is one of the main results of such interaction. It is crucial for flocking that each group member has only limited field of view, and therefore is aware only of things happening within some neighborhood of its current position. To avoid objects in the environment, the simplest, if imperfect, strategy is to set up a limited extent repulsive force field around each such object. This will create a second desired velocity vector $\mathbf{v}_{new}^{col\_avoid}$, also given by Newton's law. Interaction with other group members can be modeled by simultaneously applying different steering behaviors resulting in several additional desired velocity vectors $\mathbf{v}_{new}^{steer}$. Moving away from neighbors to avoid crowding, steering towards flock mates to ensure flock cohesion and adjusting a boid's speed to align with average heading of neighbors are most common. Finally, some additional desired velocity vectors $\mathbf{v}_{new}^{goal}$ are usually applied to achieve needed global goals. These can be vectors along some path in space, following some specific designated leader of the flock, or simply representing migratory urge of a flock member.

Once all $\mathbf{v}_{new}$ are determined, the final desired vector is negotiated based on priorities among them. Collision avoidance and velocity matching typically have higher priority. Instead of simple averaging of desired velocity vectors which can lead to cancellation of urges and unnatural "moving nowhere" behavior, an acceleration allocation strategy is used. Some fixed total amount of acceleration is made available for a boid and fractions of it are being given to each urge in order of priority. If the total available acceleration runs out, some lower priority urges will have less effect on the motion or be completely ignored. The hope is that once the currently most important task (collision avoidance in most situations) is accomplished, other tasks can be taken care of in near future. It is also important to respect some physical limitations of real objects, for example, clamping too high accelerations or speeds to some realistic values. Depending on the internal complexity of the flock member, the final stage of animation might be to turn the negotiated velocity vector into a specific set of parameters (bird's wing positions, orientation of plane model in space, leg skeleton bone configuration) used to control a boid's motion. A diagram of a system implementing flocking is shown on Figure 16.28 (right).

A much simpler, but still very useful, version of group control is implemented by *particle systems* (Reeves, 1983). The number of particles in a system is typically much larger than number of boids in a flock and can be in the tens or hundreds of thousands, or even more. Moreover, the exact number of particles can fluctuate during animation with new particles being born and some of the old ones destroyed at each step. Particles are typically completely independent from each other, ignoring one's neighbors and interacting with the environment only by experiencing external forces and collisions with objects, *not* through collision avoidance as was the case for flocks. At each step during animation, the system first creates new particles with some initial parameters, terminates old ones, and then computes necessary forces and updates velocities and positions of the remaining particles according to Newton's law.

All parameters of a particle system (number of particles, particle life span, initial velocity, and location of a particle, etc.) are usually under the direct control of the animator. Prime applications of particle systems include modeling fireworks, explosions, spraying liquids, smoke and fire, or other fuzzy objects and phenomena with no sharp boundaries. To achieve a realistic appearance, it is important to introduce some randomness to all parameters, for example, having a random number of particles born (and destroyed) at each step with their velocities generated according to some distribution. In addition to setting appropriate initial

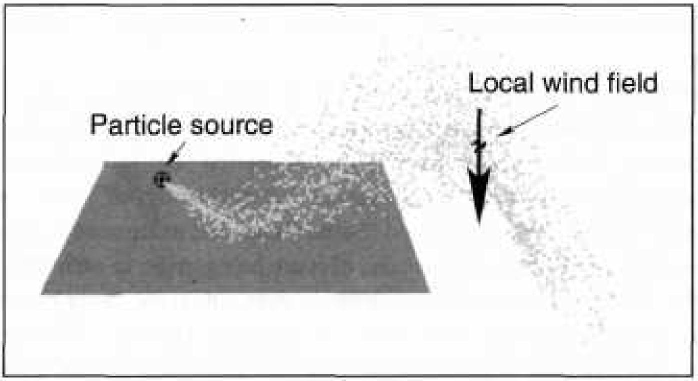


**Figure 16.29.** After being emitted by a directional source, particles collide with an object and then are blown down by a local wind field once they clear the obstacle.

parameters, controlling the motion of a particle system is commonly done by creating a specific force pattern in space—blowing a particle in a new direction once it reaches some specific location or adding a center of attraction, for example. One should remember that with all their advantages, simplicity of implementation and ease of control being the prime ones, particle systems typically do not provide the level of realism characteristic of true physics-based simulation of the same phenomena.

## 16.8 Notes

In this chapter we have concentrated on techniques used in 3D animation. There also exist a rich set of algorithms to help with 2D animation production and post-processing of images created by computer graphics rendering systems. These include techniques for cleaning up scanned-in artist drawings, feature extraction, automatic 2D in-betweening, colorization, image warping, enhancement and compositing, and many others.

One of the most significant developments in the area of computer animation has been the increasing power and availability of sophisticated animation systems. While different in their specific set of features, internal structure, details of user interface, and price, most such systems include extensive support not only for animation, but also for modeling and rendering turning them into complete production platforms. It is also common to use these systems to create still images. For example, many images for figures in this section were produced using Maya software generously donated by Alias.

Large-scale animation production is an extremely complex process which typically involves a combined effort by dozens of people with different backgrounds spread across many departments or even companies. To better coordinate this activity, a certain production pipeline is established which starts with a story and character sketches, proceeds to record necessary sound, build models, and rig characters for animation. Once actual animation commences, it is common to go back and revise the original designs, models, and rigs to fix any discovered motion and appearance problems. Setting up lighting and material properties is then necessary, after which it is possible to start rendering. In most sufficiently complex projects, extensive postprocessing and compositing stages bring together images from different sources and finalize the product.

We conclude this chapter by reminding the reader that in the field of computer animation any technical sophistication is secondary to a good story, expressive characters, and other artistic factors, most of which are hard or simply impossible to quantify. It is safe to say that Snow White and her seven dwarfs will always share the screen with green ogres and donkeys, and most of the audience will be much more interested in the characters and the story rather than in which, if any, computers (and in what exact way) helped to create either of them.

Peter Willemsen

17

# Using Graphics Hardware

Throughout most of this book, the focus has been on the fundamentals underlying computer graphics rather than on implementation details. This chapter takes a slightly different route and blends the details of using graphics hardware with the practical issues associated with programming that hardware.

This chapter, however, is not written to teach you OpenGL,$^{TM}$ other graphics APIs, or even the nitty gritty specifics of graphics hardware programming. The purpose of this chapter is to introduce the basic concepts and thought processes that are necessary when writing programs that use graphics hardware.

## 17.1 What is Graphics Hardware

*Graphics hardware* describes the hardware components necessary to quickly render 3D objects as pixels on your computer's screen using specialized rasterization-based hardware architectures. The use of this term is meant to elicit a sense of the physical components necessary for performing these computations. In other words, we're talking about the chipsets, transistors, buses, and processors found on many current video cards. As we will see in this chapter, current graphics hardware is very good at processing descriptions of 3D objects and transforming them into the colored pixels that fill your monitor.

One thing has been certain with graphics hardware: it changes very *quickly* with new extensions and features being added continually! One explanation for the fast pace is the video game industry and its economic momentum. Essentially

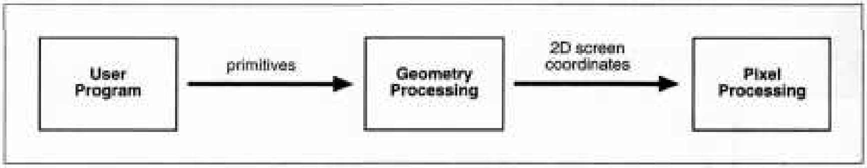


**Figure 17.1.** The basic graphics hardware pipeline consists of stages that transform 3D data into 2D screen objects ready for rasterizing and coloring by the pixel processing stages.

what this means is that each new graphics card provides better performance and processing capabilities. As a result, graphics hardware is being used for tasks that support a much richer use of 3D graphics. For instance, researchers are performing computation on graphics hardware to perform ray-tracing (Purcell, Buck, Mark, & Hanrahan, 2002) and even solve the Navier-Stokes equations to simulate fluid flow (Harris, 2004).

**Real-Time Graphics**: By real-time graphics, we generally mean that the graphics-related computations are being carried out fast enough that the results can be viewed immediately. Being able to conduct operations at 60Hz is considered real time. Once the time to refresh the display (*frame rate*) drops below 15Hz, the speed is considered more interactive than it is real-time, but this distinction is not critical. Because the computations need to be fast, the equations used to render the graphics are often approximations to what could be done if more time were available.

Most graphics hardware has been built to perform a set of fixed operations organized as a pipeline designed to push vertices and pixels through different stages. The fixed functionality of the pipeline ensures that basic coloring, lighting, and texturing can occur very quickly—often referred to as *real-time graphics*.

Figure 17.1 illustrates the real-time graphics pipeline. The important things to note about the pipeline follow:

- The user program, or application, supplies the data to the graphics hardware in the form of *primitives*, such as points, lines, or polygons describing the 3D geometry. Images or bitmaps are also supplied for use in texturing surfaces.

- Geometric primitives are processed on a per-vertex basis and are transformed from 3D coordinates to 2D screen triangles.

- Screen objects are passed to the pixel processors, rasterized, and then colored on a per-pixel basis before being output to the frame buffer, and eventually to the monitor.

## 17.2 Describing Geometry for the Hardware

As a graphics programmer, you need to be concerned with how the data associated with your 3D objects is transferred onto the memory cache of the graphics hardware. Unfortunately (or maybe fortunately), as a programmer you don't have complete control over this process. There are a variety of ways to place your data on the graphics hardware, and each has its own advantages which will be discussed in this section. Any of the APIs you might use to program your video card will provide different methods to load data onto the graphics hardware memory. The examples that follow are presented in pseudocode that is based loosely on the C function syntax of OpenGL,™ but semantically the examples should be applicable to other graphics APIs.

Most graphics hardware work with specific sets of geometric primitives. The primitive types leverage primitive complexity for processing speed on the graphics hardware. Simpler primitives can be processed very fast. The caveat is that the primitive types need to be general purpose so as to model a wide range of geometry from very simple to very complex. On typical graphics hardware, the primitive types are limited to one or more of the following:

- **Points:** single vertices used to represent points or particle systems;
- **Lines:** pairs of vertices used to represent lines, silhouettes, or edge-highlighting;
- **Polygons:** (e.g., triangles, triangle strips, indexed triangles, indexed triangle strips, quadrilaterals, general convex polygons, etc.), used for describing triangle meshes, geometric surfaces, and other solid objects, such as spheres, cones, cubes, or cylinders.

These three primitives form the basic building blocks for most geometry you will define. (An example of a triangle mesh is shown in Figure 17.2.) Using these primitives, you can build descriptions of your geometry using one of the graphics APIs and send the geometry to the graphics hardware for rendering. For instance,

**Primitives:**The three primitives (points, lines, and polygons) are the only primitives available! Even when creating spline-based surfaces, such as NURBs, the surfaces are tessellated into triangle primitives by the graphics hardware.

**Point Rendering:** Point and line primitives may initially appear to be limited in use, but researchers have used points to render very complex geometry (Rusinkiewicz & Levoy, 2000; Dachsbacher, Vogelgsang, & Stamminger, 2003).

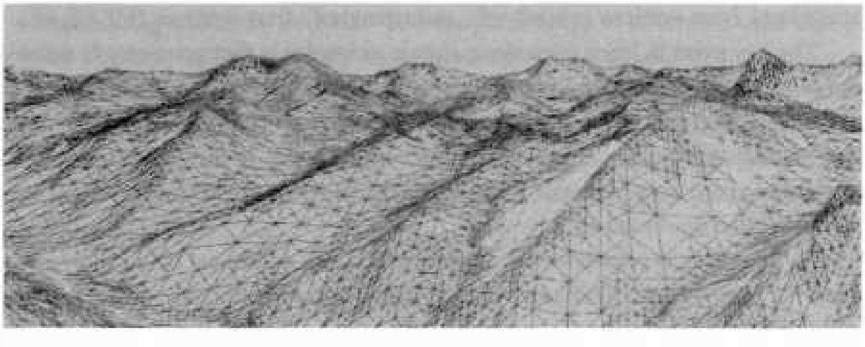


**Figure 17.2.** How your geometry is organized will affect the performance of your application. This wireframe depiction of the Little Cottonwood Canyon terrain dataset shows tens of thousands of triangles organized in a triangle mesh running at real-time rates. *The image is rendered using the VTerrain Project terrain system courtesy of Ben Discoe.*

to transfer the description of a line to the graphics hardware, we might use the following:

```
beginLine();
  vertex( x1, y1, z1 );
  vertex( x2, y2, z2 );
endLine();
```

In this example, two things occur. First, one of the primitive types is declared and made active by the `beginLine()` function call. The line primitive is then made inactive by the `endLine()` function call. Second, all vertices declared between these two functions are copied directly to the graphics card for processing with the `vertex` function calls.

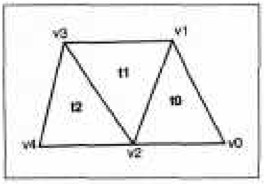


**Figure 17.3.** A triangle strip composed of five vertices defining three triangles.

A second example creates a set of triangles grouped together in a strip (refer to Figure 17.3); we could use the following code:

```
beginTriangleStrip();
  vertex( x0, y0, z0 );
  vertex( x1, y1, z1 );
  vertex( x2, y2, z2 );
  vertex( x3, y3, z3 );
  vertex( x4, y4, z4 );
endTriangleStrip();
```

In this example, the primitive type, `TriangleStrip`, is made active and the set of vertices that define the triangle strip are copied to the graphics card memory for processing. Note that ordering does matter when describing geometry. In the triangle strip example, connectivity between adjacent triangles is embedded within the ordering of the vertices. Triangle $t0$ is constructed from vertices $(v0, v1, v2)$, triangle $t1$ from vertices $(v1, v3, v2)$, and triangle $t2$ from vertices $(v2, v3, v4)$.

The key point to learn from these simple examples is that geometry is defined for rendering on the graphics hardware using a primitive type along with a set of vertices. The previous examples are simple and push the vertices directly onto the graphics hardware. However, in practice, you will need to make conscious decisions about how you will push your data to the graphics hardware. These issues will be discussed shortly.

As geometry is passed to the graphics hardware, additional data can be specified for each vertex. This extra data is useful for defining *state* attributes, that might represent the color of the vertex, the normal direction at the vertex, texture coordinates at the vertex, or other per-vertex data. For instance, to set the color and normal state parameters at each vertex of a triangle strip, we might use the following code:

```
beginTriangleStrip();
  color( r0, g0, b0 ); normal( n0x, n0y, n0z );
  vertex( x0, y0, z0 );
  color( r1, g1, b1 ); normal( n1x, n1y, n1z );
  vertex( x1, y1, z1 );
  color( r2, g2, b2 ); normal( n2x, n2y, n2z );
  vertex( x2, y2, z2 );
  color( r3, g3, b3 ); normal( n3x, n3y, n3z );
  vertex( x3, y3, z3 );
  color( r4, g4, b4 ); normal( n4x, n4y, n4z );
  vertex( x4, y4, z4 );
endTriangleStrip();
```

Here, the color and normal direction at each vertex are specified just prior to the vertex being defined. Each vertex in this example has a unique color and normal direction. The `color` function sets the active color state using a RGB 3-tuple. The normal direction state at each vertex is set by the `normal` function. Both the `color` and `normal` function affect the current rendering state on the graphics hardware. Any vertices defined after these state attributes are set will be bound with those state attributes.

This is a good moment to mention that the graphics hardware maintains a fairly elaborate set of state parameters that determine how vertices and other components are rendered. Some state is bound to vertices, such as color, normal direction, and texture coordinates, while another state may affect pixel level rendering. The *graphics state* at any particular moment describes a large set of internal hardware parameters. This aspect of graphics hardware is important to consider when you write 3D applications. As you might suspect, making frequent changes to the graphics state affects performance at least to some extent. However, attempting to minimize graphics state changes is only one of many areas where thoughtful programming should be applied. You should attempt to minimize state changes when you can, but it is unlikely that you can group all of your geometry to completely reduce state context switches. One data structure that can help minimize state changes, especially on static scenes, is the scene graph data structure. Prior to rendering any geometry, the scene graph can re-organize the geometry and associated graphics state in an attempt to minimize state changes. Scene graphs are described in Chapter 13.

```
color( r, g, b );
normal( nx, ny, nz );
beginTriangleStrip();
  vertex( x0, y0, z0 );
  vertex( x1, y1, z1 );
  vertex( x2, y2, z2 );
```

```
    vertex( x3, y3, z3 );
    vertex( x4, y4, z4 );
  endTriangleStrip();
```

All vertices in this `TriangleStrip` have the same color and normal direction, so these state parameters can be set prior to defining the vertices. This minimizes both function call overhead and changes to the internal graphics state.

Many things can affect the performance of a graphics program, but one of the potentially large contributors to performance (or lack thereof) is how your geometry is organized and whether it is stored in the memory cache of the graphics card. In the pseudocode examples provided so far, geometry has been pushed onto the graphics hardware in what is often called *immediate mode* rendering. As vertices are defined, they are sent directly to the graphics hardware. The primary disadvantage of immediate mode rendering is that the geometry is sent to the graphics hardware each iteration of your application. If your geometry is static (i.e., it doesn't change), then there is no real need to resend the data each time you redraw a frame. In these and other circumstances, it is more desirable to store the geometry in the graphics card's memory.

The graphics hardware in your computer is connected to the rest of the system via a data bus, such as the PCI, AGP, or PCI-Express buses. When you send data to the graphics hardware, it is sent by the CPU on your machine across one of these buses, eventually being stored in the memory on your graphics hardware. If you have very large triangle meshes representing complex geometry, passing all this data across the bus can end up resulting in a large hit to performance. This is especially true if the geometry is being rendered in immediate mode, as the previous examples have illustrated.

There are various ways to organize geometry; some can help reduce the overall bandwidth needed for transmitting the geometry across the graphics bus. Some possible organization approaches include:

- **Triangles**: triangles are specified with three vertices. A triangle mesh created in this manner requires that each triangle in the mesh be defined separately with many vertices potentially duplicated. For a triangle mesh containing $m$ triangles, $3m$ vertices will be sent to the graphics hardware.

- **Triangle strips**: triangles are organized in strips; the first three vertices specify the first triangle in the strip and each additional vertex adds a triangle. If you create a triangle mesh with $m$ triangles organized as a single triangle strip, you send three vertices to the graphics hardware for the first triangle followed by a single vertex for each additional triangle in the strip for a total of $m + 2$ vertices.

- **Indexed triangles**: triangle vertices are arranged as an array of vertices with a separate array defining the triangles using indices into the vertex array. Vertex arrays are sent to the graphics card with very few function calls.

- **Indexed triangle strips**: similar to indexed triangles, triangle vertices are stored in a vertex array. However, triangles are organized in strips with the index array defining the strip layout. This is the most compact of the organizational structures for defining triangle meshes as it combines the benefits of triangles strips with the compactness of vertex arrays.

Of the different organizational structures, the use of vertex arrays, either through indexed triangles or indexed triangle strips, provides a good option for increasing the performance of your application. The tight encapsulation of the organization means that many fewer function calls need to be made as well. Once the vertices and indices are stored in an array, only a few function calls need to be made to transfer the data to the graphics hardware, whereas with the pseudocode examples illustrated previously, a function is called for each vertex.

At this point, you may be wondering how the graphics state such as colors, normals, or texture coordinates are defined when vertex arrays are used. In the immediate-mode rendering examples earlier in the chapter, interleaving the graphics state with the associated vertices is obvious based on the order of the function calls. When vertex arrays are used, graphics state can either be interleaved in the vertex array or specified in separate arrays that are passed to the graphics hardware.

Even if the geometry is organized efficiently when it is sent to the graphics hardware, you can achieve higher performance gains if you can store your geometry in the graphics hardware's memory for the duration of your application. A somewhat unfortunate fact about current graphics hardware is that many of the specifications describing the layout of the graphics hardware memory and cache structure are often not widely publicized. Fortunately though, there are ways using graphics APIs that allow programmers to place geometry into the graphics hardware memory resulting in applications that run faster.

Two commonly used methods to store geometry and graphics state in the graphics hardware cache involve creating *display lists* or *vertex buffer objects*.

Display lists compile a compact list representation of the geometry and the state associated with the geometry and store the list in the memory on the graphics hardware. The benefits of display lists are that they are general purpose and good at storing a static geometric representation plus associated graphics state on the hardware. They do not work well at all for continuously changing geometry and graphics state, since the display list must be recompiled and then stored *again* in the graphics hardware memory for every iteration in which the display list changes.

```
displayID = createDisplayList();
color( r, g, b );
normal( nx, ny, nz );
beginTriangleStrip();
  vertex( x0, y0, z0 );
  vertex( x1, y1, z1 );
  ...
  vertex( xN, yN, zN );
endTriangleStrip();
endDisplayList();
```

In the above example, a display list is created that contains the definition of a triangle strip with its associated color and normal information. The commands between the `createDisplayList` and `endDisplayList` function calls provide the elements that define the display list. Display lists are most often created during an initialization phase of an application. After the display list is created, it is stored in the memory of the graphics hardware and can be referenced for later use by the identifier assigned to the list.

```
// draw the display list created earlier
drawDisplayList(displayID);
```

When it is time to draw the contents of the display list, a single function call will instruct the graphics hardware to access the memory indexed through the display list identifier and display the contents.

**Optimal Organization**: Much research effort has gone into looking at ways to optimize triangle meshes for maximum performance on graphics hardware. A good place to start reading if you want to delve further into understanding how triangle mesh organization affects performance is the SIGGRAPH 1999 paper on the optimization of mesh locality (Hoppe, 1999).

A second method to store geometry on the graphics hardware for the duration of your application is through vertex buffer objects (VBOs). VBOs are specialized buffers that reside in high-performance memory on the graphics hardware and store vertex arrays and associated graphics state. They can also provide a mapping from your application to the memory on the graphics hardware to allow for fast access and updating to the contents of the VBO.

The chief advantage of VBOs is that they provide a mapping into the graphics hardware memory. With VBOs, geometry can be modified during an application with a minimal loss of performance as compared with using immediate mode rendering or display lists. This is extremely useful if portions of your geometry change during each iteration of your application or if the indices used to organize your geometry change.

VBOs are created in much the same way indexed triangles and indexed triangle strips are built. A buffer object is first created on the graphics card to make room for the vertex array containing the vertices of the triangle mesh. Next, the vertex array and index array are copied over to the graphics hardware. When it is time to render the geometry, the vertex buffer object identifier can be used to instruct the graphics hardware to draw your geometry. If you are already using vertex arrays in your application, modifying your code to use VBOs should likely require a minimal change.

## 17.3 Processing Geometry into Pixels

After the geometry has been placed in the graphics hardware memory, each vertex must be lit as well as transformed into screen coordinates during the geometry processing stage. In the fixed-function graphics pipeline illustrated in Figure 17.1, vertices are transformed from a model coordinate system to a screen coordinate frame of reference. This process and the matrices involved are described in Chapters 7 and 12. The modelview and projection matrices needed for this transformation are defined using functions provided with the graphics API you decide to use.

Lighting is calculated on a per-vertex basis. Depending on the global shading parameters, the triangle face will either have a flat-shaded look or the face color will be diffusely shaded (Gouraud shading) by linearly interpolating the color at each triangle vertex across the face of the triangle. The latter method produces a much smoother appearance. The color at each vertex is computed based on the assigned material properties, the lights in the scene, and various lighting parameters.

The lighting model in the fixed-function graphics pipeline is good for fast lighting of vertices; we make a tradeoff for increased speed over accurate illumination. As a result, Phong shaded surfaces are not supported with this fixed-function framework.

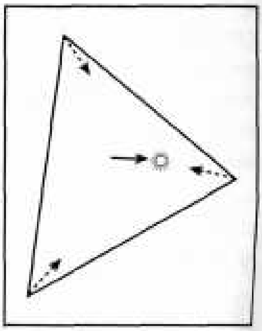


Figure 17.4. The distance to the light source is small relative to the size of the triangle.

In particular, the diffuse shading algorithm built into the graphics hardware often fails to compute the appropriate illumination since the lighting is only being calculated at each vertex. For example, when the distance to the light source is small, as compared with the size of the face being shaded, the illumination on the face will be incorrect. Figure 17.4 illustrates this situation. The center of the triangle will not be illuminated brightly despite being very close to the light source, since the lighting on the vertices, which are far from the light source, are used to interpolate the shading across the face.

With the fixed-function pipeline, this issue can only be remedied by increasing the tessellation of the geometry. This solution works but is of limited use in real-time graphics as the added geometry required for more accurate illumination can result in slower rendering.

However, with current hardware, the problem of obtaining better approximations for illumination can be solved without necessarily increasing the geometric complexity of the objects. The solution involves replacing the fixed-function routines embedded within the graphics hardware with your own programs. These small programs run on the graphics hardware and perform a part of the geometry processing and pixel-processing stages of the graphics pipeline.

### 17.3.1 Programming the Pipeline

Fairly recent changes to the organization of consumer graphics hardware has generated a substantial buzz from game developers, graphics researchers, and many others. It is quite likely that you have heard about *GPU programming*, *graphics hardware programming*, or even *shader programming*. These terms and the changes in consumer hardware that have spawned them primarily have to do with how the graphics hardware rendering pipeline can now be programmed.

**Definition**: *Fragment* is a term that describes the information associated with a pixel prior to being processed by the graphics hardware. This definition includes much of the data that might be used to calculate the color of the pixel, such as the pixel's scene depth, texture coordinates, or stencil information.

Specifically, the changes have opened up two specific aspects of the graphics hardware pipeline. Programmers now have the ability to modify how the hardware processes vertices and shades pixels by writing *vertex shaders* and *fragment shaders* (also sometimes referred to as *vertex programs* or *fragment programs*). Vertex shaders are programs that perform the vertex and normal transformations, texture coordinate generation, and per-vertex lighting computations normally computed in the geometry processing stage. Fragment shaders are programs that perform the computations in the pixel processing stage of the graphics pipeline and determine exactly how each pixel is shaded, how textures are applied, and if a pixel should be drawn or not. These small shader programs are sent to the graphics hardware from the user program (see Figure 17.5), but they are executed on the graphics hardware. What this programmability means for

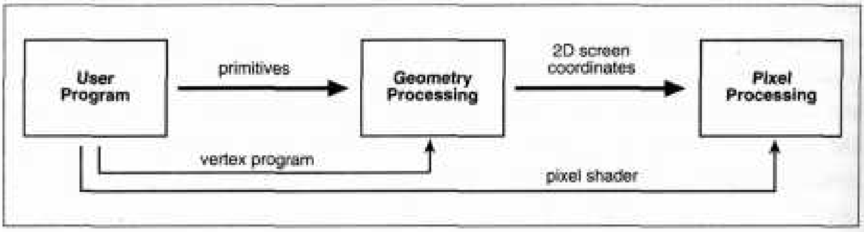


**Figure 17.5.** The programmable graphics hardware pipeline. The user program supplies primitives, vertex programs, and fragment programs to the hardware.

you is that you essentially have a multi-processor machine. This turns out to be a good way to think about your graphics hardware, since it means that you may be able to use the graphics hardware processor to relieve the load on the CPU in some of your applications. The graphics hardware processors are often referred to as *GPUs*. GPU stands for Graphics Processing Unit and highlights the fact that graphics hardware components now contain a separate processor dedicated to graphics-related computations.

Interestingly, modern GPUs contain more transistors than modern CPUs. For the time being, GPUs are utilizing most of these transistors for computations and less for memory or cache management operations.

However, this will not always be the case as graphics hardware continues to advance. And just because the computations are geared towards 3D graphics, it does not mean that you cannot perform computations unrelated to computer graphics on the GPU. The manner in which the GPU is programmed is different from your general purpose CPU and will require a slightly modified way of thinking about how to solve problems and program the graphics hardware.

The GPU is a stream processor that excels at 3D vector operations such as vector multiplication, vector addition, dot products, and other operations necessary for basic lighting of surfaces and texture mapping. As stream processors, both the vertex and fragment processing components include the ability to process multiple primitives at the same time. In this regard, the GPU acts as a SIMD (Single Instruction, Multiple Data) processor, and in certain hardware implementations of the fragment processor, up to 16 pixels can be processed at a time. When you write programs for these processing components, it will be helpful, at least conceptually, to think of the computations being performed concurrently on your data. In other words, the vertex shader program will run for all vertices at the same time. The vertex computations will then be followed by a stage in which your fragment shader program will execute simultaneously on all fragments. It is important to note that while the computations on vertices or fragments occur concurrently, the staging of the pipeline components still occur in the same order.

The manner in which vertex and fragment shaders work is simple. You write a vertex shader program and a fragment shader program and send it to the graphics hardware. These programs can be used on specific geometry, and when your geometry is processed, the vertex shader is used to transform and light the vertices, while the fragment shader performs the final shading of the geometry on a per-pixel basis. Just as you can texture map different images onto different pieces of geometry, you can also write different shader programs to act upon different objects in your application. Shader programs are a part of the graphics state so you do need to be concerned with how your shader programs might get swapped in and out based on the geometry being rendered.

**Historical**: Programming the pipeline is not entirely new. One of the first introductions of a graphics hardware architecture designed for programming flexibility were the PixelFlow architectures and shading languages from UNC (Molnar, Eyles, & Poulton, 1992; Lastra, Molnar, Olano, & Wang, 1995; Olano & Lastra, 1998). Additional efforts to provide custom shading techniques have included shade trees (Cook, 1984), RenderMan (Pixar, 2000), accelerated multi-pass rendering using OpenGL$^{TM}$ (Peercy, Olano, Airey, & Ungar, 2000), and other real-time shading languages (Proudfoot, Mark, Tzvetkov, & Hanrahan, 2001; McCool, Du Toit, Popa, Chan, & Moule, 2004).

The details tend to be a bit more complicated, however. Vertex shaders usually perform two basic actions: set the color at the vertex and transform the vertex into screen coordinates by multiplying the vertex by the modelview and projection matrices. The perspective divide and clipping steps are not performed in a vertex program. Vertex shaders are also often used to set the stage for a fragment shader. In particular, you may have vertex attributes, such as texture coordinates or other application- dependent data, that the vertex shader calculates or modifies and then sends to the fragment processing stage for use in your fragment shader. It may seem strange at first, but vertex shaders can be used to manipulate the positions of the vertices. This is often useful for generating simulated ocean wave motion entirely on the GPU.

In a fragment shader, it is required that the program outputs the fragment color. This may involve looking up texture values and combining them in some manner with values obtained by performing a lighting calculation at each pixel; or, it may involve killing the fragment from being drawn entirely. Because operations in the fragment shader operate at the fragment level, the real power of the programmable graphics hardware is in the fragment shader. This added processing power represents one of the key differences between the fixed function pipeline and the programmable pipeline. In the fixed pipeline, fragment processing used illumination values interpolated between the vertices of the triangle to compute the fragment color. With the programmable pipeline, the color at each fragment can be computed independently. For instance, in the example situation posed in Figure 17.4, Gouraud shading of a triangle face fails to produce a reasonable solution because lighting only occurs at the vertices which are farther away from the light than the center of the triangle. In a fragment shader, the lighting equation can be evaluated at each fragment, rather than at each vertex, resulting in a more accurate rendering of the face.

### 17.3.2 Basic Execution Model

When writing vertex or fragment shaders, there are a few important things to understand in terms of how vertex and fragment programs execute and access data on the GPU. Because these programs run entirely on the GPU, the first details you will need to figure out are which data your shaders will use and how to get that data to them. There are several characteristics associated with the data types used in shader programs. The following terms, which come primarily from the OpenGL$^{TM}$ Shading Language framework, are used to describe the conceptual aspects of these data characteristics. The concepts are the same across different shading language frameworks. In the shaders you write, variables are characterized using one of the following terms:

- **attributes**: Attribute variables represent data that changes frequently, often on a per-vertex basis. Attribute variables are often tied to the changing graphics state associated with each vertex. For instance, normal vectors or texture coordinates are considered to be attribute data since they are part of the graphics state associated with each vertex.

- **uniforms**: Uniform variables represent data that cannot change during the execution of a shader program. However, uniform variables can be modified by your application between executions of a shader. This provides another way for your application to communicate data to a shader. Uniform data often represent the graphics state associated with an application. For instance, the modelview and projection matrices can be accessed through uniform variables. Information about light sources in your application can also be obtained through uniform variables. In these examples, the data does not change while the shader is executing, but could (e.g., the light could move) prior to the next iteration of the application.

- **varying**: Varying data is used to pass data between a vertex shader and a fragment shader. The reason the data is considered *varying* is because it is written by vertex shaders on a per-vertex basis, but read by fragment shaders as value interpolated across the face of the primitive between neighboring vertices.

Variables defined using one of these three characteristics can either be built-in variables or user-defined variables. In addition to accessing the built-in graphics state, attribute and uniform variables are one of the ways to communicate user-defined data to your vertex and fragment programs. Varying data is the only means to pass data from a vertex shader to a fragment shader. Figure 17.6 illustrates the basic execution of the vertex and fragment processors in terms of the inputs and outputs used by the shaders.

Another way to pass data to vertex and fragment shaders is by using texture maps as sources and sinks of data. This may come as a surprise if you have been thinking of texture maps solely as images that are applied to the outside surface of geometry. The reason texture maps are important is because they give you access to the memory on the graphics hardware. When you write applications that run on the CPU, you control the memory your application requires and have direct access to it when necessary. On graphics hardware, memory is not accessed in the same manner. In fact, you are not directly able to allocate and deallocate general purpose memory chunks, and this particular aspect usually requires a slight change in thinking.

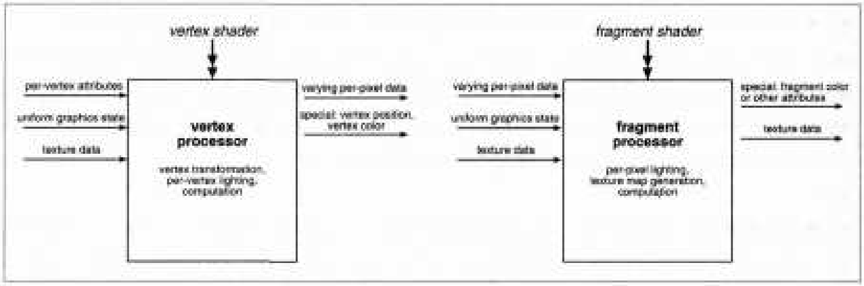


**Figure 17.6.** The execution model for shader programs. Input, such as per-vertex attributes, graphics state-related uniform variables, varying data, and texture maps are provided to vertex and fragment programs within the shader processor. Shaders output special variables used in later parts of the graphics pipeline.

Texture maps on graphics hardware, however, can be created, deleted, and controlled through the graphics API you use. In other words, for general data used by your shader, you will create texture maps that contain that data and then use texture access functions to look up the data in the texture map. Technically, textures can be accessed by both vertex and fragment shaders. However, in practice, texture lookups from the vertex shader are not currently supported on all graphics cards. An example that utilizes a texture map as a data source is bump mapping. Bump mapping uses a normal map which defines how the normal vectors change across a triangle face. A bump mapping fragment shader would look up the normal vector in the normal map "texture data" and use it in the shading calculations at that particular fragment.

**Note:** The shader language examples used in this chapter are presented using GLSL (OpenGL™ Shading Language). This language was chosen since it is being developed by the OpenGL™ Architecture Review Board and will likely become a standard shading language for OpenGL™ with the release of OpenGL™ 2.0. As of this writing, GLSL can be used on most modern graphics cards with updated graphics hardware drivers.

You need to be concerned about the types of data you put into your texture maps. Not all numerical data types are well supported and only recently has graphics hardware included floating point textures with 16-bit components. Moreover, *none* of the computation being performed on your GPU is done with double-precision math! If numerical precision is important for your application, you will need to think through these issues very carefully to determine if using the graphics hardware for computation is useful.

So what do these shader programs look like? One way to write vertex and fragment shaders is through assembly language instructions. For instance, performing a matrix multiplication in shader assembly language looks something like this:

```
DP4 p[0].x, M[0], v[0];
DP4 p[0].y, M[1], v[0];
DP4 p[0].z, M[2], v[0];
DP4 p[0].w, M[3], v[0];
```

In this example, the DP4 instruction is a 4-component dot product function. It stores the result of the dot product in the first register and performs the dot product between the last two registers. In shader programming, registers hold 4-components corresponding to the $x$, $y$, $z$, and $w$ components of a homogeneous coordinate, or the $r$, $g$, $b$, and $a$ components of a RGBA tuple. So, in this example, a simple matrix multiplication,

$$\mathbf{p} = \mathbf{M}\mathbf{v}$$

is computed by four DP4 instructions. Each instruction computes one element of the final result.

Fortunately though, you are not forced to program in assembly language. The good news is that higher-level languages are available to write vertex and fragment shaders. NVIDIA's Cg, the OpenGL™ Shading Language (GLSL), and Microsoft's High Level Shading Language (HLSL) all provide similar interfaces to the programmable aspects of graphics hardware. Using the notation of GLSL, the same matrix multiplication performed above looks like this:

```
    p = M * v;
```

where $\mathbf{p}$ and $\mathbf{v}$ are vertex data types and $\mathbf{M}$ is a matrix data type. As evidenced here, one advantage of using a higher-level language over assembly language is that various data types are available to the programmer. In all of these languages, there are built-in data types for storing vectors and matrices, as well as arrays and constructs for creating structures. Many different functions are also built in to these languages to help compute trigonometric values (sin, cos, etc...), minimum and maximum values, exponential functions (log2, sqrt, pow, etc...), and other math or geometric-based functions.

### 17.3.3 Vertex Shader Example

Vertex shaders give you control over how your vertices are lit and transformed. They are also used to set the stage for fragment shaders. An interesting aspect to vertex shaders is that you still are able to use geometry-caching mechanisms, such as display lists or VBOs, and thus, benefit from their performance gains while using vertex shaders to do computation on the GPU. For instance, if the vertices represent particles and you can model the movement of the particles using a vertex shader, you have nearly eliminated the CPU from these computations. Any bottleneck in performance that may have occurred due to data being passed between the CPU and the GPU will be minimized. Prior to the introduction of vertex shaders, the computation of the particle movement would have been performed on the CPU and each vertex would have been re-sent to the graphics hardware on each iteration of your application. The ability to perform computations on the vertices already stored in the graphics hardware memory is a big performance win.

One of the simplest vertex shaders transforms a vertex into clip coordinates and assigns the front-facing color to the color attribute associated with the vertex.

```
void main(void)
{
    gl_Position = gl_ModelViewProjectionMatrix *
                      gl_Vertex;
    gl_FrontColor = gl_Color;
}
```

In this example, gl_ModelViewProjectionMatrix is a built-in uniform variable supplied by the GLSL run-time environment. The variables gl_Vertex and gl_Color are built-in vertex attributes; the special output variables, gl_Position and gl_FrontColor are used by the vertex shader to set the transformed position and the vertex color.

A more interesting vertex shader that implements the surface- shading equations developed in Chapter 9 illustrates the effect of per-vertex shading using the Phong shading algorithm.

```
void main(void)
{
    vec4 v = gl_ModelViewMatrix * gl_Vertex;
    vec3 n = normalize(gl_NormalMatrix * gl_Normal);
    vec3 l = normalize(gl_LightSource[0].position - v);
    vec3 h = normalize(l - normalize(v));

    float p = 16;
    vec4 cr = gl_FrontMaterial.diffuse;
    vec4 cl = gl_LightSource[0].diffuse;
    vec4 ca = vec4(0.2, 0.2, 0.2, 1.0);

    vec4 color;
    if (dot(h,n) > 0)
        color = cr * (ca + cl * max(0,dot(n,l))) +
                    cl * pow(dot(h,n), p);
    else
        color = cr * (ca + cl * max(0,dot(n,l)));

    gl_FrontColor = color;
    gl_Position = ftransform();
}
```

From the code presented in this shader, you should be able to gain a sense of shader programming and how it resembles C-style programming. Several things are happening with this shader. First, we create a set of variables to hold the vectors necessary for computing Phong shading: $\mathbf{v}, \mathbf{n}, \mathbf{l}$, and $\mathbf{h}$. Note that the computation in the vertex shader is performed in *eye-space*. This is done for a variety of reasons, but one reason is that the light-source positions accessible within a shader have already been transformed into the eye coordinate frame. When you create shaders, the coordinate system that you decide to use will likely depend on the types of computations being performed; this is an important factor to consider. Also, note the use of built-in functions and data structures in the example. In particular, there are several functions used in this shader: `normalize`, `dot`, `max`, `pow`, and `ftransform`. These functions are provided with the shader language. Additionally, the graphics state associated with materials and lighting can be accessed through built-in uniform variables: `gl_FrontMaterial` and `gl_LightSource[0]`. The diffuse component of the material and light is accessed through the `diffuse` member of these variables. The color at the vertex is computed using Equation 9.8 and then stored in the special output variable `gl_FrontColor`. The vertex position is transformed using the function

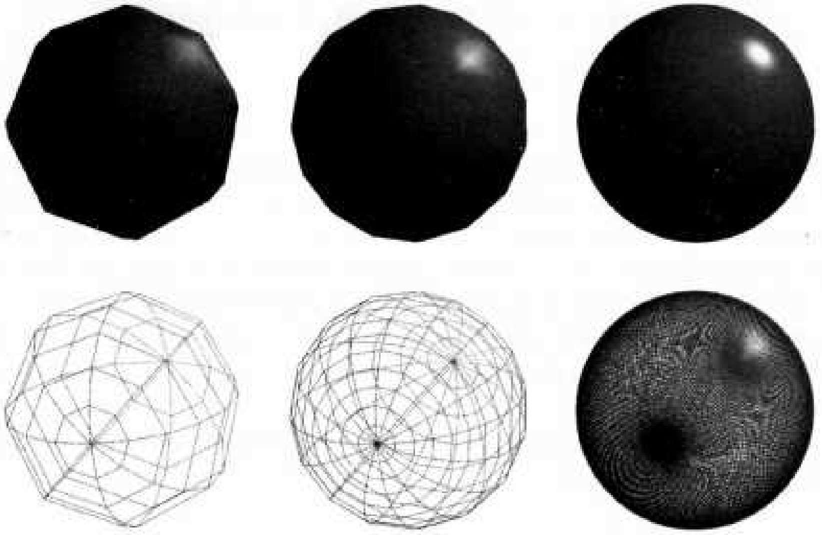


**Figure 17.7.** Each sphere is rendered using only a vertex shader that computes Phong shading. Because the computation is being performed on a per-vertex basis, the Phong highlight only begins to appear accurate after the amount of geometry used to model the sphere is increased drastically. (See also Plate VIII.)

`ftransform` which is a convenience function that performs the multiplication with the modelview and projection matrices. Figure 17.7 shows the results from running this vertex shader with differently tessellated spheres. Because the computations are performed on a per-vertex basis, a large amount of geometry is required to produce a Phong highlight on the sphere that appears correct.

### 17.3.4 Fragment Shader Example

Fragment shaders are written in a manner very similar to vertex shaders, and to emphasize this, Equation 9.8 from Chapter 9 will be implemented with a fragment shader. In order to do this, we first will need to write a vertex shader to set the stage for the fragment shader.

The vertex shader required for this example is fairly simple, but introduces the use of *varying* variables to communicate data to the fragment shader.

```
varying vec4 v;
varying vec3 n;

void main(void)
{
    v = gl_ModelViewMatrix * gl_Vertex;
    n = normalize(gl_NormalMatrix * gl_Normal);

    gl_Position = ftransform();
}
```

Recall that varying variables will be set on a per-vertex basis by a vertex shader, but when they are accessed in a fragment shader, the values will *vary* (i.e., be interpolated) across the triangle, or geometric primitive. In this case, the vertex position in eye-space **v** and the normal at the vertex **n** are calculated at each vertex. The final computation performed by the vertex shader is to transform the vertex into clip coordinates since the fragment shader will compute the lighting at each fragment. It is not necessary to set the front-facing color in this vertex shader.

The fragment shader program computes the lighting at each fragment using the Phong shading model.

```
varying vec4 v;
varying vec3 n;

void main(void)
{
    vec3 l = normalize(gl_LightSource[0].position - v);
    vec3 h = normalize(l - normalize(v));

    float p = 16;
    vec4 cr = gl_FrontMaterial.diffuse;
    vec4 cl = gl_LightSource[0].diffuse;
    vec4 ca = vec4(0.2, 0.2, 0.2, 1.0);

    vec4 color;
    if (dot(h,n) > 0)
       color = cr * (ca + cl * max(0,dot(n,l))) +
                  cl * pow(dot(h,n),p);
    else
       color = cr * (ca + cl * max(0,dot(n,l)));

    gl_FragColor = color;
}
```

The first thing you should notice is the similarity between the fragment shader code in this example and the vertex shader code presented in Section 17.3.3. The

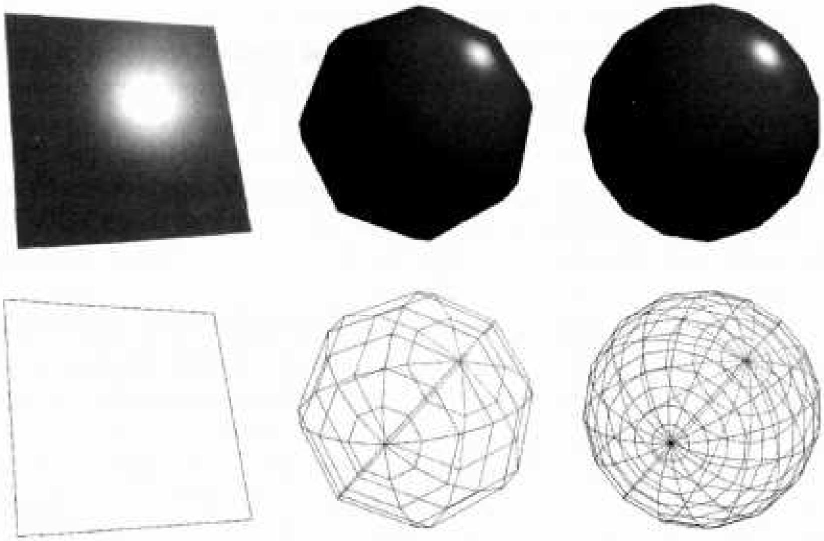


**Figure 17.8.** The results of running the fragment shader from Section 17.3.4. Note that the Phong highlight does appear on the left-most model which is represented by a single polygon. In fact, because lighting is calculated at the fragment, rather than at each vertex, the more coarsely tessellated sphere models also demonstrate appropriate Phong shading. (See also Plate IX.)

main difference is in the use of the varying variables, **v** and **n**. In the fragment shader, the view vectors and normal values are interpolated across the surface of the model between neighboring vertices. The results are shown in Figure 17.8. Immediately, you should notice the Phong highlight on the quadrilateral, which only contains four vertices. Because the shading is being calculated at the fragment level using the Phong equation with the interpolated (i.e., varying) data, more consistent and accurate Phong shading is produced with far less geometry.

### 17.3.5 General Purpose Computing on the GPU

After studying the vertex and fragment shader examples, you may be wondering if you can write programs to perform other types of computations on the GPU. Obviously, the answer is yes, as many problems can be coded to run on the GPU given the various languages available for programming on the GPU. However, a few facts are important to remember. Foremost, floating point math processing on graphics hardware is not currently double-precision. Secondly, you will likely need to transform your problem into a form that fits within a graphics-related framework. In other words, you will need to use the graphics APIs to set up the problem, use texture maps as data rather than traditional memory, and write vertex and fragment shaders to frame and solve your problem.

Having stated that, the GPU may still be an attractive platform for computation, since the ratio of transistors that are dedicated to performing computation is much higher on the GPU than it is on the CPU. In many cases, algorithms running on GPUs run faster than on a CPU. Furthermore, GPUs perform SIMD computation, which is especially true at the fragment-processing level. In fact, it can often help to think about the computation occurring on the fragment processor as a highly parallel version of a generic `foreach` construct, performing simultaneous operations on a set of elements.

There has been a large amount of investigation to perform General Purpose computation on GPUs, often referred to as GPGPU. Among other things, researchers are using the GPU as a means to simulate the dynamics of clouds (Harris, Baxter, Scheuermann, & Lastra, 2003), implement ray tracers (Purcell et al., 2002; Carr, Hall, & Hart, 2002), compute radiosity (Coombe, Harris, & Lastra, 2004), perform 3D segmentation using level sets (A. E. Lefohn, Kniss, Hansen, & Whitaker, 2003), or solve the Navier-Stokes equations (Harris, 2004).

General purpose computation is often performed on the GPU using multiple rendering "passes," and most computation is done using the fragment processor due to its highly data-parallel setup. Each pass, called a *kernel*, completes a portion of the computation. Kernels work on streams of data with several kernels strung together to form the overall computation. The first kernel completes the first part of the computation, the second kernel works on the first kernel's data, and so on, until the calculation is complete. In this style of programming, working with data and data structures on the GPU is different than conventional programming and does require a bit of thought. Fortunately, recent efforts are providing abstractions and information for creating efficient data structures for GPU programming (A. Lefohn, Kniss, & Owens, 2005).

Using the GPU for general purpose programming does require that you understand how to program the graphics hardware. For instance, most applications that perform GPGPU will render a simple quadrilateral, or sets of quadrilaterals, with vertex and fragment shaders operating on that geometry. The geometry doesn't have to be visible, or drawn to the screen, but it is necessary to allow the vertex and fragment operations to occur. This focus on graphics does make the learning curve for general purpose computing on this hardware an adventure. Fortunately, recent efforts are working to make the interface to the GPU more like traditional programming. The Brook for GPUs project (Buck et al., 2004) is a system that provides a C-like interface to afford stream computations on the GPU, which should allow more people to take advantage of the computational power on modern graphics hardware.

## Frequently Asked Questions

- How do I debug shader programs?

On most platforms, debugging both vertex shaders and fragment shaders is not simple. There is very little runtime support for debugging graphics applications in general, and even less available for runtime debugging of shader programs. However, this is starting to change. In the latest versions of Mac OS X, Linux, and Windows, support for shader programming is incorporated. A good solution for debugging shader programs is to use one of the shader development tools available from various graphics hardware manufacturers.

## Notes

There are many good resources available to learn more about the technical details involved with programming graphics hardware. A good starting point might be the OpenGL$^{TM}$ Programming Guide (Shreiner et al., 2004). The OpenGL$^{TM}$ Shading Language (Rost, 2004) and The Cg Tutorial (Fernando & Killgard, 2003) provide details on how to program using a shading language. More advanced technical information and examples for programming the vertex and fragment processors can be found in the GPU Gems series of books (Fernando, 2004; Pharr & Fernando, 2005). A source of information for learning more about general purpose computation on GPUs (GPGPU) can be found on the GPGPU.org web site (http://www.gpgpu.org).

## Exercises

1. How fast is the GPU as compared to performing the operations on the CPU? Write a program in which you can parameterize how much data is processed on the GPU, ranging from no computation using a shader program to all of the computation being performed using a shader program. How does the performance of you application change when the computation is being performed solely on the GPU?

2. Are there sizes of triangle strip lengths that work better than others? Try to determine the maximum size of a triangle strip that maximizes performance. What does this tell you about the memory, or cache structure, on the graphics hardware?

Kelvin Sung

18

# Building Interactive Graphics Applications

While most of the other chapters in this book discuss the fundamental algorithms in the field of computer graphics, this chapter treats the integration of these algorithms into applications. This is an important topic since the knowledge of fundamental graphics algorithms does not always easily lead to an understanding of the best practices in implementing these algorithms in real applications.

We start with a simple example: a program that allows the user to simulate the shooting of a ball (under the influence of gravity). The user can specify initial velocity, create balls of different sizes, shoot the ball, and examine the parabolic free fall of the ball. Some fundamental concepts we will need include mesh structure for the representation of the ball (sphere); texture mapping, lighting, and shading for the aesthetic appearance of the ball; transformations for the trajectories of the ball; and rasterization techniques for the generation of the images of the balls.

To implement the simple ball shooting program, one also needs knowledge of

- Graphical user interface (GUI) systems for efficient and effective user interaction;

- Software architecture and design patterns for crafting an implementation framework that is easy to maintain and expand;

- Application program interfaces (APIs) for choosing the appropriate support and avoiding a massive amount of unnecessary coding.

To gain an appreciation for these three important aspects of building the application, we will complete the following steps:

- analyze interactive applications;
- understand different programming models and recognize important functional components in these models;
- define the interaction of the components;
- design solution frameworks for integrating the components; and
- demonstrate example implementations based on different sets of existing APIs.

We will use the ball shooting program as our example and begin by refining the detailed specifications. For clarity, we avoid graphics-specific complexities in 3D space and confine our example to 2D space. Obviously, our simple program is neither sophisticated nor representative of real applications. However, with slightly refined specifications, this example contains all the essential components and behavioral characteristics of more complex real-world interactive systems.

We will continue to build complexity into our simple example, adding new concepts until we arrive at a software architecture framework that is suitable for building general interactive graphics applications. We will examine the validity of our results and discuss how the lessons learned from this simple example can be applied to other familiar real-world applications (e.g., PowerPoint, Maya, etc.).

## 18.1 The Ball Shooting Program

Our simple program has the following elements and behaviors:

- **The balls (objects):** The user can left-mouse-button-click and drag-out a new ball (circle) anywhere on the screen (see Figure 18.1). Dragging-out a ball includes:
  - (A): initial mouse-button-click position defines the center of the circle;
  - (B): mouse button down and moving the mouse is the dragging action;
  - (C): current mouse position while dragging allows us to define the radius and the initial velocity. The radius R (in pixel units) is the distance to the center defined in (A). The vector from the current position to the center is the initial velocity V (in units of pixel per second).

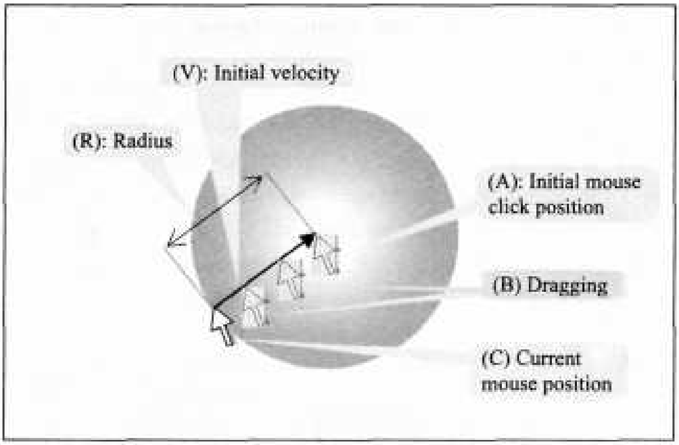


**Figure 18.1.** Dragging out a ball.

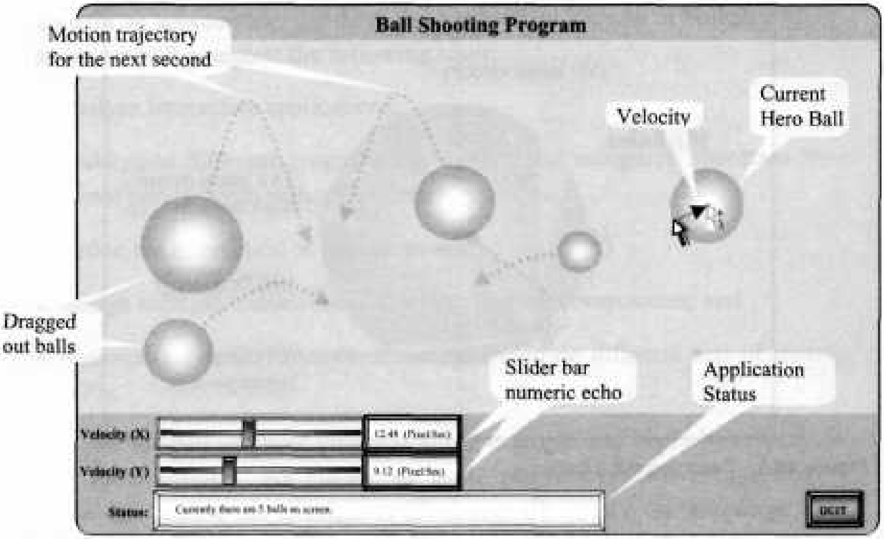
Once created, the ball will begin traveling with the defined initial velocity.

- **HeroBall (Hero/active object):** The user can also right-mouse-button-click to select a ball to be the current HeroBall. The HeroBall's velocity can be controlled by the slider bars (discussed below) where its velocity is displayed. (A newly created ball is by default the current HeroBall.) A right-mouse-button-click on unoccupied space indicates that no current HeroBall exists.

- **Velocity slider bars (GUI elements):** The user can monitor and control two slider bars ($x$- and $y$-directions with magnitudes) to change the velocity of the HeroBall. When there is no HeroBall, the slider bar values are undefined.

- **The simulation:**

  - **Ball traveling/collisions (object intrinsic behaviors):** A ball knows how to travel based on its current velocity and one ball can potentially collide with another. For simplicity, we will assume all balls have identical mass and all collisions are perfectly elastic.
  - **Gravity (external effects on objects):** The velocity of a ball is constantly changing due to the defined gravitational force.
  - **Status bar (application state echo):** The user can monitor the application state by examining the information in the status bar. In our application, the number of balls currently on the screen is updated in the status bar.



Currently there are 5 balls on screen.

**Figure 18.2.** The Ball Shooting program.

Our application starts with an empty screen. The user clicks and drags to create new balls with different radii and velocities. Once a ball travels off of the screen, it is removed. To avoid unnecessary details, we do not include the drawing of the motion trajectories or the velocity vector in our solutions. Notice that a slider bar communicates its current state to the user in two ways: the position of the slider knob and the numeric echo (see Figure 18.2).

We have now described the behavior of a simple interactive graphics application. In the rest of this chapter, we will learn the concepts that support the implementation of this type of application.

## 18.2 Programming Models

For many of us, when we were first introduced to computer programming, we learned that the program should always start and end with the *main*() function—when the *main*() function returns, all the work must have been completed and the program terminates. Since the overall control remains internal to the *main*() function during the entire life time of the program, the type of model for this approach to solving problems is called an *internal control model*, or *control-driven programming*. As we will see, an alternative paradigm, *event-driven programming* or an *external control model* approach, is the more appropriate way to design solutions to interactive programs.

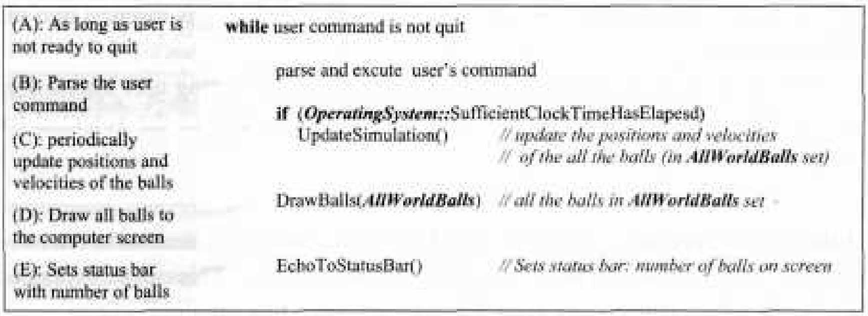
In this section, we will first formulate a solution to the 2D ball shooting program based on the, perhaps more familiar, control-driven programming model. We will then analyze the solution, identify shortcomings, and describe the motivation for the external control model or event-driven programming approach.

The pseudocode which follows is C++/Java-like. We assume typical functionality from the operating System (*OperatingSystem::*) and from a Graphical User Interface API (*GUISystem::*). The purpose of the pseudocode is to assist us in analyzing the foundation control structure (i.e., if/while/case) of the solution. For this reason, the details of application- and graphics-specific operations are intentionally glossed over. For example, the details of how to *UpdateSimulation*() is purposely omitted.

### 18.2.1 Control-Driven Programming

The main advantage of control-driven programming is that it is fairly straightforward to translate a verbal description of a solution to a program control structure. In this case, we verbalize our solution as follows:

> while the user does not want to quit (A);
>
> parse and execute the user's command (B);
>
> update the velocities and positions of the balls (C);
>
> then draw all the balls (D);
>
> and finally before we poll the user for another command,
>
> tell the user what is going on by echoing current application state to the status bar (E)


(A): As long as user is not ready to quit
(B): Parse the user command
(C): periodically update positions and velocities of the balls
(D): Draw all balls to the computer screen
(E): Sets status bar with number of balls

```
while user command is not quit

        parse and excute  user's command

        if (OperatingSystem::SufficientClockTimeHasElapesd)
           UpdateSimulation()          // update the positions and velocities
                                       //  of the all the balls (in AllWorldBalls set)

        DrawBalls(AllWorldBalls)       // all the balls in AllWorldBalls set


        EchoToStatusBar()              // Sets status bar: number of balls on screen
```

Figure 18.3. Programming structure from a verbalized solution.

Figure 18.3 shows a translation from this verbal solution into a simple programming structure. We introduce the set of *AllWorldBalls* to represent all the balls that are currently on the computer screen. The only other difference between the pseudocode in Figure 18.3 and our verbalized solution is in the added elapsed time check in Step (C): *SufficientClockTimeHasElapsed*. (Recall that the velocities are defined in pixels per second.) To support proper pixel displacements, we must know real elapsed time between updates.

As we add additional details to parse and execute the user's commands (B), the solution must be expanded. The revised solution in Figure 18.4 shows the details of a central parsing switch statement (B) and the support for all three commands a user can issue: defining a new HeroBall (B1); selecting a HeroBall (B2); and adjusting current HeroBall velocity with the slider bars (B3). Undefined user actions (e.g., mouse movement with no button pressed) are simply ignored (B4).

Notice that HeroBall creation (B1) involves three user actions: mouse down (B1), followed by mouse drag (B1-1), and finally mouse up (B1-2). The parsing of this operation is performed in multiple consecutive passes through the outer while-loop (A): the first time through, we create the new HeroBall (B1); in the subsequent passes, we perform the actual *dragging* operation (B1-1). We assume that mouse drag (B1-1) will never be invoked without mouse button down (B1) action, and thus the HeroBall is always defined during the dragging operation.

The *LeftMouseButtonUp* action (B1-2) is an implicit action not defined in the original specification. In our implementation, we choose this implicit action to activate the insertion of the new HeroBall into the AllWorldBalls set. In this way the HeroBall is not a member of the AllWorldBalls set until after the user has completed the dragging operation. This delay ensures that the HeroBall's velocity and position will not be affected when the *UpdateSimulation*() procedure updates all the balls in AllWorldBalls set (C). This means a user can take the time to drag out a new HeroBall without worrying that the ball will free fall before the release of the mouse button. The simple amendment in the drawing operation (D1) ensures a proper drawing of the new HeroBall before it is inserted into the AllWorldBalls set.

When we examine this solution in the context of supporting user interaction, we have to concern ourselves with efficiency issues as well as the potential for increased complexity.

**Efficiency Concerns.** Typically a user interacts with an application in bursts of activity—continuous actions followed by periods of idling. This can be explained by the fact that, as users, we typically perform some tasks in the application and then spend time examining the results. For example, when working with a word

```
                  main()    {
(A):
                      while ( GUISystem::UserAction != Quit )  {
(B):
                          switch (GUISystem::UserAction)  {

(B1): Define new                  // Begins creating a new Hero Ball
Hero Ball                         case GUISystem::LeftMouseButtonDown:
                                      HeroBall = CreateHeroBall()        // hero not in AllWorldBalls set
                                      DefiningNewHeroBall = true
    (B1-1) Support
    for drag actions              // Drags out the new Hero Ball
                                  case GUISystem::LeftMouseButtonDrag:
                                      RefineRadiusAndVelocityOfHeroBall()
    (B1-2) Implicit                   SetSliderBarsWithHeroBallVelocity()
    Action
                                  // Finishes creating the new Hero Ball
                                  case GUISystem::LeftMouseButtonUp:
                                      InsertHeroBallToAllWorldBalls()
                                      DefiningNewHeroBall = false
(B2): Select
current Hero Ball                 // Selects a current hero ball
                                  case GUISystem::RightMouseButtonDown:
                                      HeroBall = SelectHeroBallBasedOnCurrentMouseXY()
                                      if (HeroBall != null)
                                          SetSliderBarsWithHeroBallVelocity()
(B3): Set Hero
Ball Velocity                     // Sets hero velocity with slider bars
                                  case GUISystem::SliderBarChange:
                                      if (HeroBall != null)
(B4): Undefined                           SetHeroBallVelocityWithSliderBarValues()
actions are ignored
                                  // Ignores all other user actions e.g. Mouse Move with no buttons, etc
                                  default:
                          }  // end of switch(userAction)

(C):                      // Move balls by velocities under gravity and remove off-screen ones
                          if ( OperatingSystem::SufficientClockTimeHasElapesd)
                                  UpdateSimulation()

(D):                      DrawBalls(AllWorldBalls)
                              // Draw the new Hero Ball that is currently being defined
(D1): Draw the            if  (DefiningNewHeroBall)
new Hero Ball                 DrawBalls(HeroBall)

(E):                      EchoToStatusBar()     // Sets Status Bar with number of balls currently on screen

                      }  // end of while(UserAction != Quit)
                  }  // end of main() function. Program terminates.
```

**Figure 18.4.** Programming solution based on the control-driven programming model.

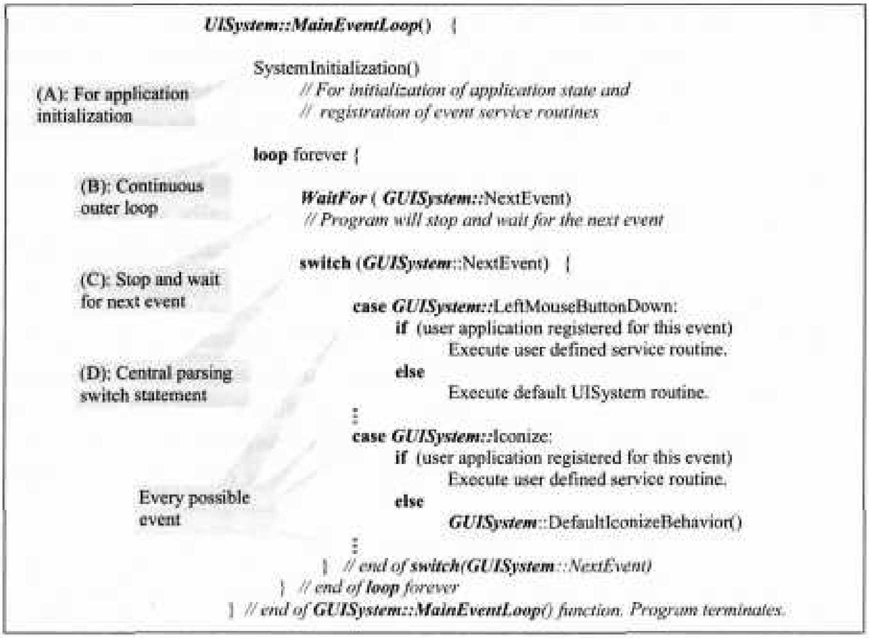
processor, our typical work pattern consists of bursts of typing/editing followed by periods of reading (with no input action). In our example application, we can expect the user to drag out some circles and then observe the free-falling of the circles. The continuous while-loop polling of user commands in the *main*() function means that when the user is not performing any action, our program will still be actively running and wasting machine resources. During activity bursts, at the maximum, users are capable of generating hundreds of input actions per second (e.g., mouse-pixel movements per second). If we compare this rate to the typical CPU instruction capacities that are measured at $10^9$ per second, the huge discrepancy indicates that, *even during activity bursts, the user command-parsing* switch statement (B) is spending most of the time in the default case not doing anything.

**Complexity Concerns.** Notice that our *entire solution* is in the *main*() function. This means that all relevant user actions must be parsed and handled by the user command-parsing switch statement (B). In a modern multi-program shared window environment, many actions performed by users are actually non-application specific. For example, if a user performs a left mouse button click or drag in the drawing area of the program window, our application should react by dragging out a new HeroBall. However, if the user performs the same actions in the title area of the program window, our application should forward these actions to the GUI/Operating/Window system and commence the coordination of moving the entire program window. As experienced users in window environments, we understand that there are numerous such non-application specific operations, and we expect all applications to honor these actions (e.g., iconize, re-size, raise or lower a window, etc.). Following the solution given in Figure 18.4, for every user action that we want to honor, we must include a matching supporting case in the parsing switch statement (B). This requirement quickly increases the complexity of our solution and becomes a burden to implementing any interactive applications.

An efficient GUI system should remain idle by default (not taking up machine resources) and only become active in the presence of interesting activities (e.g., user input actions). Furthermore, to integrate interactive applications in sophisticated multi-programming window environments, it is important that the supporting GUI system automatically takes care of mundane and standard user actions.

### 18.2.2 Event-Driven Programming

Event-driven programming remedies the efficiency and complexity concerns with a default *MainEventLoop*() function defined in the GUI system. For event-driven programs, the *MainEventLoop*() replaces the *main*() function, because all programs start and end in this function. Just as in the case of the *main*() function for control-driven programming, when the *MainEventLoop*() function re-



```
UISystem::MainEventLoop() {

    SystemInitialization()
        // For initialization of application state and
        // registration of event service routines

    loop forever {

        WaitFor ( GUISystem::NextEvent)
        // Program will stop and wait for the next event

        switch (GUISystem::NextEvent) {

            case GUISystem::LeftMouseButtonDown:
                if (user application registered for this event)
                    Execute user defined service routine.
                else
                    Execute default UISystem routine.
            ⋮
            case GUISystem::Iconize:
                if (user application registered for this event)
                    Execute user defined service routine.
                else
                    GUISystem::DefaultIconizeBehavior()
            ⋮
        } // end of switch(GUISystem::NextEvent)
    } // end of loop forever
} // end of GUISystem::MainEventLoop() function. Program terminates.
```

(A): For application initialization
(B): Continuous outer loop
(C): Stop and wait for next event
(D): Central parsing switch statement
Every possible event

**Figure 18.5.** The default *MainEventLoop* function.

turns, all work should have been completed, and the program terminates. The *MainEventLoop*() function defines the central control structure for all event-driven programming solutions and typically cannot be changed by a user application. In this way, the overall control of an application is actually external to the user's program code. For this reason, event-driven programming is also referred to as the external control model.

Figure 18.5 depicts a typical *MainEventLoop*() implementation. In this case, our program is the user application that is based on the *MainEventLoop*() function. Structurally, the *MainEventLoop*() is very similar to the *main*() function of Figure 18.4: with a continuous loop (B) containing a central parsing switch statement (D). The important differences between the two functions include:

- (A) SystemInitialization(): Recall that event-driven programs start and end in the *MainEventLoop*() function. *SystemInitialization*() is a mechanism defined to invoke the user program from within the *MainEventLoop*(). It is expected that user programs implement *SystemInitialization*() to initialize the application state and to register event service routines (refer to the discussion in (D)).

- (B) Continuous outer loop: Since this is a general control structure to be shared by all event-driven programs, there is no way to determine the termination condition. User program are expected to override appropriate event service routines and terminate the program from within the service routine.

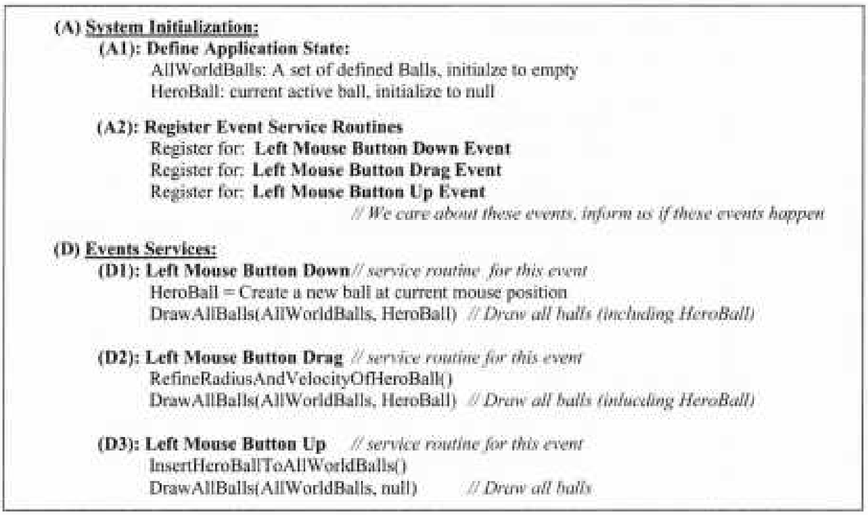
- (C) Stop and wait: Instead of actively polling the user for actions (wasting machine resources), the *MainEventLoop*() typically stops the entire application process and waits for asynchronous operating system calls to re-activate the application process in the presence of relevant user actions.

- (D) Events and central parsing switch statement: Included in this statement are all possible actions/events (cases) that a user can perform. Associated with each event (case) is a default behavior and a toggle that allows user applications to override the default behavior. During *SystemInitialization*(), the user application can register an alternate service routine for an event by toggling the override.

To develop an event-driven solution, our program must first register event service routines with the GUI system. After that, our entire program solution is based on waiting and servicing user events. While control-driven programming solutions are based on an algorithmic organization of control structures in the *main*() function, an event-driven programming solution is based on the specification of events that cause changes to a defined application state. This is a different paradigm for designing programming solutions. The key difference here is that, as programmers, we have no explicit control over the algorithmic organization of the events: over which, when, or how often an event should occur.

The program in Figure 18.6 implements the left mouse button operations for our ball shooting program. We see that during system initialization (A), the program defines an appropriate application state (A1) and registers left mouse button (LMB) down/drag/up events (A2). The corresponding event service routines (D1, D2, and D3) are also defined. At the end of each event service routine, we redraw all the balls to ensure that the user can see an up-to-date display at all times. Notice the absence of any control structure organizing the initialization and service routines. Recall that this is an event-driven program: the overall control structure is defined in the MainEventLoop which is external to our solution.

Figure 18.7 shows how our program from Figure 18.6 is linked with the predefined *MainEventLoop*() from the GUI system. The *MainEventLoop*() calls the *SystemInitialization*() function defined in our solution (A). As described, after the initialization, our entire program is essentially the three event service routines (D1, D2, and D3). However, we have no control over the invocation of these routines. Instead, a user performs actions that trigger events which drive



```
(A) System Initialization:
        (A1): Define Application State:
                AllWorldBalls: A set of defined Balls, initialze to empty
                HeroBall: current active ball, initialize to null

        (A2): Register Event Service Routines
                Register for:  Left Mouse Button Down Event
                Register for:  Left Mouse Button Drag Event
                Register for:  Left Mouse Button Up Event
                                        // We care about these events, inform us if these events happen

(D) Events Services:
        (D1): Left Mouse Button Down// service routine  for this event
                HeroBall = Create a new ball at current mouse position
                DrawAllBalls(AllWorldBalls, HeroBall)  // Draw all balls (including HeroBall)

        (D2): Left Mouse Button Drag  // service routine for this event
                RefineRadiusAndVelocityOfHeroBall()
                DrawAllBalls(AllWorldBalls, HeroBall)  // Draw all balls (inlucding HeroBall)

        (D3): Left Mouse Button Up     // service routine for this event
                InsertHeroBallToAllWorldBalls()
                DrawAllBalls(AllWorldBalls, null)        // Draw all balls
```

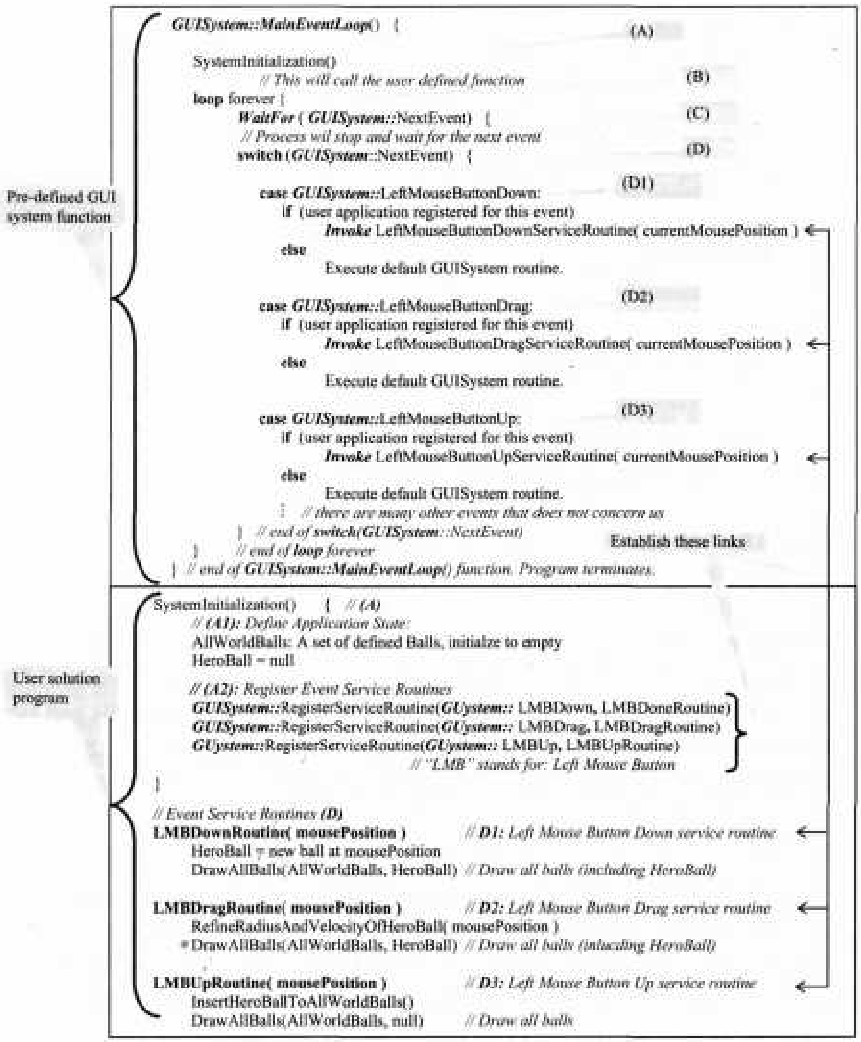
**Figure 18.6.** A simple event-driven program specification.

these routines. These routines in turn change the application state. In this way, an event-driven programming solution is based on specification of events (LMB events) that cause changes to a defined application state (AllWorldBalls and HeroBall). Since the user command parsing switch statement (D in Figure 18.7) in the *MainEventLoop*() contains a case and the corresponding default behavior for every possible user actions, without any added complexity, our solution honors the non-application specific actions in the environment (e.g., iconize, moving, etc).

In the context of event-driven programming, an event can be perceived as an asynchronous notification that something interesting has happened. The messenger for the *notification* is the underlying GUI system. The mechanism for receiving an event is via overriding the corresponding event service routine.

For these reasons, when discussing event-driven programming, there is always a supporting GUI system. This GUI system is generally referred to as the Graphics User Interface (GUI) Application Programming Interface (API). Examples of GUI APIs include: *Java Swing Library*, *OpenGL Utility ToolKit (GLUT)*, *The Fast Light ToolKit (FLTK)*, *Microsoft Foundation Classes (MFC)*, etc.

From the above discussion, we see that the registration for services of appropriate events is the core of designing and developing solutions for event-driven programs. Before we begin developing a complete solution for our ball shooting program, let us spend some time understanding *events*.



```
GUISystem::MainEventLoop()  {                                              (A)

   SystemInitialization()
         // This will call the user defined function                          (B)
   loop forever {
        WaitFor ( GUISystem::NextEvent)  {                                    (C)
        // Process wil stop and wait for the next event
        switch (GUISystem::NextEvent)  {                                      (D)

           case GUISystem::LeftMouseButtonDown:                          (D1)
              if (user application registered for this event)
                   Invoke LeftMouseButtonDownServiceRoutine( currentMousePosition )
              else
                   Execute default GUISystem routine.

           case GUISystem::LeftMouseButtonDrag:                          (D2)
              if (user application registered for this event)
                   Invoke LeftMouseButtonDragServiceRoutine( currentMousePosition )
              else
                   Execute default GUISystem routine.

           case GUISystem::LeftMouseButtonUp:                            (D3)
              if (user application registered for this event)
                   Invoke LeftMouseButtonUpServiceRoutine( currentMousePosition )
              else
                   Execute default GUISystem routine.
              ⋮  // there are many other events that does not concern us
        }  // end of switch(GUISystem::NextEvent)
   }     // end of loop forever
} // end of GUISystem::MainEventLoop() function. Program terminates.
```

```
SystemInitialization()     {  // (A)
     // (A1): Define Application State:
     AllWorldBalls: A set of defined Balls, initialze to empty
     HeroBall = null

     // (A2): Register Event Service Routines
     GUISystem::RegisterServiceRoutine(GUystem:: LMBDown, LMBDoneRoutine)
     GUISystem::RegisterServiceRoutine(GUystem:: LMBDrag, LMBDragRoutine)
     GUystem::RegisterServiceRoutine(GUystem:: LMBUp, LMBUpRoutine)
                                   // "LMB" stands for: Left Mouse Button
}

// Event Service Routines (D)
LMBDownRoutine( mousePosition )          // D1: Left Mouse Button Down service routine
     HeroBall = new ball at mousePosition
     DrawAllBalls(AllWorldBalls, HeroBall) // Draw all balls (including HeroBall)

LMBDragRoutine( mousePosition )          // D2: Left Mouse Button Drag service routine
     RefineRadiusAndVelocityOfHeroBall( mousePosition )
     DrawAllBalls(AllWorldBalls, HeroBall) // Draw all balls (inlucding HeroBall)

LMBUpRoutine( mousePosition )            // D3: Left Mouse Button Up service routine
     InsertHeroBallToAllWorldBalls()
     DrawAllBalls(AllWorldBalls, null)      // Draw all balls
```

**Figure 18.7.** Linking MainEventLoop with our solution.

## Graphical User Interface (GUI) Events

In general, an application may receive events generated by the user, the application itself, or by the GUI system. In this section, we describe each of these event sources and discuss the application's role in servicing these events.

**S1: The User.** These are events triggered by the actions a user performs on the input devices. Notice that input devices include actual hardware devices (e.g., mouse, keyboard, etc.) and/or software-simulated GUI elements (e.g., slider bars, combo boxes, etc.). Typically, a user performs actions for two very different reasons:

- **S1a: Application specific.** These are input actions that are part of the application. Clicking and dragging in the application screen area to create a HeroBall is an example of an action performed on a hardware input device. Changing the slider bars to control the HeroBall's velocity is an example of an action performed on a software-simulated GUI element. Both of these actions and the resulting events are application specific; the application (our program) is solely responsible for servicing these events.
- **S1b: General.** These are input actions defined by the operating environment. For example, a user clicks and drags in the window title-bar area expecting to move the entire application window. The servicing of these types of events requires collaboration between our application and the GUI system. We will discuss the servicing of these types of events in more detail when explaining events that originate from the GUI system in S3c.

Notice that the meaning of a user's action is *context sensitive*. It depends on where the action is performed: click and drag in the application screen area vs. slider bar vs. application window title-bar area. In any case, the underlying GUI system is responsible for parsing the context and determining which application element receives a particular event.

**S2: The Application.** These are events defined by the application, typically depending on some run-time conditions. During run time, if and when the condition is favorable, the supporting GUI system triggers the event and conveys the favorable conditions to the application. A straightforward example is a periodic alarm. Modern GUI systems typically allow an application to define (sometimes multiple) *timer events* . Once defined, the GUI system will trigger an event to wake up the application when the timer expires. As we will see, this timer event is essential for supporting real-time simulations. Since the application (our program) requested the generation of these types of events, our program is solely responsible for serving them. The important distinction between application-defined and user-generated events is that application-defined events can be *spontaneous*: when properly defined, even when the user is not doing anything, these types of events may trigger.

**S3: The GUI System.** These are events that originate from within the GUI system in order to convey state information to the application. There are typically three reasons for these events:

- **S3a: Internal GUI states change.** These are events signaling an internal state change of the GUI system. For example, the GUI system typically generates an event before the creation of the application's main window. This provides an opportunity for the application to perform the corresponding initialization. In some GUI systems (e.g., MFC) the *SystemInitialization*() functionality is accomplished with these types of events: user applications are expected to override the appropriate windows' creation event and initialize the application state. Modern, general purpose commercial GUI systems typically define a large number of events signaling detailed state changes in anticipation of supporting different types of applications and requirements. For example, for the creation of the application's main window, the GUI system may define events for the following states:

  - before resource allocation;
  - after resource allocation but before initialization;
  - after initialization but before initial drawing, etc.

  A GUI system usually defines meaningful default behaviors for such events. To program an effective application based on a GUI system, one must understand the different groups of events and only service the appropriate selections.

- **S3b: External environment requests attention.** These are events indicating that there are changes in the operating environment that potentially require application attention. For example, a user has moved another application window to cover a portion of our application window, or a user has minimized our application window. The GUI system and the window environment typically have appropriate service routines for these types of events. An application would only choose to service these events when special actions must be performed. For example, in a real-time simulation program, the application may choose to suspend the simulation if the application window is minimized. In this situation, an application must service the minimized and the maximized events.

- **S3c: External environment requests application collaboration.** These are typically events requesting the application's collaboration to complete the service of *general user actions* (please refer to S1b). For example, if a user click-drags the application window's title bar, the GUI system reacts by letting the user "drag" the entire application window. This "drag" operation is implemented by continuously erasing and redrawing the entire application window at the current mouse pointer position on the computer display. The GUI system has full knowledge of the appearance of the application window (e.g., the window frames, the menus, etc.), but it has no knowledge of the application window content (e.g., how many free falling balls traveling at what velocity, etc.). In this case, the GUI system redraws the application window frame and generates a Redraw/Paint event for the application, requesting assistance in completing the service of the user's "drag" operation. As an application in a shared window environment, our application is expected to honor and service these types of events. The most common events in this category include: Redraw/Paint and Resize. Redraw/Paint is the single most important event an application must service, because it supports the most common operations a user may perform in a shared window environment. Resize is also an important event to which the application must respond because the application is in charge of GUI element placement policy (e.g., if window size is increased, how should the GUI elements be placed in the larger window).

### 18.2.3 The Event-Driven Ball Shooting Program

In Section 18.2.1, we started a control-driven programming solution to the ball shooting program based on verbalizing the conditions (controls) under which the appropriate actions should be taken:

> while favorable condition, parse the input ...

As we have seen, with appropriate modifications, we were able to detail the control structures for our solution.

From the discussion in Section 18.2.2, we see that to design an event-driven programming solution we must

1. define the application state;
2. describe how user actions change the application state;
3. map the user actions to events that the GUI system supports; and
4. override corresponding event service routines to implement user actions.

The specification in Section 18.1 detailed the behaviors of our ball shooting program. The description is based on actions performed on familiar input devices (e.g., slider bars and mouse) that change the appearance on the display screen. Thus, the specification from Section 18.1 describes items (2) and (3) from the above list without explicitly defining what the application state is. Our job in designing a solution is to derive the implicitly defined application state and design the appropriate service routines.

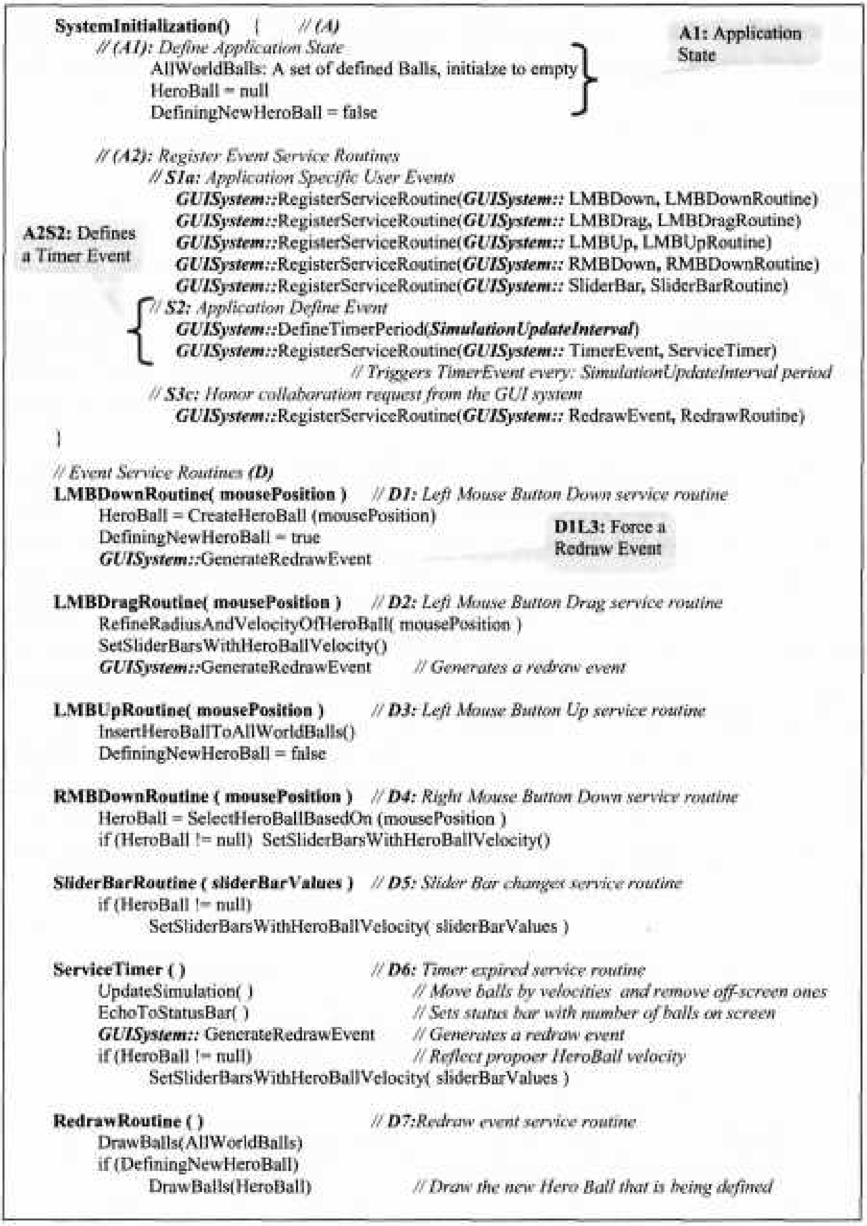
Figure 18.8 presents our event-driven programming solution. As expected, the application state (A1) is defined in *SystemInitialization*(). The AllWorldBalls set and HeroBall can be derived from the specification in Section 18.1. The *DefiningNewHeroBall* flag is a transient (temporary) application state designed to support user actions across multiple events (click-and-drag). Using *transient* application states is a common approach to support consecutive inter-related events.

Figure 18.8 shows the registration of three types of service routines (A2):

- user-generated application specific events (S1a);
- an application defined event (S2);
- a GUI system-generated event requesting collaboration (S3c).

The timer event definition (A2S2) sets up a periodic alarm for the application to update the simulation of the free falling balls. The service routines of the user-generated application specific events (D1-D5) are remarkably similar to the corresponding case statements in the control-driven solution presented in Figure 18.4 (B1-B3). It should not be surprising that this is so, because we are implementing the exact same user actions based on the same specification. Line 3 of the *LMBDownRoutine*() (D1L3) demonstrates that, when necessary, our application can request the GUI system to initiate events. In this case, we signal the GUI system that an application redraw is necessary. Notice that event service routines are simply functions in our program. This means, at D1L3 we could also call *RedrawRoutine*() (D7) directly. The difference is that a call to *RedrawRoutine*() will force a redraw immediately while requesting the generation of a redraw event allows the GUI system to optimize the number of redraws. For example, if the user performs a LMB click and starts dragging immediately, with our D1 and D2 implementation, the GUI system can gather the many *GenerateRedrawEvent* requests in a short period of time and only generate one re-draw event. In this way, we can avoid performing more redraws than necessary.

In order to achieve a smooth animation, we should perform about 20–40 updates per second. It follows that the *SimulationUpdateInterval* should be no more than 50 milliseconds so that the *ServiceTimer*() routine can be invoked more than 20 times per second. (Notice that a redraw event is requested at the end of the *ServiceTimer*() routine.) This means, at the very least, our application is guaranteed to receive more than 20 redraw events in one second. For this reason, the



```
SystemInitialization()   {      // (A)
        // (A1): Define Application State
                AllWorldBalls: A set of defined Balls, initialze to empty
                HeroBall = null
                DefiningNewHeroBall = false

        // (A2): Register Event Service Routines
                // S1a: Application Specific User Events
                    GUISystem::RegisterServiceRoutine(GUISystem:: LMBDown, LMBDownRoutine)
                    GUISystem::RegisterServiceRoutine(GUISystem:: LMBDrag, LMBDragRoutine)
                    GUISystem::RegisterServiceRoutine(GUISystem:: LMBUp, LMBUpRoutine)
                    GUISystem::RegisterServiceRoutine(GUISystem:: RMBDown, RMBDownRoutine)
                    GUISystem::RegisterServiceRoutine(GUISystem:: SliderBar, SliderBarRoutine)
                // S2: Application Define Event
                    GUISystem::DefineTimerPeriod(SimulationUpdateInterval)
                    GUISystem::RegisterServiceRoutine(GUISystem:: TimerEvent, ServiceTimer)
                                        // Triggers TimerEvent every: SimulationUpdateInterval period
                // S3c: Honor collaboration request from the GUI system
                    GUISystem::RegisterServiceRoutine(GUISystem:: RedrawEvent, RedrawRoutine)
}

// Event Service Routines (D)
LMBDownRoutine( mousePosition )     // D1: Left Mouse Button Down service routine
        HeroBall = CreateHeroBall (mousePosition)
        DefiningNewHeroBall = true
        GUISystem::GenerateRedrawEvent

LMBDragRoutine( mousePosition )     // D2: Left Mouse Button Drag service routine
        RefineRadiusAndVelocityOfHeroBall( mousePosition )
        SetSliderBarsWithHeroBallVelocity()
        GUISystem::GenerateRedrawEvent        // Generates a redraw event

LMBUpRoutine( mousePosition )       // D3: Left Mouse Button Up service routine
        InsertHeroBallToAllWorldBalls()
        DefiningNewHeroBall = false

RMBDownRoutine ( mousePosition )    // D4: Right Mouse Button Down service routine
        HeroBall = SelectHeroBallBasedOn (mousePosition )
        if (HeroBall != null)  SetSliderBarsWithHeroBallVelocity()

SliderBarRoutine ( sliderBarValues )   // D5: Slider Bar changes service routine
        if (HeroBall != null)
                SetSliderBarsWithHeroBallVelocity( sliderBarValues )

ServiceTimer ( )                    // D6: Timer expired service routine
        UpdateSimulation( )                 // Move balls by velocities  and remove off-screen ones
        EchoToStatusBar( )                  // Sets status bar with number of balls on screen
        GUISystem:: GenerateRedrawEvent     // Generates a redraw event
        if (HeroBall != null)               // Reflect propoer HeroBall velocity
                SetSliderBarsWithHeroBallVelocity( sliderBarValues )

RedrawRoutine ( )                   // D7:Redraw event service routine
        DrawBalls(AllWorldBalls)
        if (DefiningNewHeroBall)
                DrawBalls(HeroBall)         // Draw the new Hero Ball that is being defined
```

**A1:** Application State

**A2S2:** Defines a Timer Event

**D1L3:** Force a Redraw Event

**Figure 18.8.** Programming solution based on the event-driven programming model.

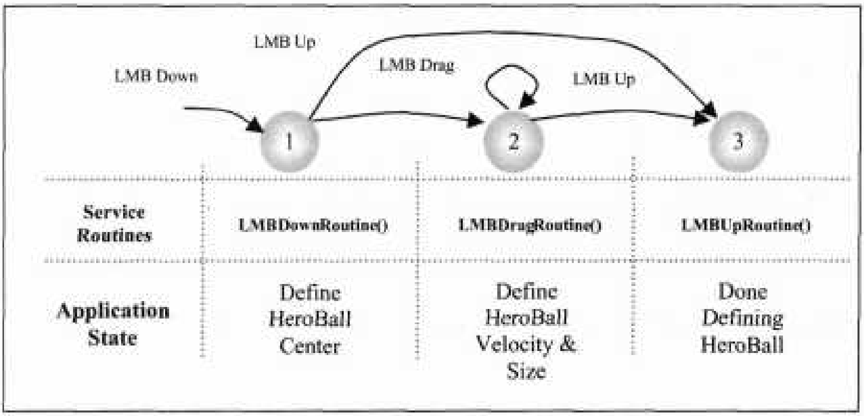
*GenerateRedrawEvent* requests in D1 and D2 are really not necessary. The servicing of our timer events will guarantee us an up-to-date display screen at all times.

### 18.2.4 Implementation Notes

The application state of an event-driven program must persist over the entire life time of the program. In terms of implementation, this means that the application state must be defined based on variables that are dynamically allocated during run time and that reside on the heap memory. These are in contrast to local variables that reside on the stack memory and which do not persist over different function invocations.

The mapping of user actions to events in the GUI system often results in *implicit* and/or undefined events. In our ball shooting program, the actions to define a HeroBall involve left mouse button down and drag. When mapping these actions to events in our implementation (in Figure 18.4 and Figure 18.8), we realize that we should also pay attention to the implicit mouse button up event. Another example is the HeroBall selection action: right mouse button down. In this case, right mouse button drag and up events are not serviced by our application, and thus, they are undefined (to our application).

When one user action (e.g., *"drag out the HeroBall"*) is mapped to a group of consecutive events (e.g., mouse button down, then drag, then up) a finite state diagram can usually be derived to help design the solution. Figure 18.9 depicts the finite state diagram for defining the HeroBall in our ball shooting program.



| | | | |
|---|---|---|---|
| **Service Routines** | **LMBDownRoutine()** | **LMBDragRoutine()** | **LMBUpRoutine()** |
| **Application State** | Define HeroBall Center | Define HeroBall Velocity & Size | Done Defining HeroBall |

**Figure 18.9.** State diagram for defining the HeroBall.

The left mouse button down event puts the program into State 1 where, in our solution from Figure 18.8, *LMBDownRoutine*() implements this state and defines the center of the HeroBall, etc. In this case the transition between states is triggered by the mouse events, and we see that it is physically impossible to move from State 2 back to State 1. However, we do need to handle the case where the user action causes a transition from State 1 to State 3 directly (mouse button down and release without any dragging actions). This state diagram helps us analyze possible combinations of state transitions and perform appropriate initializations.

Event-driven applications interface with the user through physical (e.g., mouse clicks) or simulated GUI elements (e.g., quit button, slider bars). An input GUI element (e.g., the quit button) is an artifact (e.g.,·an icon) for the user to direct changes to the application state, while an output GUI element (e.g., the status bar) is an avenue for the application to present application state information to the user as feedback. For both types of elements, information only flows in one direction—either from the user to the application (input) or from the application to the user (output). When working with GUI elements that serve both input and output purposes, special care is required. For example, after the user selects or defines a HeroBall, the slider bars reflects the velocity of the free falling HeroBall (output), while at any time, the user can manipulate the slider bar to alter the HeroBall velocity (input). In this case, the GUI element's displayed state and the application's internal state are connected. The application must ensure that these two states are consistent. Notice that in the solution shown in Figure 18.4, this state consistency is not maintained. When a user clicks the RMB (B2 in Figure 18.4) to select a HeroBall, the slider bar values are updated properly; however, as the HeroBall free falls under gravity, the slider bar values are not updated. The solution presented in Figure 18.8 fixes this problem by using the *ServiceTimer*() function.

Event service routines are functions defined in our program that cause a *callback* from the MainEventLoop in the presence of relevant events. For this reason, these service routines are also referred to as *callback* functions. The application program registers callback functions with the GUI system by passing the address of the function to the GUI system. This is the registration mechanism implied in Figure 18.7 and Figure 18.8. Simple GUI systems (e.g., GLUT or FLTK) usually support this form of registration mechanism. The advantage of this mechanism is that it is easy to understand, straightforward to program, and often contributes to a small memory footprint in the resulting program. The main disadvantage of this mechanism is the lack of organizational structure for the callback functions.

In commercial GUI systems, there are a large numbers of events with which user applications must deal, and a structured organization of the service routines can assist the programmability of the GUI system. Modern commercial GUI systems are often implemented based on object-oriented languages (e.g., C++ for MFC, Java for Java Swing). For these systems, many event service registrations are implemented as sub-classes of an appropriate GUI system class, and they override corresponding virtual functions. In this way, the event service routines are organized according to the functionality of GUI elements. The details of different registration mechanisms will be explained in Section 18.4.1 when we describe the implementation details.

Event service routines (or callback functions) are simply functions in our program. However, these functions also serve the important role as the server of external asynchronous events. The following are guidelines one should take into account when implementing event service routines:

1. An event service routine should only service the triggering event and immediately return the control back to the *MainEventLoop*(). This may seem to be a "no-brainer." However, because of our familiarity with control-driven programming, it is often tempting to anticipate/poll subsequent events with a control structure in the service routine. For example, when servicing the left mouse button down event, we know that the mouse drag event will happen next. After allocating and defining the circle center, we have properly initialized data to work with the HeroBall object. It may seem easier to simply include a while loop to poll and service mouse drag events. However, with all the other external events that may happen (e.g., timer event, external redraw events, etc.), this monopolizing of control in one service routine is not only a bad design decision, but also it may cause the program to malfunction.

2. An event service routine should be *stateless*, and individual invocations should be independent. In terms of implementation, this essentially means event service routines should not define local *static* variables that record data from previous invocations. Because we have no control over when, or how often, events are triggered, when these variables are used as data, or conditions for changing application states, it can easily lead to disastrously and unnecessarily complex solutions. We can always define extra state variables in the application state to record temporary state information that must persist over multiple event services. The *DefiningNewHeroBall* flag in Figure 18.8 is one such example.

3. An event service routine should check for invocation conditions regardless of common sense *logical* sequence. For example, although logically, a mouse drag event can never happen unless a mouse down event has already occurred, in reality, a user may depress a mouse button from outside

of our application window and then drag the mouse into our application window. In this case, we will receive a mouse drag event without the corresponding mouse down event. For this reason, the mouse drag service routine should check the *invocation condition* that the proper initialization has indeed happened. Notice in Figure 18.8, we do not include proper invocation condition checking. For example, in the *LMBDragRoutine*(), we do not verify that *LMBDownRotine*() has been invoked (by checking the *DefiningNewHeroBall* flag). In a real system, this may causes the program to malfunction and/or crash.

### 18.2.5 Summary

In this section we have discussed *programming models* or *strategies for organizing statements of our program.* We have seen that for *interactive* applications, where an application continuously waits and reacts to a user's input actions, organizing the program statements based on designing control structures results in complex and inefficient programs. Existing GUI systems analyze all possible user actions, design control structures to interact with the user, implement default behaviors for all user actions, and provide this functionality in GUI APIs. To develop interactive applications, we take advantage of the existing control structure in the GUI API (i.e., the *MainEventLoop*()) and modify the default behaviors (via event service routines) of user actions. In order to properly collaborate with existing GUI APIs, the strategy for organizing the program statements should be based on specifying user actions that cause changes to the application state.

Now that we understand how to organize the statements of our program, let's examine strategies for organizing functional modules of our solution.

## 18.3 The Modelview-Controller Architecture

The event-driven ball shooting program presented in Section 18.2.3 and Figure 18.8 addresses programmability and efficiency issues when interacting with a user. In the development of that model, we glossed over many supporting functions (e.g., *UpdateSimulation*()) needed in our solution. In this section, we develop strategies for organizing these functions. Notice that we are not interested in the implementation details of these functions. Instead, we are interested in grouping related functions into components. We then pay attention to how the different *components* collaborate to support the functionality of our application.

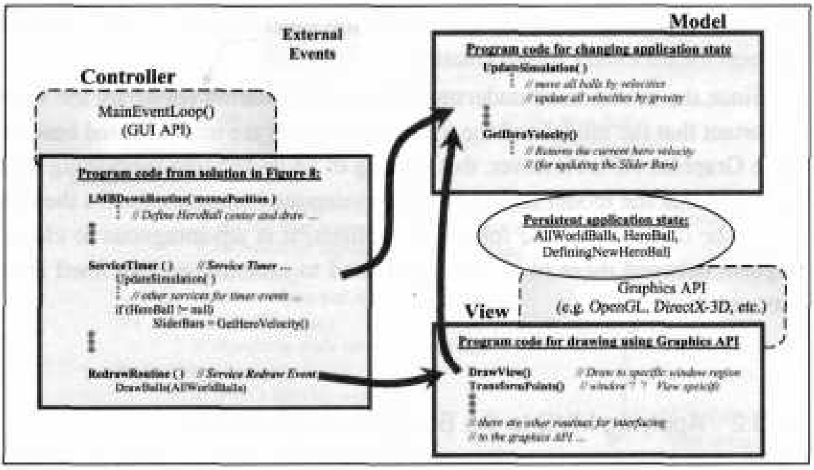
In this way, we derive a framework that is suitable for implementing general interactive graphics applications. With a proper framework guiding our design and implementation, we will be better equipped to develop programs that are easier to understand, maintain, modify, and expand.

### 18.3.1 The Modelview-Controller Framework

Based on our experience developing solutions in Section 18.2, we understand that *interactive graphics applications* can be described as applications that allow users to interactively update their internal states. These applications provide real-time visualization of their internal states (e.g., the free-falling balls) with computer graphics (e.g., drawing circles). The *modelview-controller (MVC)* framework provides a convenient structure for discussing this type of application. In the MVC framework, the *model* is the application state, the *view* is responsible for setting up support for the model to present itself to the user, and the *controller* is responsible for providing the support for the user to interact with the model. Within this framework, our solution from Figure 18.8 is simply the implementation of a controller. In this section, we will develop the understanding of the other two components in the MVC framework and how these components collaborate to support interactive graphics applications.

Figure 18.10 shows the details of a MVC framework to describe the behavior of a typical interactive graphics application. We continue to use the ball shooting program as our example to illustrate the details of the components. The top-right rectangular box is the model, the bottom-right rectangular box is the view, and the rectangular box on the left is the controller component. These three boxes represent program code we, as application developers, must develop. The two dotted rounded boxes represent external graphics and GUI APIs. These are *the external libraries* that we will use as a base for building our system. Examples of popular Graphics APIs include OpenGL, Microsoft Direct-3D (D3D), Java 3D, among others. As mentioned in Section 18.2.2, examples of popular GUI APIs include GLUT, FLTK, MFC, and Java Swing Library.

The model component defines the persistent application state (e.g., AllWorldBalls, HeroBalls, etc.) and implements interface functions for this application state (e.g., *UpdateSimulation*()). Since we are working with a "graphics" application, we expect graphical primitives to be part of the representation for the application state (e.g., CirclePrimitives). This fact is represented in Figure 18.10 by the application state (the ellipse) partially covering the Graphics API box. In the rest of this section, we will use the terms model and persistent application state interchangeably.


External
Events
Controller
MainEventLoop()
(GUI API)
Program code from solution in Figure 8:
Model
Program code for changing application state
Persistent application state:
AllWorldBalls, HeroBall,
DefiningNewHeroBall
Graphics API
(e.g. OpenGL, DirectX-3D, etc.)
View
Program code for drawing using Graphics API

```
LMBDownRoutine( mousePosition )
    // Define HeroBall center and draw ...

ServiceTimer ( )     // Service Timer ...
    UpdateSimulation( )
    // other services for timer events ...
    if (HeroBall != null)
        SliderBars = GetHeroVelocity()

RedrawRoutine ( )    // Service Redraw Event
    DrawBalls(AllWorldBalls)
```

```
UpdateSimulation( )
    // move all balls by velocities
    // update all velocities by gravity

GetHeroVelocity()
    // Returns the current hero velocity
    // (for updating the Slider Bars)
```

```
DrawView()          // Draw to specific window region
TransformPoints()   // window ? ? View specific

// there are other routines for interfacing
// to the graphics API ...
```

**Figure 18.10.** Components of an interactive graphics application.

The view component is in charge of *drawing* to the *drawing area* on the application window (e.g., drawing the free falling balls). More specifically, the view component is responsible for initializing the graphics API transformation such that drawing of the model's graphical primitives will appear in the appropriate drawing area. The arrow from the view to the model component signifies that the actual application state redraw must be performed by the model component. Only the model component knows the details of the entire application state (e.g., size and location of the free falling circles) so only the model component can redraw the entire application. The view component is also responsible for transforming user mouse click positions to a coordinate system that the model understands (e.g., mouse button clicks for dragging out the hero ball).

The top left *external events* arrow in Figure 18.10 shows that all external events are handled by the *MainEventLoop*(). The relevant events will be forwarded to the event service routines in the controller component. Since the controller component is responsible for interacting with the user, the design is typically based on event-driven programming techniques. The solution presented in Section 18.2.3 and Figure 18.8 is an example of a controller component implementation. The arrow from the controller to the model indicates that most external events eventually change the model component (e.g., creating a new HeroBall or changing the current HeroBall velocity). The arrow from the controller to the view component indicates that the user input point transformation is handled by the view component. Controllers typically return mouse click positions in the device coordinate with the origin at the top-left corner. In the application model, it is more convenient for us to work with a coordinate system with a lower-left origin.

The view component with its transformation functionality has the knowledge to perform the necessary transformation.

Since the model must understand the transformation set up by the view, it is important that the model and the view components are implemented based on the same Graphics API. However, this sharing of an underlying supporting API does not mean that the model and view are an integrated component. On the contrary, as will be discussed in the following sections, it is advantageous to clearly distinguish between these two components and to establish well-defined interfaces between them.

### 18.3.2 Applying MVC to the Ball Shooting Program

With the described MVC framework and the understanding of how responsibilities are shared among the components, we can now extend the solution presented in Figure 18.8 and complete the design of the ball shooting program.

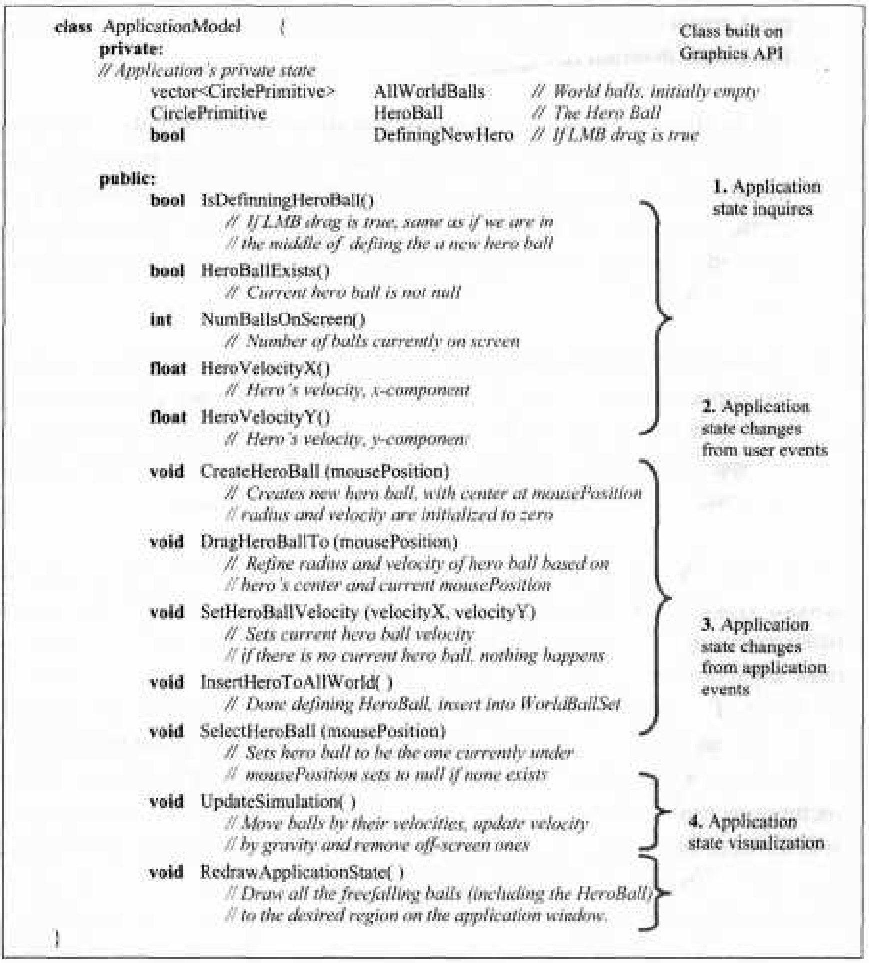
#### The Model

The model is the application state and thus this is the core of our program. When describing approaches to designing an event-driven program in Section 18.2.3, the first two points mentioned were:

1. define the application state, and
2. describe how a user changes this application state.

These two points are the guidelines for designing the model component. In an object-oriented environment, the model component can be implemented as classes, and *state of the application* can be implemented as instance variables, with "*how a user changes this application state*" implemented as methods of the classes.

Figure 18.11 shows that the instance variables representing the state are typically private to the model component. As expected, we have a "very graphical" application state. To properly support this state, we define the CirclePrimitive class based on the underlying graphics API. The CirclePrimitive class supports the definition of center, radius, drawing, and moving of the circle, etc. Figure 18.11 also shows the four categories of methods that a typical model component must support:



```
class ApplicationModel      {
      private:
      // Application's private state
            vector<CirclePrimitive>     AllWorldBalls       // World balls, initially empty
            CirclePrimitive             HeroBall            // The Hero Ball
            bool                        DefiningNewHero     // If LMB drag is true

      public:
            bool    IsDefinningHeroBall()
                        // If LMB drag is true, same as if we are in
                        // the middle of  defing the a new hero ball
            bool    HeroBallExists()
                        // Current hero ball is not null
            int     NumBallsOnScreen()
                        // Number of balls currently on screen
            float   HeroVelocityX()
                        // Hero's velocity, x-component
            float   HeroVelocityY()
                        // Hero's velocity, y-componen
            void    CreateHeroBall (mousePosition)
                        // Creates new hero ball, with center at mousePosition
                        // radius and velocity are initialized to zero
            void    DragHeroBallTo (mousePosition)
                        // Refine radius and velocity of hero ball based on
                        // hero's center and current mousePosition
            void    SetHeroBallVelocity (velocityX, velocityY)
                        // Sets current hero ball velocity
                        // if there is no current hero ball, nothing happens
            void    InsertHeroToAllWorld( )
                        // Done defining HeroBall, insert into WorldBallSet
            void    SelectHeroBall (mousePosition)
                        // Sets hero ball to be the one currently under
                        // mousePosition sets to null if none exists
            void    UpdateSimulation( )
                        // Move balls by their velocities, update velocity
                        // by gravity and remove off-screen ones
            void    RedrawApplicationState( )
                        // Draw all the freefalling balls (including the HeroBall)
                        // to the desired region on the application window.
}
```

**Figure 18.11.** The model component of the ball shooting program.

1. **Application state inquiries.** These are functions that return the application state. These functions are important for maintaining up-to-date GUI elements (e.g., status echo or velocity slider bars).

2. **Application state changes from user events.** These are functions that change the application state according to a user's input actions. Notice that the function names should reflect the functionality (e.g., CreateHeroBall) and not the user event actions (e.g., ServiceLMBDown). It is common for a group of functions to support a defined finite state transition. For exam-

ple, CreateHeroBall, DragHeroBall, and InsertHeroToWorld implement the finite state diagram of Figure 18.9.

3. **Application state changes from application (timer) events.** This is a function that updates the application state resulting from purposeful and usually synchronous application timer events. For the ball shooting program, we update all of the velocities, displace the balls' positions by the updated velocities and compute ball-to-ball collisions, as well as remove off-screen balls.

4. **Application state visualization.** This is a function that knows how to draw the application state (e.g., drawing the necessary number of circles at the corresponding positions). It is expected that a view component will initialize appropriate regions on the application window, set up transformations, and invoke this function to draw into the initialized region.

It is important to recognize that the user's asynchronous events are arriving in between synchronous application timer events. In practice, a user observes an instantaneous application state (the graphics in the application window) and generates asynchronous events to alter the application state for the next round of simulation. For example, a user sees that the HeroBall is about to collide with another ball and decides to change the HeroBall's velocity to avoid the collision that would have happened in the next round of simulation. This means, before synchronous timer update, we must ensure all existing asynchronous user events are processed. In addition, the application should provide continuous feedback to ensure that users are observing an up-to-date application state. This subtle handling of event arrival and processing order is not an issue for simple, single-user applications like our ball shooting program. On large scale multi-user networked interactive systems, where input event and output display latencies may be significant, the *UpdateSimulation*() function is often divided into pre-update, update, and post-update.

### The View

Figure 18.12 shows the ApplicationView class supporting the two main functionalities of a view component: coordinate space transformation and initialization for redraw. As discussed earlier, the controller is responsible for calling the *DeviceToWorldXform*() to communicate user input points to the model component. The viewport class is introduced to encapsulate the highly API-dependent device initialization and transformation procedures.

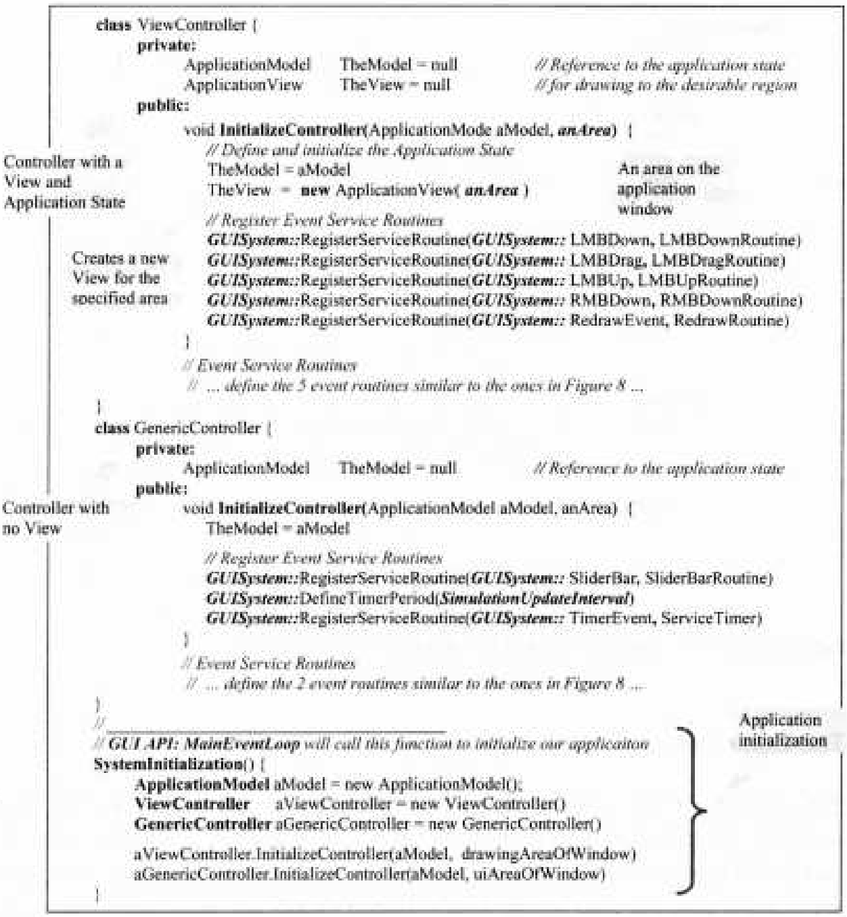
```
    class Viewport {
            private:
              // An area on application window for drawing.
              // Actual implemenation of the viewport is GraphicsAPI dependent.
            public:
              void          EraseViewport()
                               // Erase the area on the application window
              void          ActivateViewportForDrawing()
                               // All subsequent Graphics API draw commands
                               //  will show up on this viewport
 }
 class ApplicationView {
            private:
              // a view's private state information
              Viewport     TargetDrawArea
                               //  An area of the application main window that
                               // this view will be drawing to
            public:
              void          DeviceToWorldXform( inputDevicePoint, outputModelPoint)
                               // transform the input device coordinate point to
                               // output point in a coordinate system that the model understands

              void          DrawView( ApplicationModel TheModel)
                               // Erase and activate the TargetDrawArea and then
                               // Sets up transformation for TheModel
                               // calls TheModel.DrawApplicationState() to draw all the balls.
    }
```

Figure 18.12. The view component of the ball shooting program.

## The Controllers

We can improve the solution of Figure 18.8 to better support the specified functionality of the ball shooting program. Recall that the application window depicted in Figure 18.2 has two distinct regions for interpreting events: the upper application drawing area where mouse button events are associated with defining/selecting the HeroBall and the lower GUI element area where mouse button events on the GUI elements have different meanings (e.g., mouse button events on the slider bars generate SliderBarChange events, etc.). We also notice that the upper application drawing area is the exact same area where the ApplicationView must direct the drawings of the ApplicationModel state.

Figure 18.13 introduces two types of controller classes: a *ViewController* and a *GenericController*. Each controller class is dedicated to receiving input events from the corresponding region on the application window. The ViewController creates an ApplicationView during initialization such that the view can be tightly paired for drawing of the ApplicationModel state in the same area. In addition, the ViewController class also defines the appropriate mouse event service routines to support the interaction with the HeroBall. The GenericController is meant to contain GUI elements for interacting with the application state.



```
class ViewController {
      private:
            ApplicationModel      TheModel = null          // Reference to the application state
            ApplicationView       TheView = null           // for drawing to the desirable region
      public:
            void InitializeController(ApplicationMode aModel, anArea)  {
               // Define and initialize the Application State
               TheModel = aModel
               TheView  =  new ApplicationView( anArea )
               // Register Event Service Routines
               GUISystem::RegisterServiceRoutine(GUISystem:: LMBDown, LMBDownRoutine)
               GUISystem::RegisterServiceRoutine(GUISystem:: LMBDrag, LMBDragRoutine)
               GUISystem::RegisterServiceRoutine(GUISystem:: LMBUp, LMBUpRoutine)
               GUISystem::RegisterServiceRoutine(GUISystem:: RMBDown, RMBDownRoutine)
               GUISystem::RegisterServiceRoutine(GUISystem:: RedrawEvent, RedrawRoutine)
            }
            // Event Service Routines
             // ... define the 5 event routines similar to the ones in Figure 8 ...
}
class GenericController {
      private:
            ApplicationModel      TheModel = null          // Reference to the application state
      public:
            void InitializeController(ApplicationModel aModel, anArea)  {
               TheModel = aModel

               // Register Event Service Routines
               GUISystem::RegisterServiceRoutine(GUISystem:: SliderBar, SliderBarRoutine)
               GUISystem::DefineTimerPeriod(SimulationUpdateInterval)
               GUISystem::RegisterServiceRoutine(GUISystem:: TimerEvent, ServiceTimer)
            }
            // Event Service Routines
             // ... define the 2 event routines similar to the ones in Figure 8 ...
}
//________________________________________
// GUI API: MainEventLoop will call this function to initialize our applicaiton
SystemInitialization() {
      ApplicationModel aModel = new ApplicationModel();
      ViewController      aViewController = new ViewController()
      GenericController aGenericController = new GenericController()

      aViewController.InitializeController(aModel,  drawingAreaOfWindow)
      aGenericController.InitializeController(aModel, uiAreaOfWindow)
}
```

Controller with a View and Application State
Creates a new View for the specified area
An area on the application window
Controller with no View
Application initialization

**Figure 18.13.** The controller component of the ball shooting program.

The bottom of Figure 18.13 illustrates that the GUI API MainEventLoop will still call the *SystemInitialization*() function to initialize the application. In this case, we create one instance each of ViewController and GenericController. The ViewController is initialized to monitor mouse button events in the drawing area of the application window (e.g., LMB click to define HeroBall), while the GenericController is initialized to monitor the GUI element state changes (e.g., LMB dragging of a slider bar). Notice that the service of the timer event is global to the entire application and should be defined in only one of the controllers (either one will do).

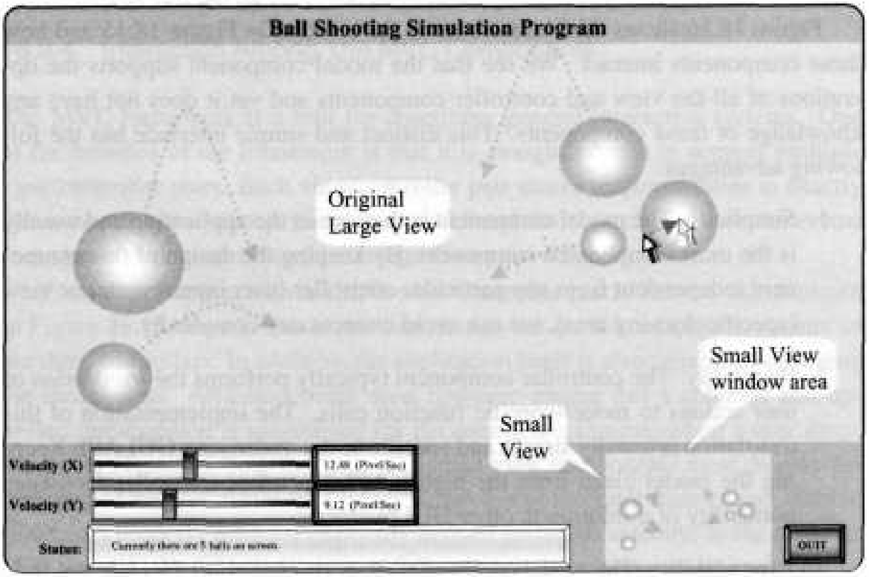
In practice, the GUI API MainEventLoop *dispatches* events to the controllers based on the *context of the event*. The context of an event is typically defined by the location of the mouse pointer or the current *focus* of the GUI element (i.e., which element is active). The application is responsible for creating a controller for any region on the window that it will receive events directly from the GUI API.

### 18.3.3 Using the MVC to Expand the Ball Shooting Program

One interesting characteristic of the MVC solution presented in Section 18.3.2 is that the model component does not have any knowledge of the view or the controller components. This clean interface allows us to expand our solution by inserting additional view/controller pairs.

For example, Figure 18.14 shows an extension to the ball shooting program given in Figure 18.2. It has an additional small view in the UI (user interface) area next to the quit button. The small view is exactly the same as the *original large view*, except that it covers a smaller area on the application window.

Figure 18.15 shows that, with our MVC solution design, we can implement the small view by creating a new instance of ViewController (an additional ApplicaitonView will be created by the ViewController) for the desired application window area. Notice that the GenericController's window area actually contains



Original Large View

Small View window area

Small View

Velocity (X): 12.88 (Pixel/Sec)

Velocity (Y): 9.12 (Pixel/Sec)

Status: Currently there are 5 balls on screen.

**Figure 18.14.** The ball shooting program with large and small views.

```
//_____________________________________________
// GUI API: MainEventLoop will call this function to initialize our applicaiton
SystemInitialization() {
      ApplicationModel   aModel = new ApplicationModel();
      ViewController     aLargeViewController = new ViewController()
      GenericController  aGenericController = new GenericController()

      aLargeViewController.InitializeController(aModel, drawingAreaOfWindow)
      aGenericController.InitializeController(aModel, uiAreaOfWindow)

      ViewController aSmallViewController = new ViewController()
      aSmallViewController.InitializeController(aModel, smallViewDrawingArea)
   }
```

New instance of ViewController (and ApplicationView)

**Figure 18.15.** Implementing the small view for the ball shooting program.

the area of the small ViewController. When a user event is triggered in this area, the "top-layer" controller (the visible one) will receive the event. After the initialization, the new small view will behave in exactly the same manner as the original large view.

For simplicity, Figure 18.14 shows two identical view/controller pairs. In general, a new view/controller pair is created to present a different visualization of the application state. For example, with slight modifications to the view component's transformation functionality, the large view of Figure 18.14 can be configured into a *zoom view* and the small view can be configured into a *work view*, where the zoom view can zoom into different regions (e.g., around the HeroBall) and the work view can present the entire application space (e.g., all the free falling balls).

Figure 18.16 shows the components of the solution in Figure 18.15 and how these components interact. We see that the model component supports the operations of all the view and controller components and yet it does not have any knowledge of these components. This distinct and simple interface has the following advantages:

1. Simplicity. The model component is the core of the application and usually is the most complicated component. By keeping the design of this component independent from any particular controller (user input/events) or view (specific drawing area), we can avoid unnecessary complexity.

2. Portability. The controller component typically performs the *translation* of user actions to model-specific function calls. The implementation of this translation is usually simple and specific to the underlying GUI API. Keeping the model clean from the highly API-dependent controller facilitates portability of a solution to other GUI platforms.

3. Expandability. The model component supports changing of its internal state and understands how to draw its contents. As we have seen (Figures 18.15

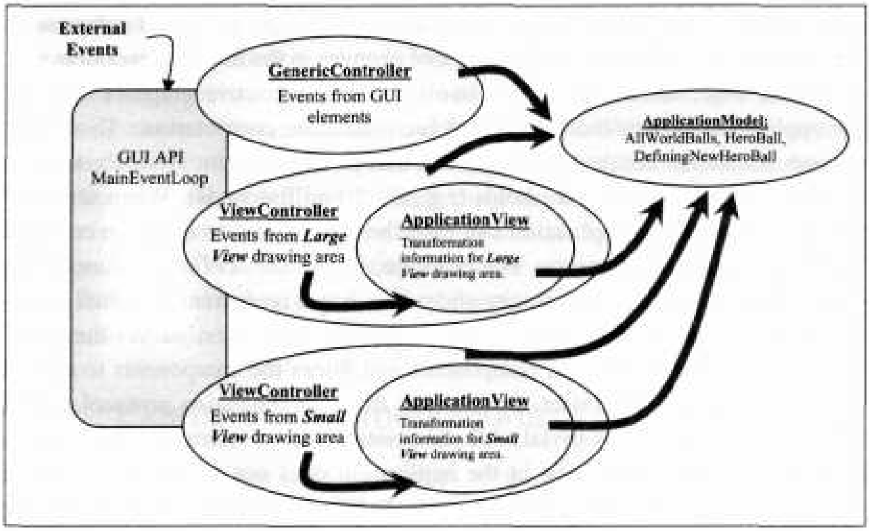

External Events
GenericController
Events from GUI elements
ApplicationModel:
AllWorldBalls, HeroBall, DefiningNewHeroBall
GUI API
MainEventLoop
ViewController
Events from *Large View* drawing area
ApplicationView
Transformation information for *Large View* drawing area.
ViewController
Events from *Small View* drawing area
ApplicationView
Transformation information for *Small View* drawing area.

**Figure 18.16.** Components of the ball shooting program with small view.

and 18.16), this means that it is straightforward to add new view/controller pairs to increase the interactivity of the application.

### 18.3.4 Interaction Among The MVC Components

The MVC framework is a tool for describing general interactive systems. One of the beauties of the framework is that it is straightforward to support multiple view/controller pairs. Each view/controller pair shares responsibilities in exactly the same way: the view *presents* the model and the controller allows the events (user-generated or otherwise) to change the model component.

For an application with multiple view/controller pairs, like the one depicted in Figure 18.16, we see that a user can change the model component via any of the three controllers. In addition, the application itself is also capable of changing the model state. All components must however, ensure that a coherent and up-to-date presentation is maintained for the user. For example, when a user drags out a new HeroBall, both the large and small view components must display the dragging of the ball, while the GenericController component must ensure that the slider bars properly echo the implicitly defined HeroBall velocity. In the classical MVC model, the coherency among different components is maintained with an elaborate protocol (e.g., via the observer design pattern). Although the classical

MVC model works very well, the elaborate protocol requires that all components communicate or otherwise to *keep track* of changes in the model component.

In our case, and in the case of most modern interactive graphics systems, the application defines the timer event for simulation computation. To support *smooth* simulation results, we have seen that the timer event typically triggers within real-time response thresholds (e.g., 20–50 milliseconds). When servicing the timer events, our application can take the opportunity to maintain coherent states among all components. For example, in the *ServiceTimer*() function in Figure 18.8, we update the velocity slider bars based on current HeroBall velocity. In effect, during each timer event service, the application *pushes* the up-to-date model information to all components and *forces* the components to refresh their presentation for the user. In this way, the communication protocol among the components becomes trivial. All components keep a reference to the model, and each view/controller pair in the application does not need to be aware of the existence of other view/controller pairs. In between periodic timer events, the user's asynchronous events change the model. These changes are only made in the model component, and no other components in the application need to be aware of the changes. During the periodic timer service, besides computing the model's simulation update, all components poll the model for up-to-date state information. For example, when the user clicks and drags with the left mouse button pressed, a new HeroBall will be defined in the model component. During this time, the large and small view components will not display the new HeroBall, and the velocity slider bars will not show the new HeroBall's velocity. These components will get and display up-to-date HeroBall information only during the application timer event servicing. Since the timer event is triggered more than 30 times per second, the user will observe a smooth and up-to-date application state in all components at all times.

### 18.3.5 Applying the MVC Concept

The MVC framework is applicable to general interactive systems. As we have seen in this section, interactive systems with the MVC framework result in clearly defined component behaviors. In addition, with clearly defined interfaces among the components, it becomes straightforward to expand the system with additional view/controller pairs.

An *interactive system* does not need to be an elaborate software application. For example, the slider bar is a fully functional interactive system. The model component contains a current *value* (typically a floating point number), the view component presents this value to the user, and the controller allows the user to interactively change this value. A typical view component draws rectangular icons (bar and knobs) representing the current value in the model component, while the controller component typically supports mouse down and drag events to interactively change the value in the model component. With this understanding, it becomes straightforward to expand the system with additional view/controller pairs. For example, in our ball shooting program, the slider bars have an additional view component where the numeric value of the model is displayed. In this case, there is no complementary controller component defined for the numeric view; an example complementary controller would allow the user to type in numeric values.

## 18.4 Example Implementations

Figure 18.17 shows two implementations of the solution presented in Section 18.3.3. The version on the left is based on OpenGL and FLTK, while the version to the right is based on D3D and MFC. In this section, we present the details of these two implementations. The lessons we want to learn are that (a) a proper MVC solution framework should be independent from any implementation and (b) a well designed implementation should be realizable based on and/or easily ported to any suitable API.

Before examining the details of each implementation, we will develop some understanding for working with modern GUI and graphics APIs.

### 18.4.1 Working with GUI APIs

Building the Graphical User Interface (GUI) of an application involves two distinct steps. The first step is to *design the layout* of the user interface system. In this step, an application developer places GUI elements (e.g., buttons, slider bars, etc.)

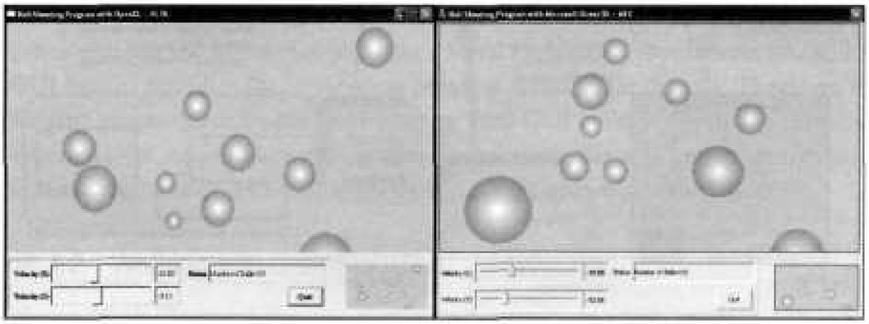


Figure 18.17. Ball shooting programs with OpenGL+FLTK and D3D+MFC.

in an area that represents the application window. The GUI elements are typically two-dimensional graphical artifacts (e.g., a 3D looking icon representing a slider bar). The goal of this first step is to arrange these graphical artifacts to achieve user friendliness and maximum usability (e.g., what is the best place/color/size for the slider bar, etc.). The second step in building a GUI for an application is to *semantically link* the GUI elements to the functionality of the application (e.g., update HeroBall velocity when the slider bar is dragged). In this step, an application developer builds the code for the necessary functionality (e.g., code for changing HeroBall velocity) and *registers* this code with the on-screen graphical artifacts (e.g., the slider bar). This is precisely the *event service registration* described in Section 18.2.2.

Modern GUI APIs support the building of a graphical user interface with a *GUI builder*. A GUI builder is an interactive graphical editor that allows its user to interactively place and manipulate the appearances of GUI elements. In addition, the GUI builder assists the application developer to compose or generate service routines and links those service routines to the events generated by the GUI elements.

Figure 18.18 illustrates the mechanism by which the GUI builder (in the middle of the figure) links the graphical user interface front-end (left side of the the figure) to the user-developed program code (right side of the figure). The patterned ellipse, the GUI Builder, is shown in the middle of Figure 18.18 The arrow pointing left towards the application (*A Simple Program*) indicates that the application developer works with the GUI builder to design the layout of the application (e.g., where to place the button or the status echo area). The arrows pointing from the GUI builder toward the *MainEventLoop* and *Event Service Linkage* mod-

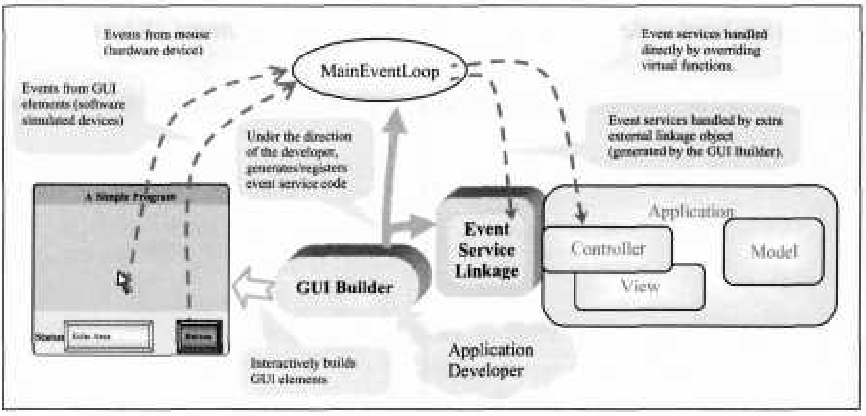

Events from mouse (hardware device)
MainEventLoop
Event services handled directly by overriding virtual functions.
Events from GUI elements (software simulated devices)
Under the direction of the developer, generates/registers event service code
Event services handled by extra external linkage object (generated by the GUI Builder).
A Simple Program
Event Service Linkage
Application
Controller
Model
View
GUI Builder
Status
Echo Area
Button
Interactively builds GUI elements
Application Developer

Figure 18.18. Working with a GUI API.

ules indicate that the GUI builder is capable of generating programming code to register event services. In Figure 18.18, there are two dotted connections between the mouse and the button GUI element through the MainEventLoop module to the event service linkage and the application controller modules. These two connections represent the two different mechanisms with which GUI APIs support event services:

1. External Service Linkage. Some GUI builders generate extra program modules (e.g., in the form of source code files) with code fragments supplied by the application developer to semantically link the GUI elements to the application functionality. For example, when the "button" of "A Simple Program" is clicked, the GUI builder ensures that a function in the "Event Service Linkage" module will be called. It is the application developer's responsibility to insert code fragments into this function to implement the required action.
2. Internal Direct Code Modification. Some GUI builders insert linkage programming code directly into the application source code. For example, the GUI builder modifies the source code of the application's controller class and inserts a new function to be called when the "button" of "A Simple Program" is clicked. Notice that the GUI Builder only inserts an empty function; the application developer is still responsible for filling in the details of this new function.

The advantage of an external service linkage mechanism is that the GUI builder only has minimal knowledge of the application source code. This provides a simple and flexible development environment where the developer is free to organize the source code structure, variable names, etc., in any appropriate way. However, the externally generated programming module implies a loosely integrated environment. For example, to modify the "button" behavior of "A Simple Program," the application developer must invoke the GUI builder, modify code fragments, and re-generate the external program module. The Internal Direct Code Modification mechanism in contrast provides a better integrated environment where the GUI builder modifies the application program source code directly. However, to support proper "direct code modification," the GUI builder must have intimate knowledge of, and often places severe constraints on, the application source code system (e.g., source code organization, file names, variable names, etc.).

### 18.4.2 Working with Graphics APIs

Figure 18.19 illustrates that one way to understand a modern graphics API is by considering the API as a functional interface to the underlying graphics hardware.

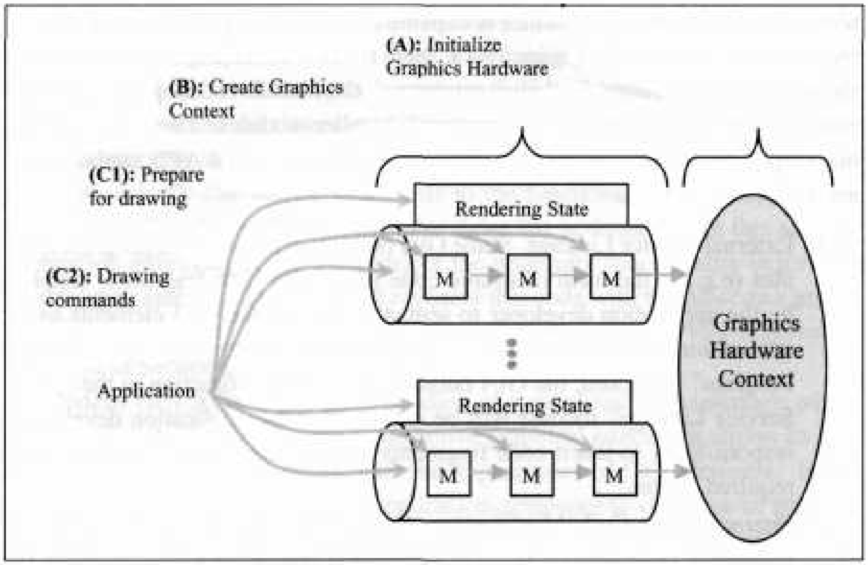

(A): Initialize Graphics Hardware
(B): Create Graphics Context
(C1): Prepare for drawing
(C2): Drawing commands
Application
Rendering State
M
M
M
Rendering State
M
M
M
Graphics Hardware Context

**Figure 18.19.** Working with a graphics API.

It is convenient to consider this functional interface as consisting of two stages: *Graphics Hardware Context (GHC)* and *Graphics Device Context (GDC)*.

**Graphics Hardware Context (GHC).** This stage is depicted as the vertical ellipse on the right of Figure 18.19. We consider the GHC as a configuration which wraps over the hardware video display card. An application creates a GHC for each unique configuration (e.g., depth of frame buffer or z-buffer, etc.) of the hardware video card(s). Many Graphics Device Contexts (see below) can be connected to each GHC to support drawing to multiple on-screen areas from the same application.

**Graphics Device Context (GDC).** This stage is depicted as a cylindrical *pipe* in Figure 18.19. The multiple pipes in the figure illustrates that an application can create multiple GDCs to connect to the same GHC. Through each GDC, an application can draw to distinct areas on the application window. To properly support this functionality, each GDC represents a complete *rendering state*. A rendering state encompasses all the information that affects the final appearance of an image. This includes primitive attributes, illumination parameters, coordinate transformations, etc. Examples of primitive attributes are color, size, pattern, etc., while examples of illumination parameters include light position, light color, surface material properties, etc. Graphics APIs typically support coordinate transformation with a series of two or three matrix processors. In Figure 18.19, the "M" boxes inside the GDC pipes are the matrix processors. Each matrix processor has a transformation matrix and transforms input vertices using this matrix. Since these processors operate in series, together they are capable of implementing multi-stage coordinate space transformations (e.g., object to world, world to eye, and eye to projected space). The application must load these matrix processors with appropriate matrices to implement a desired transformation.

With this understanding, Figure 18.19 illustrates that to work with a graphics API, an application will

(A) initialize one or more GHCs. Each GHC represents a unique configuration of the graphics video card(s). In typical cases, one GHC is initialized and configured to be shared by the entire application.

(B) create one or more GDCs. Each GDC supports drawing to distinct areas on the application window. For example, an application might create a GDC for each view component in an application.

(C) draw using a GDC. An application draws to a desired window area via the corresponding GDC. Referring to Figure 18.19, an application sets the rendering state (C1) and then issues drawing commands to the GDC (C2). Setting of the rendering state involves setting of all relevant primitive and illumination attributes and computing/loading appropriate transformation matrices into the matrix processors. A drawing command is typically a series of vertex positions accompanied by instructions on how to interpret the vertices (e.g., two vertex positions and an instruction that these are end points of a line).

In practice, modern graphics APIs are highly configurable and support many abstract programming modes. For example, Microsoft's Direct3D supports a drawing mode where the matrix processors can be by-passed entirely (e.g., when vertices are pre-transformed).

### 18.4.3 Implementation Details

Figure 18.20 shows the design of our implementation for the solution presented in Section 18.3.3.[1] Here, the MainUIWindow object represents the entire ball shooting program. This object contains the GUI elements (slider bars, quit button, etc.), the model (application state), and two instances of view/controller pairs (one each for LargeView and SmallView).

[1] Source code for this section can be found at http://faculty.washington.edu/ksung/fcg2/ball.tar.zip

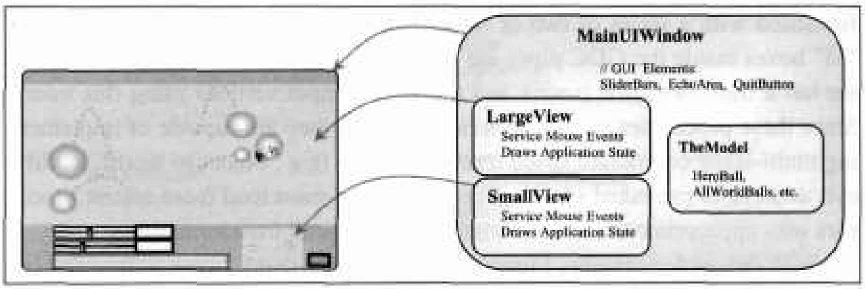


**Figure 18.20.** Implementation of the ball shooting program with two views.

## OpenGL with FLTK

Figure 18.21 shows a screen shot of *Fluid*, FLTK's GUI builder, during the construction of the GUI for the ball shooting program. In the lower-right corner of Figure 18.21, we see that (A) Fluid allows an application developer to interactively place graphical representations of GUI elements (3D-looking icons); (B) is an area representing the application window. In addition (C), the application developer can interactively select each GUI element to define its physical appearances (color, shape, size, etc.). In the lower-left corner of Figure 18.21, we see that (D) the application developer has the option to type in program fragments to service events generated by the corresponding GUI element. In this case, we can see that the developer must type in the program fragment for handling the X velocity slider bar events. Notice that this program fragment is separated from the rest of the program source code system and is associated with Fluid (the GUI builder). At the conclusion of the GUI layout design, Fluid generates new source code files to be included with the rest of the application development environment.

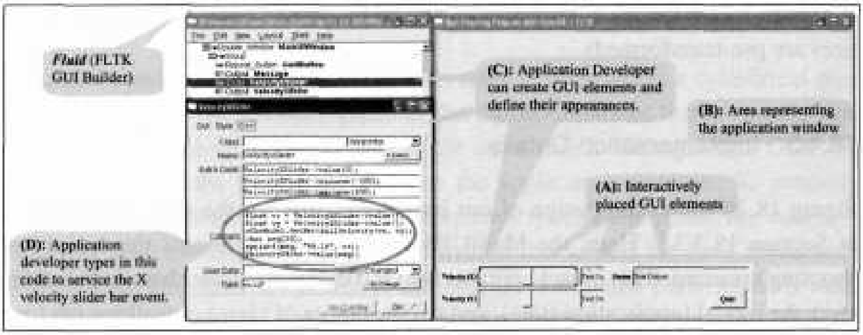

*Fluid* (FLTK GUI Builder)
(C): Application Developer can create GUI elements and define their appearances.
(B): Area representing the application window
(A): Interactively placed GUI elements
(D): Application developer types in this code to service the X velocity slider bar event.

**Figure 18.21.** Fluid: FLTK's GUI Builder.

```
// Forward declaration of  mouse event service routines
        void ServiceMouse(int button, int state, int x, int y); // service mouse button click
        void ServiceActiveMouse(int x, int y);                  // service mouse drag
class MainUIWindow {
        UserInterface       UI;              // This is Linkage Code generated by Fluid (GUI Builder)
                                             //      This object services events geneated by GUI  elements
        Model               *TheModel;       // The application State (Figure 11)
        FlGlutWindow        *LargeView;      // These are View/Controller pairs that understand graphics
        FlGlutWindow        *SmallView;      //      outputs (GDC) and mouse events (controller)

        MainUIWindow(Model *m)     {                                // The constructor
                TheModel = m;                                       // Sets the model ...
                LargeView = new FlGlutWindow(TheModel);             // Create LargeView
                LargeView->mouse = ServiceMouse;               // callback functions for service mouse events
                LargeView->motion = ServiceActiveMouse;
                // Create SmallView ... exactly the same as LargeView (not shown)
                glutTimeFunc( // set up timer and services )        // Set up timer ...
        }
};
```

**Figure 18.22.** MainUIWindow based on OpenGL and FLTK.

Since these source code files are controlled and generated by the GUI builder, the application developer must invoke the GUI builder in order to update/maintain the event service routines. In this way, FLTK implements external service linkage as described in Section 18.4.1. In our implementation, we instruct *Fluid* to create a *UserInterface* class (.h and .cpp files) for the integration with the rest of our application development environment.

Figure 18.22 shows the *MainUIWindow* implementation with OpenGL and FLTK. In this case, graphics operations are performed through OpenGL and user interface operations are supported by FLTK. As described, the *UserInterface* object in the MainUIWindow is created by Fluid for servicing GUI events. TheModel is the application state as detailed in Figure 18.11. The two FlGlutWindow objects are based on a predefined FLTK class designed specifically for supporting drawing with OpenGL. The constructor of MainUIWindow shows that the mouse event services are registered via a callback mechanism. As discussed in Section 18.2.4, the FLTK (Fast Light ToolKit) is an example of a light weight GUI API. Here, we see examples of using callback as a registration mechanism for receiving user events.

FlGlutWindow is a FLTK pre-defined Fl_Glut_Window class object (see Figure 18.23) designed specifically to support drawing with OpenGL. Each instance of a FlGlutWindow object is a combination of a controller (e.g., to receive mouse events) and a Graphics Device Context (GDC). We see that the *draw*() function first sets the rendering state (e.g., clear color and matrix values), including computing and programming the matrix processor (e.g., GL_PROJECTION), before calling TheModel to re-draw the application state.

```
// Fl_Glut_Window is a pure virtual class supplied by FLTK specifically for supporting
// windows with OpenGL output and for receiving mouse events.
class FlGlutWindow : public Fl_Glut_Window {
    FlGlutWindow(Model *m);              // Constructor
    Model     *TheModel;                 // The application state: initialized during construction time.
    float     WorldWidth, WorldHeight;   // World Space Dimension

    void HardwareToWorldPoint(int hwX, int hwY, float &wcX, float &wcY);
                                   // Transform mouse clicks (hwX, hwY) to World Cooridnate (wcX, wcY)

    virtual void draw() {                        // virtual function from Fl_Glut_Window for drawing
        glClearColor( 0.8f, 0.8f, 0.95f, 0.0f );
        glClear(GL_COLOR_BUFFER_BIT);            // Clearing the background color
        glMatrixMode(GL_PROJECTION);             // Programming the OpenGL's GL_PROJECTION
                glLoadIdentity( );               //      Matrix Processor to the propoer transfrotm
                gluOrtho2D(0.0f, WorldWidth, 0.0f, WorldHeight);
        TheModel->DrawApplicaitonState( );       // Drawing of the application state
    }
};
```

**Figure 18.23.** FlGlutWindow: OpenGL/FLTK view/controller pair.

## Direct3D with MFC

Figure 18.24 shows a screen shot of the MFC resource editor, MFC's GUI builder, during the construction of the ball shooting program. Similar to Fluid (Figure 18.21), in the middle of Figure 18.24, (A) we see that the resource editor also supports interactive designing of the GUI element layout in (B), an area representing the application window. Although the GUI builder interfaces operate differently, we observe that in (C), the MFC resource editor also supports the definition/modification of the physical appearance of GUI elements. However,

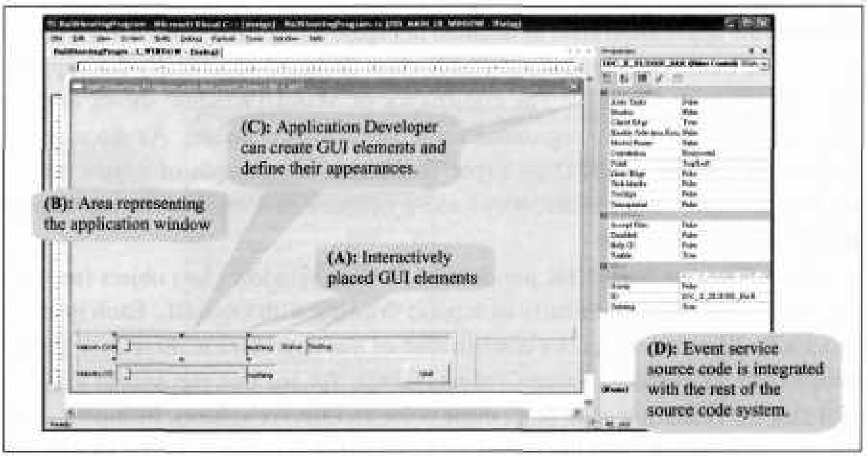
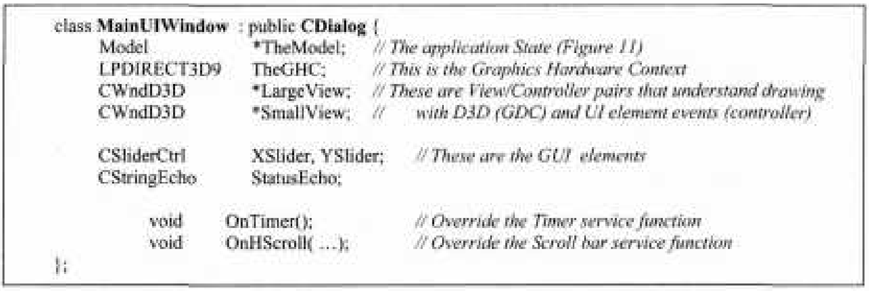

(C): Application Developer can create GUI elements and define their appearances.
(B): Area representing the application window
(A): Interactively placed GUI elements
(D): Event service source code is integrated with the rest of the source code system.

**Figure 18.24.** The MFC resource editor.



```
class MainUIWindow  : public CDialog {
        Model            *TheModel;     // The application State (Figure 11)
        LPDIRECT3D9      TheGHC;        // This is the Graphics Hardware Context
        CWndD3D          *LargeView;    // These are View/Controller pairs that understand drawing
        CWndD3D          *SmallView;    //      with D3D (GDC) and UI element events (controller)

        CSliderCtrl      XSlider, YSlider;    // These are the GUI  elements
        CStringEcho      StatusEcho;

                void     OnTimer();           // Override the Timer service function
                void     OnHScroll( ...);     // Override the Scroll bar service function
};
```

**Figure 18.25.** MainUIWindow based on Microsoft Direct3D and MFC.

unlike Fluid, the MFC resource editor is tightly integrated with the rest of the development environment. In this case, a developer can register for event services by inheriting or overriding appropriate service routines. The MFC resource editor automatically inserts code fragments into the application source code system. To support this functionality, the application source code organization is governed/shared with the GUI builder; the application developer is not entirely free to rename files/classes and/or to re-organize implementation source code file system structure. MFC implements internal direct code modification for event service linkage, as described in Section 18.4.1.

Figure 18.25 shows the *MainUIWindow* implementation with Direct3D and MFC. In this implementation, graphics operations are performed through Direct3D while user interface operations are supported by MFC. Once again, The-Model is the application state as detailed in Figure 18.11. *LPDIRECT3D9* is the Graphics Hardware Context (GHC) interface object. This object is created and initialized in the MainUIWindow constructor (not shown here). The two CWndD3D objects are defined to support drawing with Direct3D. We notice that one major difference between Figure 18.25 and Figure 18.22 is in the GUI element support. In Figure 18.25, we see that the GUI element objects (e.g., XSlider) and the corresponding service routines (e.g., OnHScroll()) are integrated into the MainUIWindow object. This is in contrast to the solution shown in Figure 18.22 where GUI elements are grouped into a separate object (e.g., the UserInterface object) with callback event service registrations. As discussed in Section 18.2.4, MFC is an example of a large commercial GUI API, where many event services are registered based on object-oriented function overrides (e.g., the OnHScroll() and OnTimer() functions).

CWndD3D is a sub-class of the MFC CWnd class (see Figure 18.26). CWnd is the base class designed for a generic MFC window. By sub-classing from this base class, CWndD3D can support all default window-related events (e.g., mouse

```
// CWnd is the MFC base class for all window objects. Here we subclass to create a D3D output
// window by including a D3D Graphics Device Context.
class CWndD3D: public CWnd {
   LPDIRECT3DDEVICE9  D3DDevice;     // This is the D3D Graphics Device Context (GDC)
   Model              *TheModel;     // The application state
   void InitD3D(LPDIRECT3D9);        // Create D3DDevice (GDC) to connect to GHC

   void RedrawView()    {            // Draws the Application State
            // Compute world coordinate to device transform
            D3DMATRIX transform = ComputeTransformation();
            D3DDevice->SetTransform(D3DTS_WORLD, &transform);
                    // Programming  the D3D_WORLD matrix with the computed transform matrix

            D3DDevice->Clear( bgColor, D3DCLEAR_TARGET);
            D3DDevice->BeginScene();
               TheModel->DrawApplicationState(D3DDevice);
            D3DDevice->EndScene();
            D3DDevice->Present();
   }

   void HardwareToWorldPoint(CPoint hwPt, float &wcX, float &wcY);
                              // Transform mouse clicks (hwPt) to world coordinate (wcX, wcY)

   void OnLButtonDown(CPoint hwPt);  // Override mouse button/drag service functions
   ⋮
};
```

**Figure 18.26.** CWndD3D: Direct3D/MFC view/controller pair.

events). The *LPDIRECT3DDEVICE9* object is the D3D Graphics Device Context (GDC) interface object. The *InitD3D*() function creates and initializes the GDC object and connects this object to the *LPDIRECT3D9* (GHC). In this way, a CWndD3D sub-class is a basic view/controller pair: it supports the view functionality with drawing via the D3D GDC and controller functionality with input via MFC. The *RedrawView*() function is similar to the *draw*() function of Figure 18.23 where we first set up the rendering state (e.g., bgColor and matrix), including programming the matrix processor (e.g., *D3DTS_WORLD*), before calling the model to draw itself.

In conclusion, we see that Figure 18.20 represents an implementation of the solution presented in Section 18.3.3 while Section 18.4.3 presented two versions of the implementation for Figure 18.20. Although the GUI Builder, event service registration, and actual API function calls are very different, the final programming source code structures are remarkably similar. In fact, the two versions share the exact same source code files for the *Model* class. In addition, although the drawing functions for *CirclePrimitive* are different for OpenGL and D3D, we were able to share the source code files for the rest of the primitive behaviors (e.g., set center/radius, travel with velocity, collide, etc.). We reaffirm our assertion that software framework, solution structures, and event implementations should be designed independent of any APIs.

## 18.5 Applying Our Results

We have seen that the event-driven programming model is well suited for designing and implementing programs that interact with users. In addition, we have seen that the modelview-controller framework is a convenient and powerful structure for organizing functional modules in an interactive graphics application. In developing a solution to the ball shooting program, we have demonstrated that knowledge from event-driven programming helps us design the controller component (e.g., handling of mouse events, etc.), computer graphics knowledge helps us design the view component (e.g., transformation and drawing of circles, etc.), while the model component is highly dependent upon the specific application (e.g., free falling and colliding circles). Our discussion so far has been based on a very simple example. We will now explore the applicability of the MVC framework and its implementation in real-world applications.

### 18.5.1 Example 1: PowerPoint

Figure 18.27 shows how we can apply our knowledge in analyzing and gaining insights into Microsoft PowerPoint,[2] a popular interactive graphics application. A screen shot of a slide creation session using the PowerPoint application is shown at the left of Figure 18.27. The right side of Figure 18.27 shows how we can apply the implementation framework to gain insights into the PowerPoint application. The MainUIWindow at the right of Figure 18.27 is the GUI window of

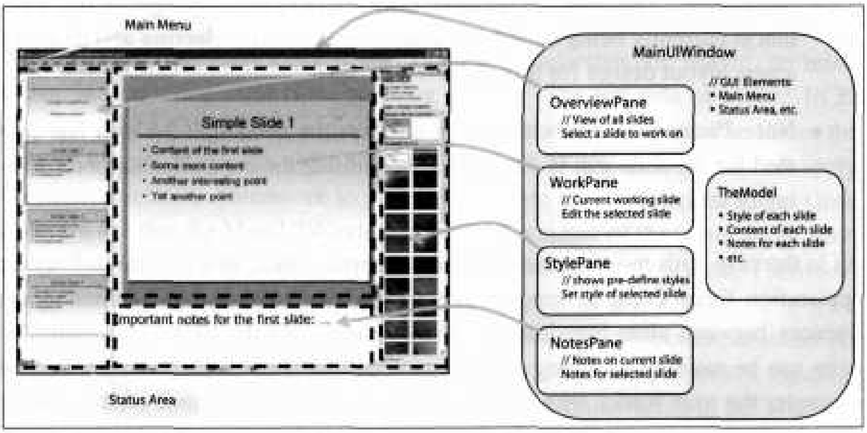


**Figure 18.27.** Understanding PowerPoint using the MVC implementation framework.

[2] Powerpoint is a registered trademark of Microsoft.

the entire application, and it contains the GUI elements that affect/echo the entire application state (e.g., main menu, status area, etc.). We can consider the MainUI-Window as the module that contains TheModel component and includes the four view/controller pairs.

Recall that TheModel is the state of the application and that this component contains all the data that the user interactively creates. In the case of PowerPoint, the user creates a collection of presentation slides, and thus TheModel contains all the information about these slides (e.g. layout design style, content of the slides, notes associated with each slide, etc.). With this understanding of TheModel component, the rest of the application can be considered as a convenient tool for presenting TheModel (the view) to the user and changing TheModel (the controller) by the user. In this way, these convenient tools are precisely the view/controller pairs (e.g., ViewController components from Figure 18.16).

In Figure 18.27, each of the four view/controller pairs (i.e., OverviewPane, WorkPane, StylePane, and NotesPane) presents, and supports changing of different aspects of TheModel component:

- **OverviewPane.** The view component displays multiple consecutive slides from all the slides that the user has created; the controller component supports user scrolling through all these slides and selecting one for editing.

- **WorkPane.** The view component displays the details of the slide that is currently being edited; the controller supports selecting and editing the content of this slide.

- **StylePane.** The view component displays the layout design of the slide that is currently being edited; the controller supports selecting and defining a new layout design for this slide.

- **NotesPane.** The view component displays the notes that the user has created for the slide that is currently being edited; the controller supports editing of this notes.

As is the case with most modern interactive applications, PowerPoint defines an application timer event to support user-defined animations (e.g., animated sequences between slide transitions). The coherency of the four view/controller pairs can be maintained during the servicing of this application timer event. For example, the user works with the StylePane to change the layout of the current slide in TheModel component. In the meantime, before servicing the next timer event, OverviewPane and WorkPane are not aware of the changes and display an out-of-date design for the current slide. During the servicing of the timer event,

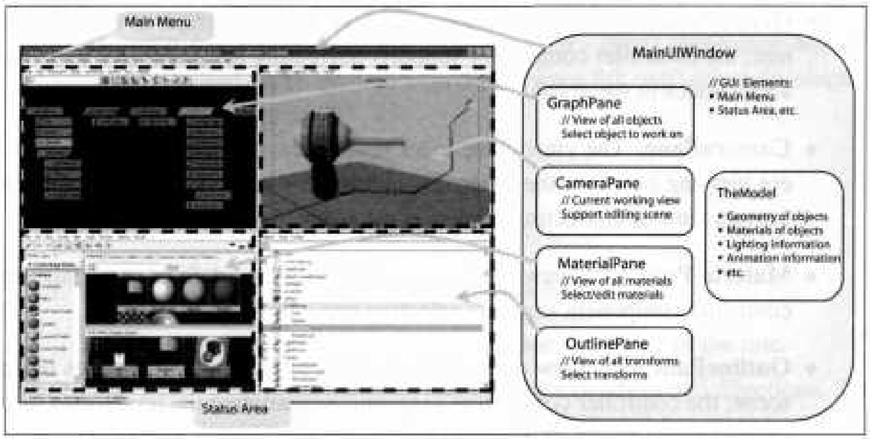


**Figure 18.28.** Understanding Maya with the MVC implementation framework.

the MainUIWindow forces all view/controller pairs to poll TheModel and refresh their contents. As discussed in Section 18.3.4, since the timer events are typically triggered more than 30 times in a second, the user is not be able to detect the brief out-of-date display and observes a consistent display at all times. In this way, the four view/controller pairs only need to keep a reference to TheModel component and do not need to have any knowledge of each other. Thus, it is straightforward to insert and delete view/controller pairs into/from the application.

### 18.5.2 Example 2: Maya

We now apply our knowledge in analyzing and understanding Maya[3], an interactive 3D modeling/animation/rendering system. The left side of Figure 18.28 shows a screen shot of Maya in a simple 3D content creation session. As in the case of Figure 18.27, the right side of Figure 18.28 shows how we can apply the implementation framework to gain insights into the Maya application. Once again we see that the MainUIWindow is the GUI window of the entire application containing GUI elements that affect/echo the entire application state, TheModel component, and all the view/controller pairs.

Since Maya is a 3D media creation system, TheModel component contains 3D content information (e.g. scene graph, 3D geometry, material properties, lighting, camera, animation, etc.). Once again, the rest of the components in the MainUIWindow are designed to facilitate the user's view and to change TheModel. Here is the functionality of the four view/controller pairs:

[3] Maya is a registered trademark of Alias.

- **GraphPane.** The view component displays the scene graph of the 3D content; the controller component supports navigating the graph and selecting scene nodes in the graph.
- **CameraPane.** The view component renders the scene graph from a camera viewing position; the controller component supports manipulating the camera view and selecting objects in the scene.
- **MaterialPane.** The view component displays all the defined materials; the controller component supports selecting and editing materials.
- **OutlinePane.** The view component displays all the transform nodes in the scene; the controller component supports manipulating the transforms (e.g. create/change parent-child relationships, etc.).

Once again, the coherency among the different view/controller pairs can be maintained while servicing the application timer events.

We do not speculate that PowerPoint or Maya is implemented according to our framework. These are highly sophisticated commercial applications and the underlying implementation is certainly much more complex. However, based on the knowledge we have gained from this chapter, we can begin to understand how to approach discussing, designing, and building such interactive graphics applications. Remember that *the important lesson we want to learn from this chapter* is how to organize the functionality of an interactive graphics application into components and understand how the components interact so that we can better understand, maintain, modify, and expand an interactive graphics application.

## 18.6 Notes

I first learned about the model view controller framework and event-driven programming from SmallTalk (Goldberg & Robson, 1989) (You may also want to refer to the SmallTalk web site (http://www.smalltalk.org/main/).) Both *Design Patterns—Elements of Reusable Object-Oriented Design* (Gamma, Helm, Johnson, & Vlissides, 1995) and *Pattern-Oriented Software Architecture* (Buschmann, Meunier, Rohnert, Sommerlad, & Stal, 1996) are excellent sources for finding out more about design patterns and software architecture frameworks in general. I recommend *3D Game Engine Architecture* (Eberly, 2004) as a good source for discussions on issues relating to implementing real-time graphics systems. I learned MFC and Direct3D mainly by referring to the online Microsoft Developer Network pages (http://msdn.microsoft.com). In addition, I find Prosise's book *Programming Windows with MFC* (Prosise, 1999) to be very helpful. I refer to the *OpenGL Programming Guide* (Shreiner et al., 2004), *Reference Manual* (Shreiner, 2004), and FLTK on-line help (http://www.fltk.org/) when developing my OpenGL/FLTK programs.

## 18.7 Exercises

1. Here is the specification for dragging out a line:

    - Left mouse button (LMB) clicks define the center of the line.
    - LMB drags out a line such that the line extends in two directions. The first direction extends from the center (LMB click) position toward the current mouse position. The second direction extends in the opposite direction from the first with exactly the same length.
    - Right mouse button (RMB) click-drag moves the line such that the center of the line follows the current mouse position.

    (a) Follow the steps outlined in Section 18.2.3 and design an event-driven programming solution for this specification.

    (b) Implement your design with FLTK and OpenGL.

    (c) Implement your design with MFC and Direct-3D.

    Notice that in this case the useful application internal state information (the center position of the line) and the drawing presentation requirements (end points of the line) do not coincide exactly. When defining the application state, we should pay attention to what is the most important and convenient information to store in order to support the specified functionality.

2. For the line defined in Exercise 1, define a velocity that is the same as the slope of the line: once created, the line will travel along the direction defined by its slope. Use the length of the line as the speed. (Note that longer lines travel faster than shorter lines).

3. Here is the specification for dragging out a rectangle:

    - LMB click defines the center of the rectangle.
    - LMB drag out a rectangle such that the rectangle extends from the center position and one of the corner positions of the rectangle always follows the current mouse position.

- RMB click-drag moves the rectangle such that the center of the rectangle follows the current mouse position.

(a) Follow the steps outlined in Section 18.2.3 and design an event-driven programming solution for this specification.

(b) Implement your design with FLTK and OpenGL.

(c) Implement your design with MFC and Direct-3D.

4. For the rectangle in Exercise 3:

(a) Support the definition of a velocity similar to that of HeroBall velocity in Section 18.1: once created, the rectangle will travel along a direction that is the vector defined from its center towards the LMB release position.

(b) Design and implement collision between two rectangles (this is a simple 2D bound intersection check).

5. With results from Exercise 4, we can approximate a simple Pong game:

- The paddles are rectangles;
- A pong-ball is drawn as a circle but we will use the bounding square (a square that centers at the center of the circle, with dimension defined by the diameter of the circle) to approximate collision with the paddle.

Design and implement a single-player pong-game where a ball (circle) drops under gravitational force and the user must manipulate a paddle to bounce the ball upward to prevent it from dropping below the application window. You should:

(a) design a specification (similar to that of Section 18.1) for this pong game;

(b) follow the steps outlined in Section 18.2.3 to design an event-driven programming solution;

(c) implement your design either with OpenGL or Direct-3D.

6. Extend the ApplicationView in Figure 18.12 to include functionality for setting a world coordinate window bound. The world coordinate window bound defines a rectangular region in the world for displaying in the Viewport. Define a method for setting the world coordinate window bound and modify the *ApplicaitonView::DeviceToWorldXform*() function to support transforming mouse clicks to world coordinate space.

7. Integrate your results from Exercise 6 into the two-view ball shooting program from Figure 18.14 such that the small view can be focused around the current HeroBall. When there is no current HeroBall, the small view should display nothing. When user LMB click-drags, or when user RMB selects a HeroBall, the small view's world coordinate window bound should center at the HeroBall center and include a region that is 1.5 times the HeroBall diameter.

# 19

# Light

In this chapter, we discuss the practical issues of measuring light, usually called *radiometry*. The terms that arise in radiometry may at first seem strange and have terminology and notation that may be hard to keep straight. However, because radiometry is so fundamental to computer graphics, it is worth studying radiometry until it sinks in. This chapter also covers *photometry*, which takes radiometric quantities and scales them to estimate how much "useful" light is present. For example, a green light may seem twice as bright as a blue light of the same power because the eye is more sensitive to green light. Photometry attempts to quantify such distinctions.

## 19.1 Radiometry

Although we can define radiometric units in many systems, we use *SI* (International System of Units) units. Familiar SI units include the metric units of *meter* ($m$) and *gram* ($g$). Light is fundamentally a propagating form of energy, so it is useful to define the SI unit of energy, which is the *joule* ($J$).

### 19.1.1 Photons

To aid our intuition, we will describe radiometry in terms of collections of large numbers of *photons*, and this section establishes what is meant by a photon in this context. For the purposes of this chapter, a photon is a quantum of light that has a position, direction of propagation, and a wavelength $\lambda$. Somewhat strangely, the SI unit used for wavelength is *nanometer* ($nm$). This is mainly for historical reasons, and $1\,\mathrm{nm} = 10^{-9}$ m. Another unit, the *angstrom*, is sometimes used, and one nanometer is ten angstroms. A photon also has a speed $c$ that depends only on the refractive index $n$ of the medium through which it propagates. Sometimes the frequency $f = c/\lambda$ is also used for light. This is convenient because unlike $\lambda$ and $c$, $f$ does not change when the photon refracts into a medium with a new refractive index. Another invariant measure is the amount of energy $q$ carried by a photon, which is given by the following relationship:

$$q = hf = \frac{hc}{\lambda}, \tag{19.1}$$

where $h = 6.63 \times 10^{-34}$ J s is Plank's Constant. Although these quantities can be measured in any unit system, we will use SI units whenever possible.

### 19.1.2 Spectral Energy

If we have a large collection of photons, their total energy $Q$ can be computed by summing the energy $q_i$ of each photon. A reasonable question to ask is "How is the energy distributed across wavelengths?" An easy way to answer this is to partition the photons into bins, essentially histogramming them. We then have an energy associated with an interval. For example, we can count all the energy between $\lambda = 500$ nm and $\lambda = 600$ nm and have it turn out to be 10.2 J, and this might be denoted $q[500, 600] = 10.2$. If we divided the wavelength interval into two 50 nm intervals, we might find that $q[500, 550] = 5.2$ and $q[550, 600] = 5.0$. This tells us there was a little more energy in the short wavelength half of the interval $[500, 600]$. If we divide into 25 nm bins, we might find $q[500, 525] = 2.5$, and so on. The nice thing about the system is that it is straightforward. The bad thing about it is that the choice of the interval size determines the number.

A more commonly used system is to divide the energy by the size of the interval. So instead of $q[500, 600] = 10.2$ we would have

$$Q_\lambda[500, 600] = \frac{10.2}{100} = 0.12\,\mathrm{J(nm)}^{-1}.$$

This approach is nice, because the size of the interval has much less impact on the overall size of the numbers. An immediate idea would be to drive the interval size $\Delta\lambda$ to zero. This could be awkward, because for a sufficiently small $\Delta\lambda$, $Q_\lambda$ will either be zero or huge depending on whether there is a single photon or no photon in the interval. There are two schools of thought to solve that dilemma. The first is to assume that $\Delta\lambda$ is small, but not so small that the quantum nature of light comes into play. The second is to assume that the light is a continuum rather than individual photons, so a true derivative $dQ/d\lambda$ is appropriate. Both ways of thinking about it are appropriate and lead to the same computational machinery. In practice, it seems that most people who measure light prefer small, but finite, intervals, because that is what they can measure in the lab. Most people who do theory or computation prefer infinitesimal intervals, because that makes the machinery of calculus available.

The quantity $Q_\lambda$ is called *spectral energy*, and it is an *intensive* quantity as opposed to an *extensive* quantity such as energy, length, or mass. Intensive quantities can be thought of as density functions that tell the density of an extensive quantity at an infinitesimal point. For example, the energy $Q$ at a specific wavelength is probably zero, but the spectral energy (energy density) $Q_\lambda$ is a meaningful quantity. A probably more familiar example is that the population of a country may be 25 million, but the population at a point in that country is meaningless. However, the population *density* measured in people per square meter is meaningful, provided it is measured over large enough areas. Much like with photons, population density works best if we pretend that we can view population as a continuum where population density never becomes granular even when the area is small.

We will follow the convention of graphics where spectral energy is almost always used, and energy is rarely used. This results in a proliferation of $\lambda$ subscripts if "proper" notation is used. Instead, we will drop the subscript and use $Q$ to denote spectral energy. This can result in some confusion when people outside of graphics read graphics papers, so be aware of this standards issue. Your intuition about spectral power might be aided by imagining a measurement device with an energy sensor that measures light energy $q$. If you place a colored filter in front of the sensor that allows only light in the interval $[\lambda - \Delta\lambda/2, \lambda + \Delta\lambda/2]$, then the spectral power at $\lambda$ is $Q = \Delta q/\Delta\lambda$.

### 19.1.3 Power

It is useful to estimate a rate of energy production for light sources. This rate is called *power*, and it is measured in *watts*, $W$, which is another name for *joules per second*. This is easiest to understand in a *steady state*, but because power is an intensive quantity (a density over time), it is well defined even when energy production is varying over time. The units of power may be more familiar, e.g., a 100-watt light bulb. Such bulbs draw approximately 100 J of energy each second. The power of the light produced will actually be less than 100 W because of heat loss, etc., but we can still use this example to help understand more about photons. For example, we can get a feel for how many photons are produced in a second by a 100 W light. Suppose the average photon produced has the energy of a $\lambda = 500$ nm photon. The frequency of such a photon is

$$f = \frac{c}{\lambda} = \frac{3 \times 10^8 \text{ ms}^{-1}}{500 \times 10^{-9} \text{ m}} = 6 \times 10^{14} \text{ s}^{-1}.$$

The energy of that photon is $hf \approx 4 \times 10^{-19}$ J. That means a staggering $10^{20}$ photons are produced each second, even if the bulb is not very efficient. This explains why simulating a camera with a fast shutter speed and directly simulated photons is an inefficient choice for producing images.

As with energy, we are really interested in *spectral power* measured in $\text{W}(\text{nm})^{-1}$. Again, although the formal standard symbol for spectral power is $\Phi_\lambda$, we will use $\Phi$ with no subscript for convenience and consistency with most of the graphics literature. One thing to note is that the spectral power for a light source is usually a smaller number than the power. For example, if a light emits a power of 100 W evenly distributed over wavelengths 400 nm to 800 nm, then the spectral power will be 100 W/400 nm = 0.25 $\text{W}(\text{nm})^{-1}$. This is something to keep in mind if you set the spectral power of light sources by hand for debugging purposes.

The measurement device for spectral energy in the last section could be modified by taking a reading with a shutter that is open for a time interval $\Delta t$ centered at time $t$. The spectral power would then be $\Delta Q/(\Delta t \Delta\lambda)$.

### 19.1.4 Irradiance

The quantity *irradiance* arises naturally if you ask the question "How much light hits this point?". Of course the answer is "none," and again we must use a density function. If the point is on a surface, it is natural to use area to define our density function. We modify the device from the last section to have a finite $\Delta A$ area sensor that is smaller than the light field being measured. The spectral irradiance $H$ is just the power per unit area $\Delta\Phi/\Delta A$. Fully expanded this is

$$H = \frac{\Delta q}{\Delta A \; \Delta t \Delta\lambda}. \tag{19.2}$$

Thus, the full units of irradiance are $\text{Jm}^{-2}\text{s}^{-1}(\text{nm})^{-1}$. Note that the SI units for radiance include inverse-meter-squared for area and inverse-nanometer for wavelength. This seeming inconsistency (using both nanometer and meter) arises because of the natural units for area and visible light wavelengths.

When the light is leaving a surface, e.g., when it is reflected, the same quantity as irradiance is called *radiant exitance*, $E$. It is useful to have different words for incident and exitant light, because the same point has potentially different irradiance and radiant exitance.

### 19.1.5 Radiance

Although irradiance tells us how much light is arriving at a point, it tells us little about the direction that light comes from. To measure something analogous to what we see with our eyes, we need to be able to associate "how much light" with a specific direction. We can imagine a simple device to measure such a quantity (Figure 19.1). We use a small irradiance meter and add a conical "baffler" which limits light hitting the counter to a range of angles with solid angle $\Delta\sigma$. The response of the detector is as follows:

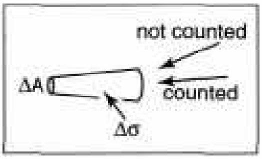


**Figure 19.1.** By adding a blinder that shows only a small solid angle $\Delta\sigma$ to the irradiance detector, we measure radiance.

$$\begin{aligned}\text{response} &= \frac{\Delta H}{\Delta\sigma} \\ &= \frac{\Delta q}{\Delta A\, \Delta\sigma\, \Delta t\, \Delta\lambda}.\end{aligned}$$

This is the spectral *radiance* of light travelling in space. Again, we will drop the "spectral" in our discussion and assume that it is implicit.

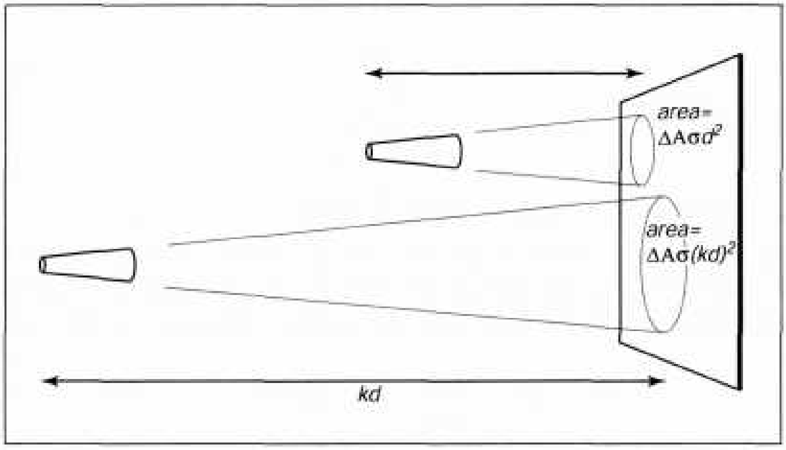


**Figure 19.2.** The signal a radiance detector receives does not depend on the distance to the surface being measured. This figure assumes the detectors are pointing at areas on the surface that are emitting light in the same way.

Radiance is what we are usually computing in graphics programs. A wonderful property of radiance is that it does not vary along a line in space. To see why this is true, examine the two radiance detectors both looking at a surface as shown in Figure 19.2. Assume the lines the detectors are looking along are close enough together that the surface is emitting/reflecting light "the same" in both of the areas being measured. Because the area of the surface being sampled is proportional to squared distance, and because the light reaching the detector is *inversely* proportional to squared distance, the two detectors should have the same reading.

It is useful to measure the radiance hitting a surface. We can think of placing the cone baffler from the radiance detector at a point on the surface and measuring the irradiance $H$ on the surface originating from directions within the cone (Figure 19.3). Note that the surface "detector" is not aligned with the cone. For this reason we need to add a cosine correction term to our definition of radiance:

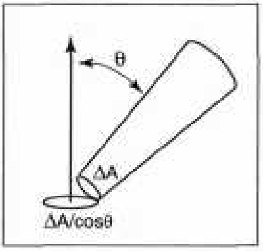


**Figure 19.3.** The irradiance at the surface as masked by the cone is smaller than that measured at the detector by a cosine factor.

$$\begin{aligned}\text{response} &= \frac{\Delta H}{\Delta\sigma\cos\theta}\\ &= \frac{\Delta q}{\Delta A\cos\theta\;\Delta\sigma\;\Delta t\;\Delta\lambda}.\end{aligned}$$

As with irradiance and radiant exitance, it is useful to distinguish between radiance incident at a point on a surface and exitant from that point. Terms for these concepts sometimes used in the graphics literature are *surface radiance* $L_s$ for the radiance of (leaving) a surface, and *field radiance* $L_f$ for the radiance incident at a surface. Both require the cosine term, because they both correspond to the configuration in Figure 19.3:

$$\begin{aligned}L_s &= \frac{\Delta E}{\Delta\sigma\cos\theta}\\ L_f &= \frac{\Delta H}{\Delta\sigma\cos\theta}.\end{aligned}$$

### Radiance and Other Radiometric Quantities

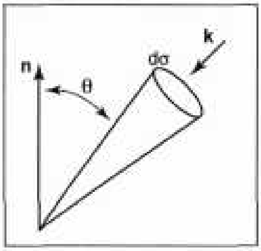


**Figure 19.4.** The direction **k** has a differential solid angle $d\sigma$ associated with it.

If we have a surface whose field radiance is $L_f$, then we can derive all of the other radiometric quantities from it. This is one reason radiance is considered the "fundamental" radiometric quantity. For example, the irradiance can be expressed as

$$H = \int_{\text{all }\mathbf{k}} L_f(\mathbf{k})\;\cos\theta\;d\sigma.$$

This formula has several notational conventions that are common in graphics that make such formulae opaque to readers not familiar with them (Figure 19.4). First, **k** is an incident direction and can be thought of as a unit vector, a direction, or a $(\theta, \phi)$ pair in spherical coordinates with respect to the surface normal. The direction has a differential solid angle $d\sigma$ associated with it. The field radiance is potentially different for every direction, so we write it as a function $L(\mathbf{k})$.

As an example, we can compute the irradiance $H$ at a surface that has constant field radiance $L_f$ in all directions. To integrate, we use a classic spherical coordinate system and recall that the differential solid angle is

$$d\sigma \equiv \sin\theta \; d\theta \; d\phi,$$

so the irradiance is

$$\begin{aligned} H &= \int_{\phi=0}^{2\pi} \int_{\theta=0}^{\frac{\pi}{2}} L_f \; \cos\theta \sin\theta \; d\theta \; d\phi \\ &= \pi L_f. \end{aligned}$$

This relation shows us our first occurrence of a potentially surprising constant $\pi$. These factors of $\pi$ occur frequently in radiometry and are an artifact of how we chose to measure solid angles, i.e., the area of a unit sphere is a multiple of $\pi$ rather than a multiple of one.

Similarly, we can find the power hitting a surface by integrating the irradiance across the surface area:

$$\Phi = \int_{\text{all } \mathbf{x}} H(\mathbf{x}) dA,$$

where $\mathbf{x}$ is a point on the surface, and $dA$ is the differential area associated with that point. Note that we don't have special terms or symbols for incoming versus outgoing power. That distinction does not seem to come up enough to have encouraged the distinction.

### 19.1.6 BRDF

Because we are interested in surface appearance, we would like to characterize how a surface reflects light. At an intuitive level, for any incident light coming from direction $\mathbf{k}_i$, there is some fraction scattered in a small solid angle near the outgoing direction $\mathbf{k}_o$. There are many ways we could formalize such a concept, and not surprisingly, the standard way to do so is inspired by building a simple measurement device. Such a device is shown in Figure 19.5, where a small light source is positioned in direction $\mathbf{k}_i$ as seen from a point on a surface, and a detector is placed in direction $\mathbf{k}_o$. For every directional pair $(\mathbf{k}_i, \mathbf{k}_o)$, we take a reading with the detector.

Now we just have to decide how to measure the strength of the light source and make our reflection function independent of this strength. For example, if we

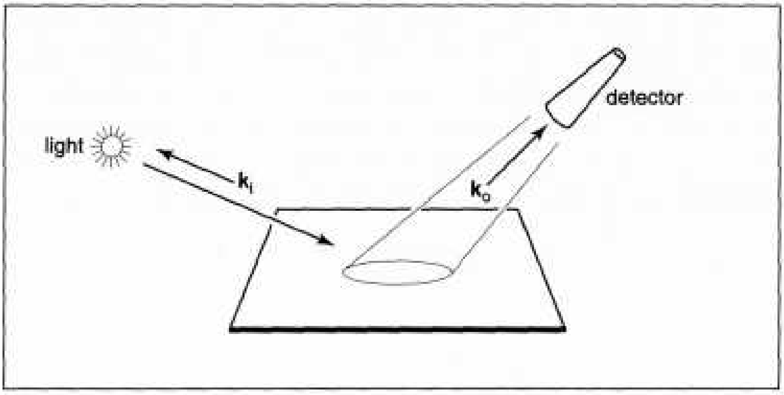


**Figure 19.5.** A simple measurement device for directional reflectance. The positions of light and detector are moved to each possible pair of directions. Note that both $\mathbf{k}_i$ and $\mathbf{k}_o$ point away from the surface to allow reciprocity.

replaced the light with a brighter light, we would not want to think of the surface as reflecting light differently. We could place a radiance meter at the point being illuminated to measure the light. However, for this to get an accurate reading that would not depend on the $\Delta\sigma$ of the detector, we would need the light to subtend a solid angle bigger than $\Delta\sigma$. Unfortunately, the measurement taken by our roving radiance detector in direction $\mathbf{k}_o$ will also count light that comes from points outside the new detector's cone. So this does not seem like a practical solution.

Alternatively, we can place an irradiance meter at the point on the surface being measured. This will take a reading that does not depend strongly on subtleties of the light source geometry. This suggests characterizing reflectance as a ratio:

$$\rho = \frac{L_s}{H},$$

where this fraction $\rho$ will vary with incident and exitant directions $\mathbf{k}_i$ and $\mathbf{k}_o$, $H$ is the irradiance for light position $\mathbf{k}_i$, and $L_s$ is the surface radiance measured in direction $\mathbf{k}_o$. If we take such a measurement for all direction pairs, we end up with a 4D function $\rho(\mathbf{k}_i, \mathbf{k}_o)$. This function is called the *bidirectional reflectance distribution function* (BRDF). The BRDF is all we need to know to characterize the directional properties of how a surface reflects light.

### Directional Hemispherical Reflectance

Given a BRDF it is straightforward to ask "What fraction of incident light is reflected?" However, the answer is not so easy; the fraction reflected depends on the directional distribution of incoming light. For this reason, we typically only set a fraction reflected for a fixed incident direction $\mathbf{k}_i$. This fraction is called the *directional hemispherical reflectance*. This fraction, $R(\mathbf{k}_i)$ is defined by

$$R(\mathbf{k}_i) = \frac{\text{power in all outgoing directions } \mathbf{k}_o}{\text{power in a beam from direction } \mathbf{k}_i}.$$

Note that this quantity is between zero and one for reasons of energy conservation. If we allow the incident power $\Phi_i$ to hit on a small area $\Delta A$, then the irradiance is $\Phi_i/\Delta A$. Also, the ratio of the incoming power is just the ratio of the radiance exitance to irradiance:

$$R(\mathbf{k}_i) = \frac{E}{H}.$$

The radiance in a particular direction resulting from this power is by the definition of BRDF:

$$\begin{aligned} L(\mathbf{k}_o) &= H\rho(\mathbf{k}_i, \mathbf{k}_o) \\ &= \frac{\Phi_i}{\Delta A}. \end{aligned}$$

And from the definition of radiance, we also have

$$L(\mathbf{k}_o) = \frac{\Delta E}{\Delta\sigma_o \cos\theta_o},$$

where $E$ is the radiant exitance of the small patch in direction $\mathbf{k}_o$. Using these two definitions for radiance we get

$$H\rho(\mathbf{k}_i, \mathbf{k}_o) = \frac{\Delta E}{\Delta\sigma_o \cos\theta_o}.$$

Rearranging terms, we get

$$\frac{\Delta E}{H} = \rho(\mathbf{k}_i, \mathbf{k}_o)\Delta\sigma_o \cos\theta_o.$$

This is just the small contribution to $E/H$ that is reflected near the particular $\mathbf{k}_o$. To find the total $R(\mathbf{k}_i)$, we sum over all outgoing $\mathbf{k}_o$. In integral form this is

$$R(\mathbf{k}_i) = \int_{\text{all } \mathbf{k}_o} \rho(\mathbf{k}_i, \mathbf{k}_o) \cos\theta_o \, d\sigma_o.$$

### Ideal Diffuse BRDF

An idealized diffuse surface is called *Lambertian*. Such surfaces are impossible in nature for thermodynamic reasons, but mathematically they do conserve energy. The Lambertian BRDF has $\rho$ equal to a constant for all angles. This means the surface will have the same radiance for all viewing angles, and this radiance will be proportional to the irradiance.

If we compute $R(\mathbf{k}_i)$ for a a Lambertian surface with $\rho = C$ we get

$$\begin{aligned} R(\mathbf{k}_i) &= \int_{\text{all } \mathbf{k}_o} C \cos\theta_o \, d\sigma_o \\ &= \int_{\phi_o=0}^{2\pi} \int_{\theta_o=0}^{\pi/2} k \cos\theta_o \sin\theta_o \, d\theta_o \, d\phi_o \\ &= \pi C. \end{aligned}$$

Thus, for a perfectly reflecting Lambertian surface ($R = 1$), we have $\rho = 1/\pi$, and for a Lambertian surface where $R(\mathbf{k}_i) = r$, we have

$$\rho(\mathbf{k}_i, \mathbf{k}_o) = \frac{r}{\pi}.$$

This is another example where the use of a steradian for the solid angle determines the normalizing constant and thus introduces factors of $\pi$.

## 19.2 Transport Equation

With the definition of BRDF, we can describe the radiance of a surface in terms of the incoming radiance from all different directions. Because in computer graphics we can use idealized mathematics that might be impractical to instantiate in the lab, we can also write the BRDF in terms of radiance only. If we take a small part of the light with solid angle $\Delta\sigma_i$ with radiance $L_i$ and "measure" the reflected radiance in direction $\mathbf{k}_o$ due to this small piece of the light, we can compute a BRDF (Figure 19.6). The irradiance due to the small piece of light is $H =$

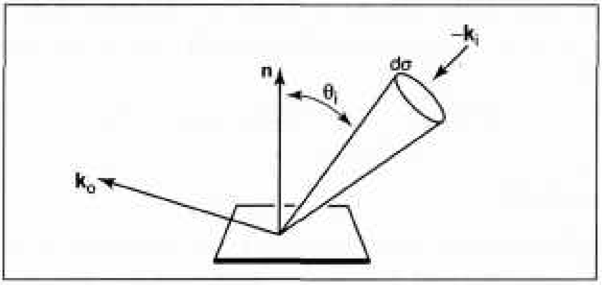


**Figure 19.6.** The geometry for the transport equation in its directional form.

$L_i \cos\theta_i \Delta\sigma_i$. Thus the BRDF is

$$\rho = \frac{L_o}{L_i \cos\theta_i \Delta\sigma_i}.$$

This form can be useful in some situations. Rearranging terms, we can write down the part of the radiance that is due to light coming from direction $\mathbf{k}_i$:

$$\Delta L_o = \rho(\mathbf{k}_i, \mathbf{k}_o) L_i \cos\theta_i \Delta\sigma_i.$$

If there is light coming from many directions $L_i(\mathbf{k}_i)$, we can sum all of them. In integral form, with notation for surface and field radiance, this is

$$L_s(\mathbf{k}_o) = \int_{\text{all } \mathbf{k}_i} \rho(\mathbf{k}_i, \mathbf{k}_o) L_f(\mathbf{k}_i) \cos\theta_i d\sigma_i.$$

This is often called the *rendering equation* in computer graphics (Immel, Cohen, & Greenberg, 1986).

Sometimes it is useful to write the transport equation in terms of surface radiances only (Kajiya, 1986). Note, that in a closed environment, the field radiance $L_f(\mathbf{k}_i)$ comes from some surface with surface radiance $L_s(-\mathbf{k}_i) = L_f(\mathbf{k}_i)$ (Figure 19.7). The solid angle subtended by the point $\mathbf{x}'$ in the figure is given by

$$\Delta\sigma_i = \frac{\Delta A' \cos\theta'}{\|\mathbf{x} - \mathbf{x}'\|^2},$$

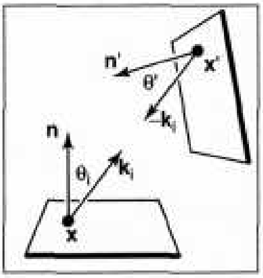


**Figure 19.7.** The light coming into one point comes from another point.

where $\Delta A'$ the the area we associate with $\mathbf{x}'$. Substituting for $\Delta\sigma_i$ in terms of $\Delta A'$ suggests the following transport equation:

$$L_s(\mathbf{x}, \mathbf{k}_o) = \int_{\text{all } \mathbf{x}' \text{ visible to } \mathbf{x}} \frac{\rho(\mathbf{k}_i, \mathbf{k}_o) L_s(\mathbf{x}', \mathbf{x} - \mathbf{x}') \cos\theta_i \cos\theta'}{\|\mathbf{x} - \mathbf{x}'\|^2} dA'.$$

Note that we are using a non-normalized vector $\mathbf{x} - \mathbf{x}'$ to indicate the direction from $\mathbf{x}'$ to $\mathbf{x}$. Also note that we are writing $L_s$ as a function of position and direction.

The only problem with this new transport equation is that the domain of integration is awkward. If we introduce a visibility function, we can trade off complexity in the domain with complexity in the integrand:

$$L_s(\mathbf{x}, \mathbf{k}_o) = \int_{\text{all } \mathbf{x}'} \frac{\rho(\mathbf{k}_i, \mathbf{k}_o) L_s(\mathbf{x}', \mathbf{x} - \mathbf{x}') v(\mathbf{x}, \mathbf{x}') \cos\theta_i \cos\theta'}{\|\mathbf{x} - \mathbf{x}'\|^2} dA',$$

where

$$v(\mathbf{x}, \mathbf{x}') = \begin{cases} 1 & \text{if } \mathbf{x} \text{ and } \mathbf{x}' \text{ are mutually visible,} \\ 0 & \text{otherwise.} \end{cases}$$

## 19.3 Photometry

For every spectral radiometric quantity there is a related *photometric quantity* that measures how much of that quantity is "useful" to a human observer. Given a spectral radiometric quantity $f_r(\lambda)$, the related photometric quantity $f_p$ is

$$f_p = 683\frac{\mathrm{lm}}{\mathrm{W}} \int_{\lambda=380\ \mathrm{nm}}^{800\ \mathrm{nm}} \bar{y}(\lambda) f_r(\lambda)\, d\lambda,$$

where $\bar{y}$ is the *luminous efficiency function* of the human visual system. This function is zero outside the limits of integration above, so the limits could be 0 and $\infty$ and $f_p$ would not change. The luminous efficiency function will be discussed in more detail in Chapter 20, but we discuss its general properties here. The leading constant is to make the definition consistent with historical absolute photometric quantities.

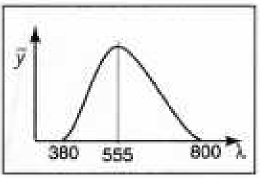


**Figure 19.8.** The luminous efficiency function versus wavelength (nm).

The luminous efficiency function is not equally sensitive to all wavelengths (Figure 19.8). For wavelengths below 380 nm (the *ultraviolet range*), the light is not visible to humans and thus has a $\bar{y}$ value of zero. From 380 nm it gradually increases until $\lambda = 555$ nm where it peaks. This is a pure green light. Then, it gradually decreases until it reaches the boundary of the infrared region at 800 nm.

The photometric quantity that is most commonly used in graphics is *luminance*, the photometric analog of radiance:

$$Y = 683\frac{\mathrm{lm}}{\mathrm{W}} \int_{\lambda=380\ \mathrm{nm}}^{800\ \mathrm{nm}} \bar{y}(\lambda) L(\lambda)\, d\lambda.$$

The symbol $Y$ for luminance comes from colorimetry. Most other fields use the symbol $L$; we will not follow that convention because it is too confusing to use $L$ for both luminance and spectral radiance. Luminance gives one a general idea of how "bright" something is independent of the adaptation of the viewer. Note that the black paper under noonday sun is subjectively darker than the lower luminance white paper under moonlight; reading too much into luminance is dangerous, but it is a very useful quantity for getting a quantitative feel for relative perceivable light output. The unit *lm* stands for *lumens*. Note that most light bulbs are rated in terms of the power they consume in watts, and the useful light they produce in lumens. More efficient bulbs produce more of their light where $\bar{y}$ is large and thus produce more lumens per watt. A "perfect" light would convert all power into 555 nm light and would produce 683 lumens per watt. The units of luminance are thus $(\mathrm{lm}/\mathrm{W})(\mathrm{W}/(\mathrm{m}^2\mathrm{sr})) = \mathrm{lm}/(\mathrm{m}^2\mathrm{sr})$. The quantity one lumen per steradian is defined to be one *candela* (*cd*), so luminance is usually described in units $\mathrm{cd}/\mathrm{m}^2$.

## Frequently Asked Questions

- What is "intensity"?

The term *intensity* is used in a variety of contexts and its use varies with both era and discipline. In practice, it is no longer meaningful as a specific radiometric quantity, but it is useful for intuitive discussion. Most papers that use it do so in place of radiance.

- What is "radiosity"?

The term *radiosity* is used in place of radiant exitance in some fields. It is also sometimes used to describe world-space light transport algorithms.

## Notes

A common radiometric quantity not described in this chapter is *radiant intensity* ($I$), which is the spectral power per steradian emitted from an infinitesimal point source. It should usually be avoided in graphics programs because point sources cause implementation problems. A more rigorous treatment of radiometry can be found in *Analytic Methods for Simulated Light Transport* (Arvo, 1995). The radiometric and photometric terms in this chapter are from the *Illumination Engineering Society's* standard that is increasingly used by all fields of science and engineering (American National Standard Institute, 1986). A broader discussion of radiometric and appearance standards can be found in *Principles of Digital Image Synthesis* (Glassner, 1995).

## Exercises

1. For a diffuse surface with outgoing radiance $L$, what is the radiant exitance?

2. What is the total power exiting a diffuse surface with an area of 4 $\mathrm{m}^2$ and a radiance of $L$?

3. If a fluorescent light and an incandescent light both consume 20 watts of power,.why is the fluorescent light usually preferred?

# 20

# Color

As discussed in Chapter 21, humans have three types of sensors (cones) active at high levels of illumination. The signals to these three sensor types determine the color response of an observer. For this reason, color is naturally a three-dimensional phenomenon. To quantitatively describe color we need to use a well-defined coordinate system on that three-dimensional space. In graphics we usually use "red-green-blue" (RGB) colors to provide such a coordinate system. However, there are infinitely many such coordinate systems we could apply to the space, and none of them is intrinsically superior to any other system. For specific circumstances, some color systems are better than others. This is analogous to having coordinate systems in a city that align with the streets rather than precise north/south/east/west directions. Thus, when we deal with colors, there are a plethora of acronyms such as CMY, XYZ, HSV, and LUV that stand for coordinate systems with three named axes. These are hard to keep straight.

In addition, color is an area that involves the physics of light entering the eye, as well as the psychology of what happens to that light. The distinction between what is physics, what is physiology, and what is cognition also tends to be confusing. Making matters even more complicated is that some color spaces are oriented toward display or print technologies, such as CMYK for ink-based printers with three colored inks plus a black (K) ink. To clarify things as much as possible, this chapter develops color perception from first principles. The discussion may seem a bit too detailed for the humble RGB color spaces that result, but the subject of color is intrinsically complex, and simplification is dangerous for such a central topic in graphics.

We begin with a section on light detectors and then develop basic trichromatic (three color) theory, and this leads naturally to dealing with RGB display systems. We also discuss the LMS and XYZ systems.

## 20.1 Light and Light Detectors

When the human eye "sees" something, it is because light enters the eye and hits a light detector on the *retina* at the back of the eye. Similarly, a digital camera records higher readings when more light hits a detector on the digital array at the back of the camera.

The signal that reaches the detector varies with wavelength and can be described by spectral radiance $L(\lambda)$ which represents the intensity of light coming from a particular direction at a particular wavelength. The retina/lens acts much like the radiance detector described in Section 19.1.5, with the lens allowing the eye to collect more light than would be possible with a simple opening.

All light is not created equal; humans are more sensitive to some wavelengths than others and are not sensitive at all to light outside the range $[380\ \mathrm{nm}, 800\ \mathrm{nm}]$. Cameras have a similar variable sensitivity. The response of any such detector can be represented as an integral of the product of a weighting function $w$ and the spectral radiance it "sees:"

$$\text{response} = \mathrm{k}\int w(\lambda)L(\lambda)d\lambda.$$

This response equation will thus be fundamental in any color theory. The somewhat arbitrary constant k will vary, as will the response function $w$ which is a characteristic of the sensors underlying the color theory.

## 20.2 Tristimulus Color Theory

If we assume that human color response is a result of several different types of sensors in the eye, an immediate question is, "With how many types of sensors are we dealing?" We now know that there are three types of sensors, called cones, that describe our day color vision. To see how this was verified in the 1800s, consider an experiment based on the hypothesis of three such sensor types. If we assume the sensors are independent, then the response of the sensors to a specific spectral radiance $A(\lambda)$ is (Wyszecki & Stiles, 1992)

$$\begin{aligned} S &= \int s(\lambda)A(\lambda)d\lambda, \\ M &= \int m(\lambda)A(\lambda)d\lambda, \\ L &= \int l(\lambda)A(\lambda)d\lambda. \end{aligned}$$

If two different radiances $A_1(\lambda)$ and $A_2(\lambda)$ produce the same $(S, M, L)$, then they are indistinguishable as far as the sensor system is concerned. Such matching spectra are seen as the same color and are called *metamers*. This observation is what allows us to verify that there are exactly three sensors.

Suppose we set up three spot lights which, when shined on a screen, have spectral curves $R(\lambda)$, $G(\lambda)$, and $B(\lambda)$, each with a control knob that scales them up and down with fractions $(r, g, b)$. The resulting spectral curve is

$$A(\lambda) = rR(\lambda) + gG(\lambda) + bB(\lambda).$$

The $S$ response to this mixed light is

$$\begin{aligned} S_A &= \int s(\lambda)A(\lambda)d\lambda \\ &= \int s(\lambda)\left(rR(\lambda) + gG(\lambda) + bB(\lambda)\right)d\lambda \\ &= r\int s(\lambda)R(\lambda)d\lambda + g\int s(\lambda)G(\lambda)d\lambda + b\int s(\lambda)B(\lambda)d\lambda \\ &\equiv rS_R + gS_G + bS_B. \end{aligned}$$

The final equivalence is just the result of defining the $S$ response to the full-strength lights to be $(S_R, S_G, S_B)$.

Now suppose we have a fourth light with spectral radiance $C(\lambda)$ that we shine on the screen next to the three overlapping colored lights. The sensor responses to the fourth light are $(S_C, M_C, L_C)$. Here is the key to our experiment: we can adjust the $(r, g, b)$ weighting for the three overlapping lights to make the lights look the same to the sensors, i.e.,

$$\begin{aligned} S_C &= S_A &= rS_R + gS_G + bS_B, \\ M_C &= M_A &= rM_R + gM_G + bM_B, \\ L_C &= L_A &= rL_R + gL_G + bL_B. \end{aligned}$$

Note that this is just a linear system with three equations and three unknowns: $(r, g, b)$. Provided the system is not degenerate, there is a unique $(r, g, b)$ that satisfies it.

This experiment was performed and users were able to make the colors match, and thus it was verified that there are exactly three sensor types. An important detail in the actual performance of the experiments is that there is no guarantee that $r$, $g$ and $b$ values are non-negative or are bounded above by one, so in these cases matches are impossible. However, if the users are allowed to also mix combinations of the first three lights in with the fourth light, matches can always be made. For example, if the match occurs when $r = -0.2$, then we can mix $0.2R(\lambda)$ in with the fourth light which has the same result as subtracting $0.2R(\lambda)$ from the overlapping lights.

Once we have established that there are three sensors, the next important question is, "What are the response functions $s(\lambda)$, $m(\lambda)$ and $l(\lambda)$?" Unfortunately, it is not possible using non-invasive procedures to infer these functions, and estimates of these response functions were determined only in the 1980s. However, we do not really need to know the cone response functions to come up with a color matching scheme. We can take any three lights that are linearly independent and use them to specify a color. For example, for the three lights discussed earlier, we can just use the values $(r, g, b)$ to specify a color. If two spectra are matched by the same $(r, g, b)$, then they are the same "color." Note that it is easily possible to have two different spectra that are matched by the same $(r, g, b)$, and they are then metamers.

What are the best lights to use for matching so that color values can be standardized? This question was addressed in the 1930s, and the $XYZ$ color system was developed. It is still the overwhelming choice for specifying tristimulus color. This system is discussed in the next section.

## 20.3 CIE Tristimulus Values

The CIE, a color standards organization, made the observation that once data was tabulated for a given set of lights, the tristimulus values for a given spectrum could be computed mathematically. They further observed that any set of real lights would result in negative tristimulus values for some test spectra. They decided there was no reason to restrict themselves to physically realizable lights. For example, if data is known for real lights $R(\lambda)$, $G(\lambda)$, $B(\lambda)$, then we can deduce data for linear combinations of those lights such as $-R(\lambda)$, $G(\lambda)-2B(\lambda)$, $B(\lambda) + R(\lambda)$ even though such lights cannot physically exist. The tristimulus values would then be $(-r, g - 2b, b + r)$.

The CIE decided to use imaginary lights that had two especially nice features:

- one of the lights is "grey" and provides no hue information;
- the other two lights have zero luminance and provide only hue information.

The response for these three lights is defined by the triple $(X, Y, Z)$ where $Y$ is the luminance. Because the eye responds only to light in the range 380 to 800 nanometers, these are the limits of integration listed. Since the weighting functions drop to zero outside that range, it is somewhat redundant to have explicit limits. The constant is 683 to conform to standards of luminance. The formula for the CIE *tristimulus* values $(X, Y, Z)$ is

$$X = 683 \int_{380}^{800} \bar{x}(\lambda)L(\lambda)d\lambda,$$
$$Y = 683 \int_{380}^{800} \bar{y}(\lambda)L(\lambda)d\lambda,$$
$$Z = 683 \int_{380}^{800} \bar{z}(\lambda)L(\lambda)d\lambda.$$

Any given spectral radiance will have a corresponding $(X, Y, Z)$.

## 20.4 Chromaticity

Often, we want to factor out luminance and concentrate on color. The standard way to do this is to use *chromaticity* values (Figure 20.1):

$$(x, y) = \left(\frac{X}{X+Y+Z}, \frac{Y}{X+Y+Z}\right).$$

There is a similar formula for $z$ but it is rarely used because $x+y+z=1$. Instead of $XYZ$, people often pass around only $xyY$. This way we can talk about "color" and "intensity" separately. We can also compute $XYZ$ from $xyY$:

$$(X, Y, Z) = \left(\frac{xY}{y}, Y, \frac{(1-x-y)Y}{y}\right). \tag{20.1}$$

In some sense, we can consider $(x, y)$ all of the information we need for *hue*, the chromatic part of color. Because it is a 2D space, the colors associated with the $(x, y)$ space can all be plotted on a flat page. An apparent oddity is that the $(x, y)$ points that have associated colors form an odd shape, and most points on the real plane have no associated colors. Because all the tristimulus values are non-negative, the values $(x, y)$ are also non-negative. Because each of $(x, y)$ is a non-negative number divided by a non-negative number at least as large, $(x, y)$ is restricted to the interval $[0, 1]^2$. However, the values are even more restricted than that.

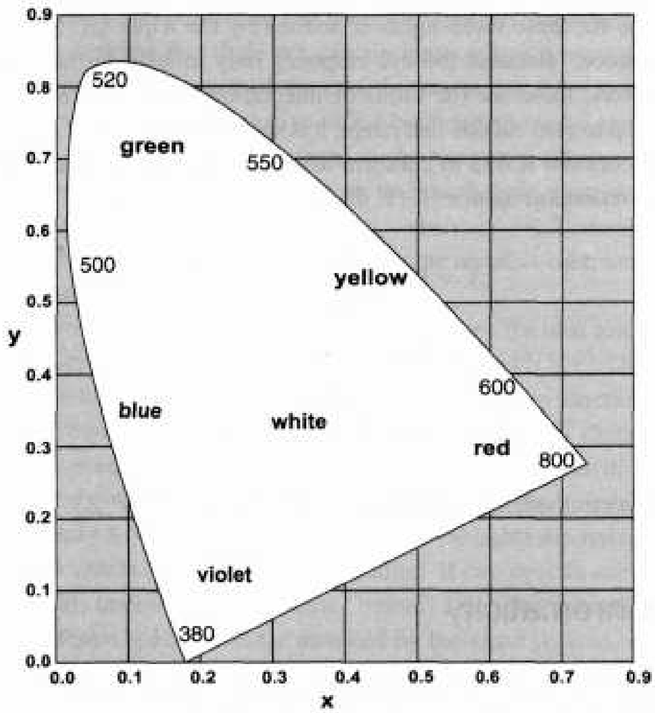


**Figure 20.1.** The CIE xy space. The pure spectral colors make up the curved boundaries, and the wavelengths (in nanometers) of the pure spectra are shown.

A table with the values for the tristimulus and scoptic sensitivity curves is given in Table 20.2. Note that the value of $\bar{y}$ is never more than five times the value of $\bar{x}$. This means that, regardless of the spectral input, $Y$ is never more than five times $X$, which restricts $y$ to be at most $5/6$. The most extreme cases occur for pure spectral colors. The spectral radiance of a pure spectral color at wavelength $\lambda_0$ is a scaled delta function $k\delta(\lambda_0)$. The tristimulus values are

$$(X, Y, Z) = 683k(\bar{x}(\lambda_0), \bar{y}(\lambda_0), \bar{z}(\lambda_0)).$$

The chromaticity values are thus

$$(x, y) = \left( \frac{\bar{x}(\lambda_0)}{\bar{x}(\lambda_0) + \bar{y}(\lambda_0) + \bar{z}(\lambda_0)}, \frac{\bar{y}(\lambda_0)}{\bar{x}(\lambda_0) + \bar{y}(\lambda_0) + \bar{z}(\lambda_0)} \right). \qquad (20.2)$$

| λ (nm) | $\bar{x}$ | $\bar{y}$ | $\bar{z}$ | $\bar{v}'$ |
|---|---|---|---|---|
| 380 | 0.0014 | 0.0000 | 0.0065 | 0.0006 |
| 390 | 0.0042 | 0.0001 | 0.0201 | 0.0022 |
| 400 | 0.0143 | 0.0004 | 0.0679 | 0.0093 |
| 410 | 0.0435 | 0.0012 | 0.2074 | 0.0348 |
| 420 | 0.1344 | 0.0040 | 0.6456 | 0.0966 |
| 430 | 0.2839 | 0.0116 | 1.3856 | 0.1998 |
| 440 | 0.3483 | 0.0230 | 1.7471 | 0.3281 |
| 450 | 0.3362 | 0.0380 | 1.7721 | 0.4550 |
| 460 | 0.2908 | 0.0600 | 1.6692 | 0.5670 |
| 470 | 0.1954 | 0.0910 | 1.2876 | 0.6760 |
| 480 | 0.0956 | 0.1390 | 0.8310 | 0.7930 |
| 490 | 0.0320 | 0.2080 | 0.4652 | 0.9040 |
| 500 | 0.0049 | 0.3230 | 0.2720 | 0.9820 |
| 510 | 0.0093 | 0.5030 | 0.1582 | 0.9970 |
| 520 | 0.0633 | 0.7100 | 0.0782 | 0.8350 |
| 530 | 0.1655 | 0.8620 | 0.0422 | 0.8110 |
| 540 | 0.2904 | 0.9540 | 0.0203 | 0.6500 |
| 550 | 0.4334 | 0.9950 | 0.0087 | 0.4810 |
| 560 | 0.5945 | 0.9950 | 0.0039 | 0.3288 |
| 570 | 0.7621 | 0.9520 | 0.0021 | 0.2076 |
| 580 | 0.9163 | 0.8700 | 0.0017 | 0.1212 |
| 590 | 1.0263 | 0.7570 | 0.0011 | 0.0655 |
| 600 | 1.0622 | 0.6310 | 0.0008 | 0.0332 |
| 610 | 1.0026 | 0.5030 | 0.0003 | 0.0159 |
| 620 | 0.8544 | 0.3810 | 0.0002 | 0.0074 |
| 630 | 0.6424 | 0.2650 | 0.0000 | 0.0033 |
| 640 | 0.4479 | 0.1750 | 0.0000 | 0.0015 |
| 650 | 0.2835 | 0.1070 | 0.0000 | 0.0007 |
| 660 | 0.1649 | 0.0610 | 0.0000 | 0.0003 |
| 670 | 0.0874 | 0.0320 | 0.0000 | 0.0001 |
| 680 | 0.0468 | 0.0170 | 0.0000 | 0.0001 |
| 690 | 0.0227 | 0.0082 | 0.0000 | 0.0000 |
| 700 | 0.0114 | 0.0041 | 0.0000 | 0.0000 |
| 710 | 0.0058 | 0.0021 | 0.0000 | 0.0000 |
| 720 | 0.0029 | 0.0010 | 0.0000 | 0.0000 |
| 730 | 0.0014 | 0.0005 | 0.0000 | 0.0000 |
| 740 | 0.0007 | 0.0002 | 0.0000 | 0.0000 |
| 750 | 0.0003 | 0.0001 | 0.0000 | 0.0000 |
| 760 | 0.0002 | 0.0001 | 0.0000 | 0.0000 |
| 770 | 0.0001 | 0.0000 | 0.0000 | 0.0000 |

**Figure 20.2.** Values for the scotopic and 1931 CIE tristimulus sensitivity curves (Rea, 1993).

## 20.5 Scotopic Luminance

For low levels of illumination, such as moonlight, your eyes go into a different perceptual mode, and the spectral sensitivity changes. The values of $XYZ$ become irrelevant, and the *scotopic luminance* determines light and dark:

$$V' = \mathrm{k}' \int_{380}^{800} \bar{v}'(\lambda) L(\lambda) d\lambda,$$

where the constant $\mathrm{k}' = 1700 \frac{\mathrm{s}}{\mathrm{W}}$ is chosen so that a monochrome 555 nm beam (the peak sensitivity of day vision) will have the same luminance and scotopic luminance.

Often we are presented with trichromatic values $(X, Y, Z)$ for data that is actually in the scotopic range and $V$ would be of more use. Although it is not possible to deduce $V$ from trichromatic values, Ward has shown that in many circumstances the following empirical formula performs reasonably (Larson, Rushmeier, & Piatko, 1997):

$$V = Y \left[ 1.33 \left( 1 + \frac{Y + Z}{X} \right) - 1.68 \right]. \tag{20.3}$$

## 20.6 RGB Monitors

We sometimes know the tristimulus values for the RGB channels of our CRT. Let's say these are $(X_r, Y_r, Z_r)$ for the red channel, $(X_g, Y_g, Z_g)$ for the green channel, and $(X_b, Y_b, Z_b)$ for the blue channel. If we assume a perfect black point for the monitor (unlikely), then given a gamma-corrected RGB signal of $(r, g, b)$, we can compute the screen tristimulus values $(X_s, Y_s, Z_s)$:

$$\begin{bmatrix} X_s \\ Y_s \\ Z_s \end{bmatrix} = \begin{bmatrix} rX_r + gX_g + bX_b \\ rY_r + gY_g + bY_b \\ rZ_r + gZ_g + bZ_b \end{bmatrix} = \begin{bmatrix} X_r & X_g & X_b \\ Y_r & Y_g & Y_b \\ Z_r & Z_g & Z_b \end{bmatrix} \begin{bmatrix} r \\ g \\ b \end{bmatrix}.$$

You can invert the above equation to figure out how to set $(r, g, b)$ given a desired $(X, Y, Z)$. Note that you may get values *much* larger than 1.0, or smaller than 0.0, so you will need to do some manipulation to deal with that problem. However, there is a bigger problem. Monitor manufacturers almost never tell you $(X_r, Y_r, Z_r)$ etc. Instead they provide the chromaticity of the phosphors $(x_r, y_r)$, $(x_g, y_g)$, $(x_b, y_b)$, and the chromaticity of the *white point* $(x_w, y_w)$. In addition, you can usually measure the luminance $Y_w$ of the brightest white screen with a photometer. If you can't measure that, assume it is approximately

$Y_w = 100\,\mathrm{cd/m^2}$. The reason manufacturers don't tell you $Y_w$ is that your brightness control changes it. The reason the white point varies is that "white" is usually the average color in the room. Thus, what looks white in a fluorescent-lit room will have dominant short wavelengths, and what looks white in an incandescent-lit room will have dominant long wavelengths. So, if you move a monitor with a white-looking image from a fluorescent-lighted room to a incandescent-lit room that same display will look blue. This same issue causes photographers to buy "daylight" or "tungsten" film.

To convert the information the manufacturers provide (tristimulus values), we need to do some algebra. First, let's write a straightforward equality:

$$\begin{bmatrix} \frac{X_r}{Y_r} & \frac{X_g}{Y_g} & \frac{X_b}{Y_b} \\ 1 & 1 & 1 \\ \frac{Z_r}{Y_r} & \frac{Z_g}{Y_g} & \frac{Z_b}{Y_b} \end{bmatrix} \begin{bmatrix} Y_r \\ Y_g \\ Y_b \end{bmatrix} = \begin{bmatrix} X_w \\ Y_w \\ Z_w \end{bmatrix}.$$

Now, after some substitutions, we have only $(Y_r, Y_g, Y_b)$ as unknowns:

$$\begin{bmatrix} \frac{x_r}{y_r} & \frac{x_g}{y_g} & \frac{x_b}{y_b} \\ 1 & 1 & 1 \\ \frac{1-x_r-y_r}{y_r} & \frac{1-x_g-y_g}{y_g} & \frac{1-x_b-y_b}{y_b} \end{bmatrix} \begin{bmatrix} Y_r \\ Y_g \\ Y_b \end{bmatrix} = \begin{bmatrix} \frac{x_w Y_w}{y_w} \\ Y_w \\ \frac{(1-x_w-y_w)Y_w}{y_w} \end{bmatrix}.$$

We can use numerical methods, or (algebraically) Cramer's rule, to solve for $(Y_r, Y_g, Y_b)$: Once we have that, we can get $(X_r, X_g, X_b)$, $(Y_r, Y_g, Y_b)$, $(Z_r, Z_g, Z_b)$ using Equation 20.1. Now that we have this result, we can apply it for specific monitor parameters. These can either be measured, or a standard can be used if appropriate such as those in (ITU, 1990) or on the web at http://www.w3.org/Graphics/Color/sRGB.html.

## 20.7 Approximate Color Manipulation

Although the techniques of the previous section are useful for highly controlled settings, in practice we rarely know enough data to use them. Often we make images that are to be displayed on many unknown monitors, e.g., an image for a web page. It is convenient to use a "normalized" space where $(R, G, B) = (1, 1, 1)$ transforms to $(X, Y, Z) = (1, 1, 1)$ and the $RGB$ space is "reasonable" for most real monitors. Such a space is given in *Color Transfer Between Images* (Reinhard, Ashikhmin, Gooch, & Shirley, 2001) and is presented here with a slight modification:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.5149 & 0.3244 & 0.1607 \\ 0.2654 & 0.6704 & 0.0642 \\ 0.0248 & 0.1248 & 0.8504 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}, \tag{20.4}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 2.5623 & -1.1661 & -0.3962 \\ -1.0215 & 1.9778 & 0.0437 \\ 0.0752 & -0.2562 & 1.1810 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}, \tag{20.5}$$

$$\begin{bmatrix} L \\ M \\ S \end{bmatrix} = \begin{bmatrix} 0.3897 & 0.6890 & -0.0787 \\ -0.2298 & 1.1834 & 0.0464 \\ 0.0000 & 0.0000 & 1.0000 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}, \tag{20.6}$$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 1.9102 & -1.1122 & 0.2019 \\ 0.3709 & 0.6291 & 0.0000 \\ 0.0000 & 0.0000 & 1.0000 \end{bmatrix} \begin{bmatrix} L \\ M \\ S \end{bmatrix}. \tag{20.7}$$

This gives the following transformation between RGB and LMS cone space:

$$\begin{bmatrix} L \\ M \\ S \end{bmatrix} = \begin{bmatrix} 0.3816 & 0.5785 & 0.0399 \\ 0.1969 & 0.7246 & 0.0785 \\ 0.0248 & 0.1248 & 0.8504 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}, \tag{20.8}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 4.4620 & -3.5832 & 0.1213 \\ -1.2178 & 2.3803 & -0.1626 \\ 0.0486 & -0.2448 & 1.1962 \end{bmatrix} \begin{bmatrix} L \\ M \\ S \end{bmatrix}. \tag{20.9}$$

## 20.8 Opponent Color Spaces

While there exist many "intuitive" color spaces that create three dimensions within color, perhaps the best known are the *opponent color models* (Wyszecki & Stiles, 1992). These have three channels that are typically similar to:

$$\begin{aligned} \text{Achromatic} &\propto r + g + b, \\ \text{Yellow-blue} &\propto r + g - b, \\ \text{Red-green} &\propto r - g. \end{aligned} \tag{20.10}$$

Although there is some biological justification for this type of model, it was proposed in the 1800s based on subjective experience (Wyszecki & Stiles, 1992). For example, there is an experience of bluish-green, but not yellowish-blue. The implication is that there are different channels for blue and green, but that there is a shared channel that is either yellow or blue, but not both.

While some opponent models assume a log space (Ruderman, Cronin, & Chiao, 1998; Reinhard et al., 2001), we present a linear model here, where $\alpha$ and $\beta$ are the chromatic channels:

$$\begin{bmatrix} \mathcal{L} \\ \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{3}} & 0 & 0 \\ 0 & \frac{1}{\sqrt{6}} & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & -2 \\ 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} L \\ M \\ S \end{bmatrix},$$

$$\begin{bmatrix} L \\ M \\ S \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & -1 \\ 1 & -2 & 0 \end{bmatrix} \begin{bmatrix} \frac{\sqrt{3}}{3} & 0 & 0 \\ 0 & \frac{\sqrt{6}}{6} & 0 \\ 0 & 0 & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} \mathcal{L} \\ \alpha \\ \beta \end{bmatrix}.$$

## Frequently Asked Questions

### • What is "hue"?

Hue is the dominant color name of the "non-white" component of the color. For example, the hue of pink is red. Many color systems encode hue as an angle from 0° to 360°, with red at 0°, green at 120° and blue at 240°.

### • What is "lightness"?

Lightness is the overall intensity of the reflectance of a surface. This is minimal for black, and maximum for white. Two colors with different hues can have the same lightness. Lightness is often encoded as a zero to one scale.

### • What is "saturation"?

Saturation is the purity of a color. For example, red is more saturated than pink, and grey is not saturated at all.

### • What is "value"?

Value is another word for lightness, but often it is expressed as a numeric scale that is approximate. For example, in some systems value is the average of the RGB values. Thus two colors with the same value might have different subjective lightness because the RGB channels are perceived differently.

## Notes

A good survey of color for computer graphics users is *Computer Generated Color* (Jackson, MacDonald, & Freeman, 1994). Books written from the color science perspective are *The Reproduction of Colour* (Hunt, 2004) and *Color Appearance Models* (Fairchild, 2005). The color spaces for color-deficient viewers are discussed in (Meyer & Greenberg, 1988)

## Exercises

1. Plot the outline of the CIE chromaticity diagram using Equation 20.2.
2. What is the one physically possible CIE $(x, y)$ value that can be zero?
3. Write a program to scale the individual color channels separately in RGB and opponent space. Which produces worse artifacts?

William B. Thompson

21

# Visual Perception

The ultimate purpose of computer graphics is to produce images for viewing by people. Thus, the success of a computer graphics system depends on how well it conveys relevant information to a human observer. The intrinsic complexity of the physical world and the limitations of display devices make it impossible to present a viewer with the identical patterns of light that would occur when looking at a natural environment. When the goal of a computer graphics system is physical realism, the best we can hope for is that the system be *perceptually effective*: displayed images should "look" as intended. For applications such as technical illustration, it is often desirable to visually highlight relevant information and perceptual effectiveness becomes an explicit requirement.

Artists and illustrators have developed empirically a broad range of tools and techniques for effectively conveying visual information. One approach to improving the perceptual effectiveness of computer graphics is to utilize these methods in our automated systems. A second approach builds directly on knowledge of the human vision system by using perceptual effectiveness as an optimization criteria in the design of computer graphics systems. These two approaches are not completely distinct. Indeed, one of the first systematic examinations of visual perception is found in the notebooks of Leonardo da Vinci.

The remainder of this chapter provides a partial overview of what is known about visual perception in people. The emphasis is on aspects of human vision that are most relevant to computer graphics. The human visual system is extremely complex in both its operation and its architecture. A chapter such as this can at best provide a summary of key points, and it is important to avoid over generalizing from what is presented here. More in-depth treatments of visual perception can be found in Wandell (1995) and Palmer (1999); Gregory (1997) and Yantis (2000) provide additional useful information. A good computer vision reference such as Forsyth and Ponce (2002) is also helpful. It is important to note that despite over 150 years of intensive research, our knowledge of many aspects of vision is still very limited and imperfect.

## 21.1 Vision Science

Vision is generally agreed to be the most powerful of the senses in humans. Vision produces more useful information about the world than does hearing, touch, smell, or taste. This is a direct consequence of the physics of light (Figure 21.1). Illumination is pervasive, especially during the day but also at night due to moonlight, starlight, and artificial sources. Surfaces reflect a substantial portion of incident illumination and do so in ways that are idiosyncratic to particular materials and that are dependent on the shape of the surface. The fact that light (mostly) travels in straight lines through the air allows vision to acquire information from distant locations.

Light:

- travels far
- travels fast
- travels in straight lines
- interacts with stuff
- bounces off things
- is produced in nature
- has lots of energy

—Steven Shafer

**Figure 21.1.** The nature of light makes vision a powerful sense.

The study of vision has a long and rich history. Much of what we know about the eye traces back to the work of philosophers and physicists in the 1600s. Starting in the mid-1800s, there was an explosion of work by perceptual psychologists exploring the phenomenology of vision and proposing models of how vision might work. The mid-1900s saw the start of modern neuroscience, which investigates both the fine-scale workings of individual neurons and the large-scale architectural organization of the brain and nervous system. A substantial portion of neuroscience research has focused on vision. More recently, computer science has contributed to the understanding of visual perception by providing tools for precisely describing hypothesized models of visual computations and by allowing empirical examination of computer vision programs. The term *vision science* was coined to refer to the multidisciplinary study of visual perception involving perceptual psychology, neuroscience, and computational analysis.

Vision science views the purpose of vision as producing information about objects, locations, and events in the world from imaged patterns of light reaching the viewer. Psychologists use the term *distal stimulus* to refer to the physical world under observation and *proximal stimulus* to refer to the retinal image.[1] Us-

[1] In computer vision, the term *scene* is often used to refer to the external world, while the term *image* is used to refer to the projection of the scene onto a sensing plane.

ing this terminology, the function of vision is to generate a description of aspects of the distal stimulus given the proximal stimulus. Visual perception is said to be *veridical* when the description that is produced accurately reflects the real world. In practice, it makes little sense to think of these descriptions of objects, locations, and events in isolation. Rather, vision is better understood in the context of the motor and cognitive functions that it serves.

## 21.2 Visual Sensitivity

Vision systems create descriptions of the visual environment based on properties of the incident illumination. As a result, it is important to understand what properties of incident illumination the human vision system can actually detect. One critical observation about the human vision system is that it is primarily sensitive to *patterns* of light rather than being sensitive to the absolute magnitude of light energy. The eye does not operate as a photometer. Instead, it detects spatial, temporal, and spectral patterns in the light imaged on the retina and information about these patterns of light form the basis for all of visual perception.

There is a clear ecological utility to the vision system's sensitivity to variations in illumination over space and time. Being able to accurately sense changes in the environment is crucial to our survival.[2] A system which measures changes in light energy rather than the magnitude of the energy itself also makes engineering sense, since it makes it easier to detect patterns of light over large ranges in light intensity. It is a good thing for computer graphics that vision operates in this manner. Display devices are physically limited in their ability to project light with the power and dynamic range typical of natural scenes. Graphical displays would not be effective if they needed to produce the identical patterns of light as the corresponding physical world. Fortunately, all that is required is that displays be able to produce similar patterns of spatial and temporal change to the real world.

### 21.2.1 Brightness and Contrast

In bright light, the human visual system is capable of distinguishing gratings consisting of high contrast parallel light and dark bars as fine as 50–60 cycles/degree. (In this case, a "cycle" consists of an adjacent pair of light and dark bars.) For

[2] It is sometime said that the primary goals of vision are to support eating, avoiding being eaten, reproduction, and avoidance of catastrophe while moving. Thinking about vision as a goal-directed activity is often useful, but needs to be done so at a more detailed level.

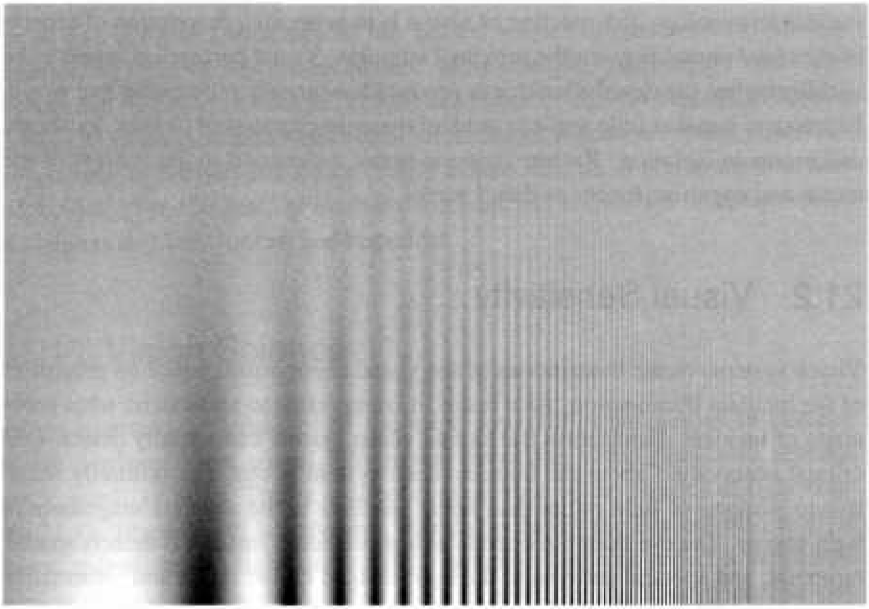


**Figure 21.2.** The contrast between stripes increases in a constant manner from top to bottom, yet the threshold of visibility varies with frequency.

comparison, the best currently available LCD computer monitor, at a normal viewing distance, can display patterns as fine as about 20 cycles/degree. The minimum contrast difference at an edge detectable by the human visual system in bright light is about 1% of the average luminance across the edge. In most 8-bit displays, differences of a single gray level are often noticeable over at least a portion of the range of intensities due to the nature of the mapping from gray levels to actual display luminance.

Characterizing the ability of the visual system to detect fine scale patterns (*visual acuity*) and to detect changes in brightness is considerably more complicated than for cameras and similar image acquisition devices. As shown in Figure 21.2, there is an interaction between contrast and acuity in human vision. In the figure, the scale of the pattern decreases from left to right while the contrast increases from top to bottom. If you view the figure at a normal viewing distance, it will be clear that the lowest contrast at which a pattern is visible is a function of the spatial frequency of the pattern.

There is a linear relationship between the intensity of light $L$ reaching the eye from a particular surface point in the world, the intensity of light $I$ illuminating that surface point, and the reflectivity $R$ of the surface at the point being observed:

$$L = \alpha I \cdot R, \tag{21.1}$$

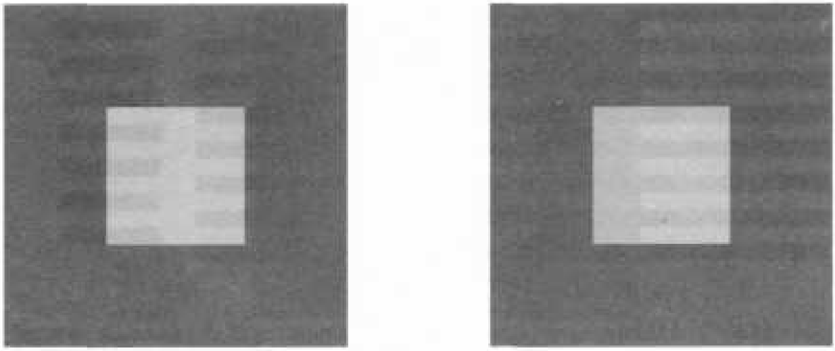


**Figure 21.3.** *Lightness constancy.* Cast a shadow over one of the patterns with your hand and notice that the apparent brightness of the two center squares remains nearly the same.

where $\alpha$ is dependent on the relationship between the surface geometry, the pattern of incident illumination, and the viewing direction. While the eye is only able to directly measure $L$, human vision is much better at estimating $R$ than $L$. To see this, view Figure 21.3 in bright direct light. Use your hand to shadow one of the patterns, leaving the other directly illuminated. While the light reflected off of the two patterns will be significantly different, the apparent brightness of the two center squares will seem nearly the same. The term *lightness* is often used to describe the apparent brightness of a surface, as distinct from its actual luminance. In many situations, lightness is invariant to large changes in illumination, a phenomenon referred to as *lightness constancy*.

The mechanisms by which the human visual system achieves lightness constancy are not well understood. As shown in Figure 21.2, the vision system is relatively insensitive to slowly varying patterns of light, which may serve to discount the effects of slowly varying illumination. Apparent brightness is affected by the brightness of surrounding regions (Figure 21.4). This can aid lightness constancy when regions are illuminated dissimilarly. While this *simultaneous contrast* effect is often described as a modification of the perceived lightness of

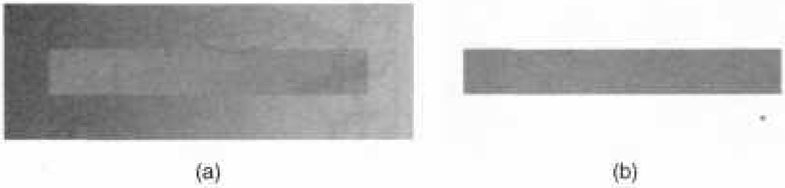


**Figure 21.4.** (a) Simultaneous contrast: the apparent brightness of the center bar is affected by the brightness of the surrounding area; (b) The same bar without a variable surround.

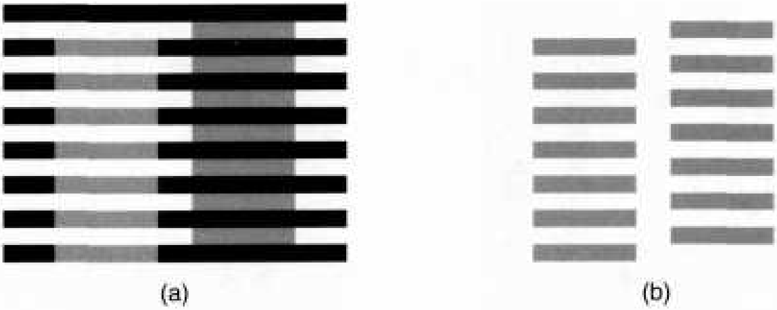


**Figure 21.5.** The Munker-White illusion shows the complexity of simultaneous contrast. In Figure 21.4, the central region looked lighter when the surrounding area was darker. In (a), the gray strips on the left look *lighter* than the gray strips on the right, even though they are nearly surrounded by regions of white; (b) shows the gray strips without the black lines.

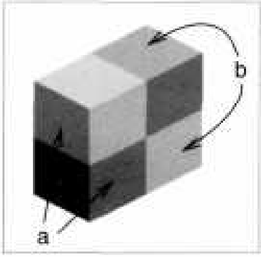


**Figure 21.6.** The perception of lightness is affected by the perception of 3D structure. The two surfaces marked (a) have the same brightness, as do the two surfaces marked (b) (after Adelson (1999)).

one region based on contrasting brightness in the surrounding region, it is actually much more complicated than that (Figures 21.5 and 21.6). For more on lightness perception, see (Gilchrist et al., 1999) and (Adelson, 1999).

While the visual system largely ignores slowly varying intensity patterns, it is extremely sensitive to *edges* consisting of lines of discontinuity in brightness. Edges in imaged light intensity often correspond to surface boundaries or other important features in the environment (Figure 21.7). The vision system can also detect localized differences in motion, stereo disparity, texture, and several other

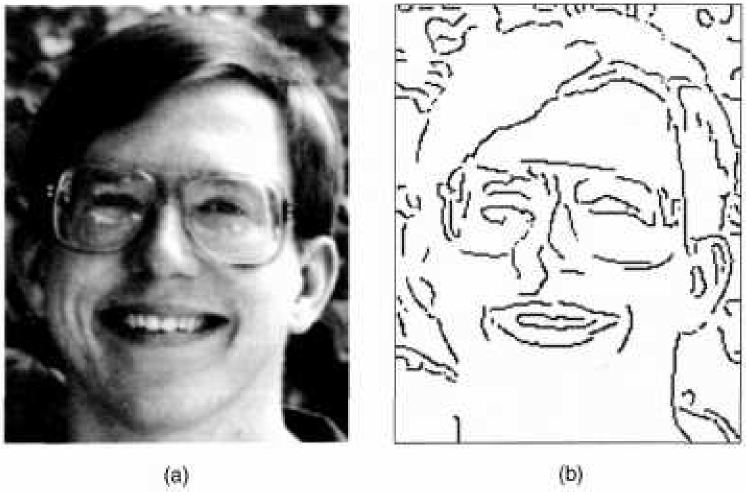


**Figure 21.7.** (a) Original gray scale image, (b) image *edges*, which are lines of high spatial variability in some direction.

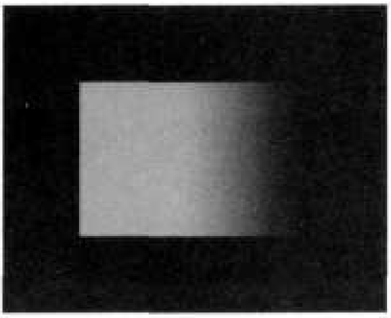


**Figure 21.8.** The visual system sometimes sees "edges" even when there are no sharp discontinuities in brightness, as is the case at the right side of the central pattern in this image.

image properties. The vision system has very little ability, however, to detect spatial discontinuities in color when not accompanied by differences in one of these other properties.

Perception of edges seems to interact with perception of form. While edges give the visual system the information it needs to recognize shapes, slowly varying brightness can appear as a sharp edge if the resulting edge creates a more complete form (Figure 21.8). Figure 21.9 shows a *subjective contour*, an extreme form of this effect in which a closed contour is seen even though no such contour exists in the actual image. Finally, the vision system's sensitivity to edges also appears to be part of the mechanism involved in lightness perception. Note that the region enclosed by the subjective contour in Figure 21.9 appears a bit brighter than the surrounding area of the page. Figure 21.10 shows a different interaction between edges and lightness. In this case, a particular brightness profile at the edge has a dramatic effect on the apparent brightness of the surfaces to either side of the edge.

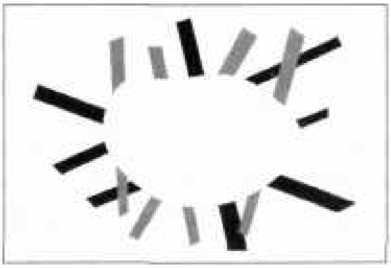


**Figure 21.9.** Sometimes, the visual system will "see" *subjective contours* without any associated change in brightness.

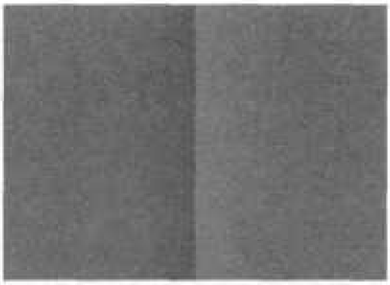


**Figure 21.10.** Perceived lightness depends more on local contrast at edges than on brightness across surfaces. Try covering the vertical edge in the middle of the figure with a pencil. This figure is an instance of the *Craik-O'Brien-Cornsweet illusion.*

As indicated above, people can detect differences in the brightness between two adjacent regions if the difference is at least 1% of the average brightness. This is an example of *Weber's law*, which states that there is a constant ratio between the *just noticeable differences* (jnd) in a stimulus and the magnitude of the stimulus:

$$\frac{\Delta I}{I} = k_1, \tag{21.2}$$

where $I$ is the magnitude of the stimulus, $\Delta I$ is the magnitude of the just noticeable difference, and $k_1$ is a constant particular to the stimulus. Weber's law was postulated in 1846 and still remains a useful characterization of many perceptual effects. *Fechner's law*, proposed in 1860, generalized Weber's law in a way that allowed for the description of the strength of any sensory experience, not just jnd's:

$$S = k_2 \log(I), \tag{21.3}$$

where $S$ is the perceptual strength of the sensory experience, $I$ is the physical magnitude of the corresponding stimulus, and $k_2$ is a scaling constant specific to the stimulus. Current practice is to model the association between perceived and actual strength of a stimulus using a power function (*Stevens's law*):

$$S = k_3 I^b, \tag{21.4}$$

where $S$ and $I$ are as before, $k_3$ is another scaling constant, and $b$ is an exponent specific to the stimulus. For a large number of perceptual quantities involving vision, $b < 1$. The CIE L*a*b* color space, described elsewhere, uses a modified Stevens's law representation to characterize perceptual differences between brightness values. Note that in the first two characterizations of the perceptual strength of a stimulus and in Steven's Law when $b < 1$, changes in the stimulus when it has a small average magnitude create larger perceptual effects than do the same physical change in the stimulus when it has a larger magnitude.

The "laws" describe above are not physical constraints on how perception operates. Rather, they are generalizations about how the perceptual system responds to particular physical stimuli. In the field of perceptual psychology, the quantitative study of the relationships between physical stimuli and their perceptual effects is called *psychophysics*. While psychophysical laws are empirically derived observations rather than mechanistic accounts, the fact that so many perceptual effects are well modeled by simple power functions is striking and may provide insights into the mechanisms involved.

### 21.2.2 Color

In 1666, Isaac Newton used prisms to show that apparently white sunlight could be decomposed into a *spectrum* of colors and that these colors could be recombined to produce light that appeared white. We now know that light energy is made up of a collection of photons, each with a particular wavelength. The *spectral distribution* of light is a measure of the average energy of the light at each wavelength. For natural illumination, the spectral distribution of light reflected off of surfaces varies significantly depending on the surface material. Characterizations of this spectral distribution can therefore provide visual information for the nature of surfaces in the environment.

Most people have a pervasive sense of color when they view the world. Color perception depends on the frequency distribution of light, with the visible spectrum for humans ranging from a wavelength of about 370 nm to a wavelength of about 730 nm (see Color Plate XVI). The manner in which the visual systems derives a sense of color from this spectral distribution was first systematically examined in 1801 and remained extremely controversial for 150 years. The problem is that the visual system responds to patterns of spectral distribution very differently than patterns of luminance distribution.

> *"The history of the investigation of colour vision is remarkable for its acrimony."*
>
> —Richard Gregory (1997)

Even accounting for phenomena such as lightness constancy, distinctly different spatial distributions almost always look distinctly different. More importantly given that the purpose of the visual system is to produce descriptions of the distal stimulus given the proximal stimulus, perceived patterns of lightness correspond at least approximately to patterns of brightness over surfaces in the environment. The same is not true of color perception. Many quite different spectral distributions of light can produce a sense of any specific color. Correspondingly, the sense that a surface is a specific color provides little direct information about the spectral distribution of light coming from the surface. For example, a spectral distribution consisting of a combination of light at wavelengths of 700 nm and 540 nm, with appropriately chosen relative strengths, will look indistinguishable from light at the single wavelength of 580 nm. (Perceptually indistinguishable colors with different spectral compositions are referred to as *metamers*.) If we see the color "yellow," we have no way of knowing if it was generated by one or the other of these distributions or an infinite family of other spectral distributions. For this reason, in the context of vision the term *color* refers to a purely perceptual quality, not a physical property.

There are two classes of photoreceptors in the human retina. *Cones* are involved in color perception, while *rods* are sensitive to light energy across the visible range and do not provide information about color. There are three types of cones, each with a different spectral sensitivity (Figure 21.11). *S-cones* respond to short wavelengths in the blue range of the visible spectrum. *M-cones* respond to wavelengths in the middle (greenish) region of the visible spectrum. *L-cones* respond to somewhat longer wavelengths covering the green and red portions of the visible spectrum.

While it is common to describe the three types of cones as *red*, *green*, and *blue*, this is neither correct terminology nor does it accurately reflect the cone sensitivities shown in Figure 21.11. The *L-cones* and *M-cones* are broadly tuned, meaning that they respond to a wide range of frequencies. There is also substantial overlap between the sensitivity curves of the three cone types. Taken together, these two properties mean that it is not possible to reconstruct an approximation to the original spectral distribution given the responses of the three cone types. This is in contrast to spatial sampling in the retina (and in digital cameras), where

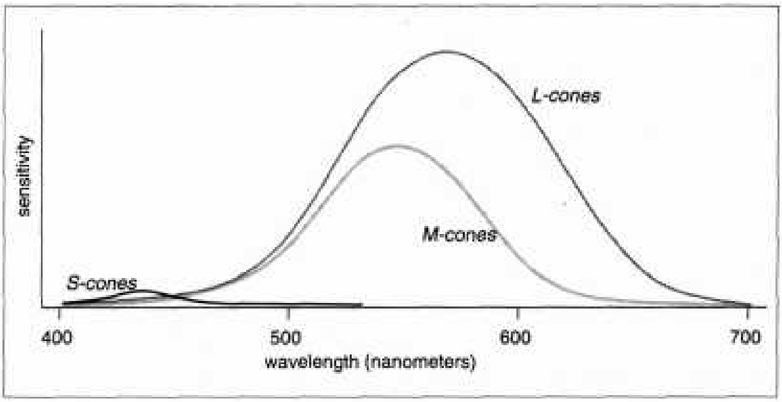


**Figure 21.11.** Spectral sensitivity of the *short*, *medium*, and *long* cones in the human retina.

the receptors are narrowly tuned in their spatial sensitivity in order to be able to detect fine detail in local contrast.

The fact that there are are only three types of color sensitive photoreceptors in the human retina greatly simplifies the task of displaying colors on computer monitors and in other graphical displays. Computer monitors display colors as a weighted combination of three fixed color distributions. Most often, the three colors are a distinct red, a distinct green, and a distinct blue. As a result, in computer graphics, color is often represented by a *red-green-blue* (RGB) triple, representing the intensities of red, green, and blue primaries needed to display a particular color. Three *basis colors* are sufficient to display most perceptible colors, since appropriately weighted combinations of three appropriately chosen colors can produce metamers for these perceptible colors.

There are at least two significant problems with the RGB color representation. The first is that different monitors have different spectral distributions for their red, green, and blue primaries. As a result, perceptually correct color rendition involves remapping RGB values for each monitor. This is of course only possible if the original RGB values satisfy some well defined standard, which is often not the case. See Chapter 20 for more information on this issue. The second problem is that RGB values do not define a particular color in a way that corresponds to subjective perception. When we see the color "yellow," we do not have the sense that it is made up of equal parts of red and green light. Rather, it looks like a single color, with additional properties involving brightness and the "amount" of color. Representing color as the output of the S-cones, M-cones, and L-cones is no help either, since we have no more phenomenological sense of color as characterized by these properties than we do as characterized by RGB display properties.

There are two different approaches to characterizing color in a way that more closely reflects human perception. The various CIE color spaces aim to to be "perceptually uniform" so that the magnitude of the difference in the represented values of two colors is proportional to the perceived difference in color (Wyszecki & Stiles, 1992). This turns out to be a difficult goal to accomplish, and there have been several modifications to the CIE model over the years. Furthermore, while one of the dimensions of the CIE color spaces corresponds to perceived brightness, the other two dimensions that specify chromaticity have no intuitive meaning.

The second approach to characterizing color in a more natural manner starts with the observation that there are three distinct and independent properties that dominate the subjective sense of color. *Lightness*, the apparent brightness of a surface, has already been discussed. *Saturation* refers to the purity or vividness of a color. Colors can range from totally unsaturated gray to partially saturated pastels to fully saturated "pure" colors. The third property, *hue*, corresponds most closely to the informal sense of the word "color" and is characterized in a manner similar to colors in the visible spectrum, ranging from dark violet to dark red. Color Plate XVII shows a plot of the hue-saturation-lightness (HSV) color space. Since the relationship between brightness and lightness is both complex and not well understood, HSV color spaces almost always use brightness instead of attempting to estimate lightness. Unlike wavelengths in the spectrum, however, hue is usually represented in a manner that reflects the fact that the extremes of the visible spectrum are actually similar in appearance (Color Plate XVIII). Simple transformations exist between RGB and HSV representations of a particular color value. As a result, while the HSV color space is motivated by perceptual considerations, it contains no more information than does an RGB representation.

The hue-saturation-lightness approach to describing color is based on the spectral distribution at a single point and so only approximates the perceptual response to spectral distributions of light distributed over space. Color perception is subject to similar constancy and simultaneous contrast effects as is lightness/brightness, neither of which are captured in the RGB representation and as a result are not captured in the HSV representation. For an example of color constancy, look at a piece of white paper indoors under incandescent light and outdoors under direct sunlight. The paper will look "white" in both cases, even though incandescent light has a distinctly yellow hue and so the light reflected off of the paper will also have a yellow hue, while sunlight has a much more uniform color spectrum.

Another aspect of color perception not captured by either the CIE color spaces or HSV encoding is the fact that we see a small number of distinct colors when looking at a continuous spectrum of visible light (Color Plate XVI) or in a naturally occurring rainbow. For most people, the visible spectrum appears to be divided into four to six distinct colors: red, yellow, green, and blue, plus perhaps light blue and purple. Considering non-spectral colors as well, there are only eleven basic color terms commonly used in English: *red*, *green*, *blue*, *yellow*, *black*, *white*, *gray*, *orange*, *purple*, *brown*, and *pink*. The partitioning of the intrinsically continuous space of spectral distributions into a relatively small set of perceptual categories associated with well defined linguistic terms seems to be a basic property of perception, not just a cultural artifact (Berlin & Kay, 1969). The exact nature of the process, however, is not well understood.

### 21.2.3 Dynamic Range

Natural illumination varies in intensity over 6 orders of magnitude (Figure 21.12). The human vision system is able to operate over this full range of brightness levels. However, at any one point in time the visual system is only able to detect variations in light intensity over a much smaller range. As the average brightness to which the visual system is exposed changes over time, the range of discriminable brightnesses changes in a corresponding manner. This effect is most obvious if we move rapidly from a brightly lit outdoor area to a very dark room. At first, we are able to see little. After a while, however, details in the room start to become apparent. The *dark adaptation* that occurs involves a number physiological changes in the eye. It takes several minutes for significant dark adaptation to occur and 40 minutes or so for complete dark adaptation. If we then move back into the bright light, not only is vision difficult but it can actually be painful. *Light adaptation* is required before it is again possible to see clearly. Light adaptation occurs much more quickly than dark adaptation, typically requiring less than a minute.

| | |
|---|---|
| *direct sunlight* | $10^5$ |
| *indoor lighting* | $10^2$ |
| *moonlight* | $10^{-1}$ |
| *starlight* | $10^{-3}$ |

**Figure 21.12.** Approximate luminance level of a white surface under different types of illumination in candelas per meter squared (cd/m$^2$). (Wandell, 1995).

The two classes of photoreceptors in the human retina are sensitive to different ranges of brightness. The cones provide visual information over most of what we consider normal lighting conditions, ranging from bright sunlight to dim indoor lighting. The rods are only effective at very low light levels. *Photopic* vision involves bright light in which only the cones are effective. *Scotopic* vision involves dark light in which only the rods are effective. There is a range of intensities within which both cones and rods are sensitive to changes in light, which is referred to as *mesopic* conditions (see Chapter 22).

### 21.2.4 Field-of-View and Acuity

Each eye in the human visual system has a field-of-view of approximately 160° horizontal by 135° vertical. With binocular viewing, there is only partial overlap between the fields-of-view of the two eyes. This results in a wider overall field-of-view (approximately 200° horizontal by 135° vertical), with the region of overlap being approximately 120° horizontal by 135° vertical.

With normal or corrected-to-normal vision, we usually have the subjective experience of being able to see relatively fine detail wherever we look. This is an illusion, however. Only a small portion of the visual field of each eye is actually sensitive to fine detail. To see this, hold a piece of paper covered with normal-sized text at arms length, as shown in Figure 21.13. Cover one eye with the hand not holding the paper. While staring at your thumb and not moving your eye, note that the text immediately above your thumb is readable while the text to either side is not. High acuity vision is limited to a visual angle slightly larger than your thumb held at arm's length. We do not normally notice this because the eyes usually move frequently, allowing different regions of the visual field to be viewed at high resolution. The visual system then integrates this information over

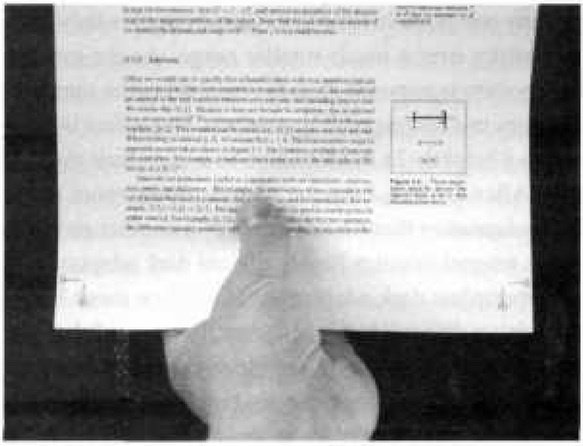


**Figure 21.13.** If you hold a page of text at arm's length and stare at your thumb, only the text near your thumb will be readable. *Photo by Peter Shirley.*

time to produce the subjective experience of the whole visual field being seen at high resolution.

There is not enough bandwidth in the human visual cortex to process the information that would result if there was a dense sampling of image intensity over the whole of the retina. The combination of variable density photoreceptor packing in the retina and a mechanism for rapid eye movements to point at areas of interest provides a way to simultaneously optimize acuity and field-of-view. Other animals have evolved different ways of balancing acuity and field-of-view that are not dependent on rapid eye movements. Some have only high acuity vision, but limited to a narrow field-of-view. Others have wide field-of-view vision, but limited ability to see detail.

The eye motions which focus areas of interest in the environment on the fovea are called *saccades*. Saccades occur very quickly. The time from a triggering stimulus to the completion of the eye movement is 150–200 ms. Most of this time is spent in the vision system planning the saccade. The actual motion takes 20 ms or so on average. The eyes are moving very quickly during a saccade, with the maximum rotational velocity often exceeding 500°/second. Between saccades, the eyes point towards an area of interest (*fixate*), taking 300 ms or so to acquire fine detail visual information. The mechanism by which multiple fixations are integrated to form an overall subjective sense of fine detail over a wide field of view is not well understood.

Figure 21.14 shows the variable packing density of cones and rods in the human retina. The cones, which are responsible for vision under normal lighting, are packed most closely at the *fovea* of the retina (Figure 21.14). When the eye

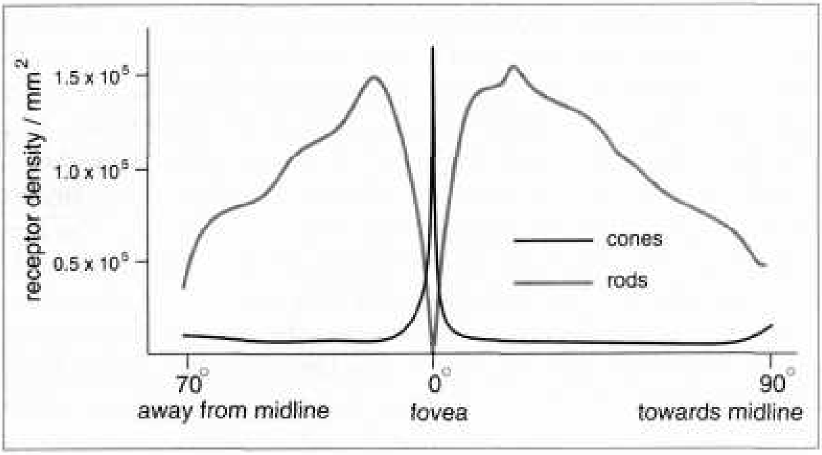


**Figure 21.14.** Density of rods and cone in the human retina (after Osterberg (1935)).

is fixated at a particular point in the environment, the image of that point falls on the fovea. The higher packing density of cones at the fovea results in a higher sampling frequency of the imaged light (see Chapter 4) and hence greater detail in the sampled pattern. Foveal vision encompasses about 1.7°, which is the same visual angle as the width of your thumb held at arm's length.

While a version of Figure 21.14 appears in most introductory texts on human visual perception, it provides only a partial explanation for the neurophysiological limitations on visual acuity. The output of individual rods and cones are pooled in various ways by neural interconnects in the eye, before the information is shipped along the optic nerve to the visual cortex.[3] This pooling filters the signal provided by the pattern of incident illumination in ways that have important impacts on the patterns of light that are detectable. In particular, the farther away from the fovea, the larger the area over which brightness is averaged. As a consequence, spatial acuity drops sharply away from the fovea. Most figures showing rod and cone packing density indicate the location of the retinal *blind spot*, where the nerve bundle carrying optical information from the eye to the brain passes through the retina, and there is no sensitivity to light. By and large, the only practical impact of the blind spot on real-world perception is its use as an illusion in introductory perception texts, since normal eye movements otherwise compensate for the temporary loss of information.

[3] All of the cells in the optic nerve and almost all cells in visual cortex have an associated retinal *receptive field*. Patterns of light hitting the retina outside of a cell's receptive field have no effect on the firing rate of that cell.

As shown in Figure 21.14, the packing density of rods drops to zero at the center of the fovea. Away from the fovea, the rod density first increases and then decreases. One result of this is that there is no foveal vision when illumination is very low. The lack of rods in the fovea can be demonstrated by observing a night sky on a moonless night, well away from any city lights. Some stars will be so dim that they will be visible if you look at at point in the sky slightly to the side of the star, but they will disappear if you look directly at them. This occurs because when you look directly at these features, the image of the features falls only on the cones in the retina, which are not sufficiently light sensitive to detect the feature. Looking slightly to the side causes the image to fall on the more light sensitive cones. Scotopic vision is also limited in acuity, in part because of the lower density of rods over much of the retina and in part because greater pooling of signals from the rods occurs in the retina in order to increase the light sensitivity of the visual information passed back to the brain.

### 21.2.5 Motion

When reading about visual perception and looking at static figures on a printed page, it is easy to forget that motion is pervasive in our visual experience. The patterns of light that fall on the retina are constantly changing due to eye and body motion and the movement of objects in the world. This section covers our ability to detect visual motion. Section 21.3.4 describes how visual motion can be used to determine geometric information about the environment. Section 21.4.3 deals with the use of motion to guide our movement through the environment.

The detectability of motion in a particular pattern of light falling on the retina is a complex function of speed, direction, pattern size, and contrast. The issue is further complicated because simultaneous contrast effects occur for motion perception in a manner similar to that observed in brightness perception. In the extreme case of a single small pattern moving against a contrasting, homogenous background, perceivable motion requires a rate of motion corresponding to 0.2°–0.3°/second of visual angle. Motion of the same pattern moving against a textured pattern is detectable at about a tenth this speed.

With this sensitivity to retinal motion, combined with the frequency and velocity of saccadic eye movements, it is surprising that the world usually appears stable and stationary when we view it. The vision system accomplishes this in three ways. Contrast sensitivity is reduced during saccades, reducing the visual effects generated by these rapid changes in eye position. Between saccades, a variety of sophisticated and complex mechanisms adjust eye position to compensate for head and body motion and the motion of objects of interest in the world. Finally, the visual system exploits information about the position of the eyes to

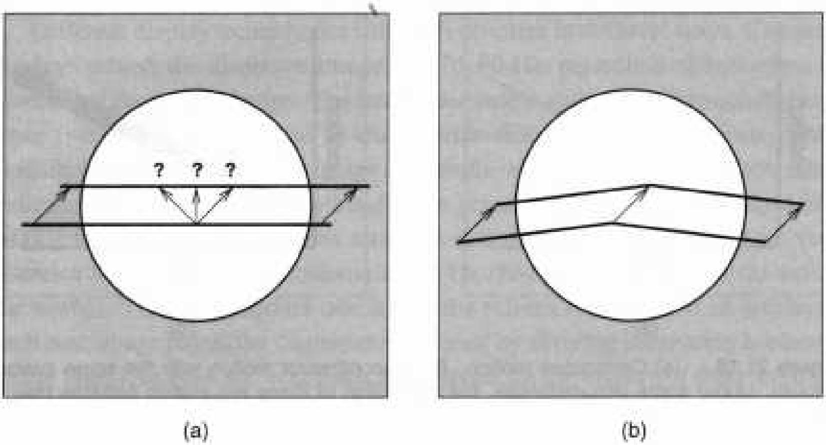


**Figure 21.15.** The aperture problem: (a) If a straight line or edge moves in such a way that its end points are hidden, the visual information is not sufficient to determine the actual motion of the line. (b) 2D motion of a line is unambiguous if there are any corners or other distinctive markings on the line.

assemble a mosaic of small patches of high resolution imagery from multiple fixations into a single, stable whole.

The motion of straight lines and edges is ambiguous if no endpoints or corners are visible, a phenomenon referred to as the *aperture problem* (Figure 21.15). The aperture problem arises because the component of motion parallel to the line or edge does not produce any visual changes. The geometry of the real world is sufficiently complex that this rarely causes difficulties in practice, except for intentional illusions such as barber poles. The simplified geometry and texturing found in some computer graphics renderings, however, has the potential to introduce inaccuracies in perceived motion.

Real-time computer graphics, film, and video would not be possible without an important perceptual phenomena: discontinuous motion, in which a series of static images are visible for discrete intervals in time and then move by discrete intervals in space, can be nearly indistinguishable from continuous motion. The effect is called *apparent motion* to highlight that the appearance of continuous motion is an illusion.

Figure 21.16 illustrates the difference between continuous motion, which is typical of the real world, and apparent motion, which is generated by almost all dynamic image display devices. The motion plotted in Figure 21.16 (b) consists of an average motion comparable to that shown in Figure 21.16 (a), modulated by a high space-time frequency that accounts for the alternation between a stationary pattern and one that moves discontinuously to a new location. Apparent percep-

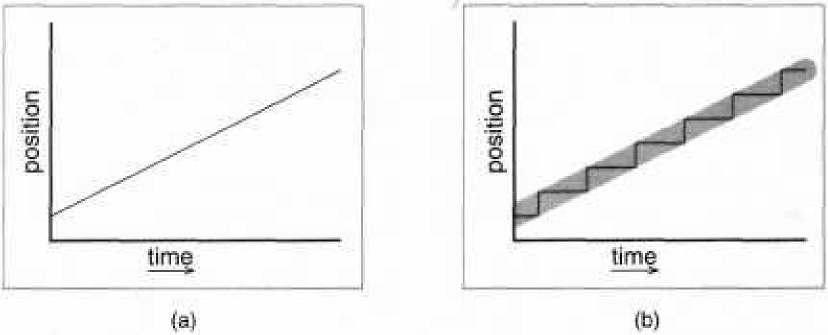


**Figure 21.16.** *(a) Continuous motion. (b) Discontinuous motion with the same average velocity. Under some circumstances, the perception of these two motion patterns may be similar.*

tion of continuous motion occurs because the visual system is insensitive to the high frequency component of the motion.

A compelling sense of apparent motion occurs when the rate at which individual images appear is above about 10 Hz, as long as the positional changes between successive images is not too great. This rate is not fast enough, however, to produce a satisfying sense of continuous motion for most image display devices. Almost all such devices introduce brightness variation as one image is switched to the next. In well lit conditions, the human visual system is sensitive to this varying brightness for rates of variations up to about 80 Hz. In lower light, detectability is present up to about 40 Hz. When the rate of alternating brightness is sufficiently high, *flicker fusion* occurs and the variation is no longer visible.

To produce a compelling sense of visual motion, an image display must therefore satisfy two separate constraints:

- Images must be updated at a rate $\geq$ 10 Hz;
- Any flicker introduced in the process of updating images must occur at a rate $\geq$ 60–80 Hz.

One solution is to require that the image update rate be greater than or equal to 60–80 Hz. In many situations, however, this is simply not possible. For computer graphics displays, the frame computation time is often substantially greater than 12–15 msec. Transmission bandwidth and limitations of older monitor technologies limit normal broadcast television to 25–30 images per second. (Some HDTV formats operate at 60 images/sec.) Movies update images at 24 frames/second due to exposure time requirements and the mechanical difficulties of physically moving film any faster than that.

Different display technologies solve this problem in different ways. Computer displays refresh the djsplayed image at ∼70–80 Hz, regardless of how often the contents of the image change. The term *frame rate* is ambiguous for such displays, since two values are required to characterize this display: *refresh rate*, which indicates the rate at which the image is redisplayed and *frame update rate*, which indicates the rate at which new images are generated for display. Standard non-HDTV broadcast television uses a refresh rate of 60 Hz (NTSC, used in North America and some other locations) or 50 Hz (PAL, used in most of the rest of the world). The frame update rate is half the refresh rate. Instead of displaying each new image twice, the display is *interlaced* by dividing alternating horizontal image lines into even and odd *fields* and alternating the display of these even and odd fields. Flicker is avoided in movies by using a mechanical shutter to blink each frame of the film three times before moving to the next frame, producing a refresh rate of 72 Hz while maintaining the frame update rate of 24 Hz.

The use of apparent motion to simulate continuous motion occasionally produces undesirable artifacts. Best known of these is the *wagon wheel illusion* in which the spokes of a rotating wheel appear to revolve in the opposite direction from what would be expected given the translational motion of the wheel. The wagon wheel illusion is an example of temporal aliasing. Spokes, or other spatially periodic patterns on a rotating disk, produce a temporally periodic signal for viewing locations that are fixed with respect to the center of the wheel or disk. Fixed frame update rates have the effect of sampling this temporally periodic signal in time. If the temporal frequency of the sampled pattern is too high, under sampling results in an aliased, lower temporal frequency appearing when the image is displayed. Under some circumstances, this distortion of temporal frequency causes a spatial distortion in which the wheel appears to move backwards. Wagon wheel illusions are more likely to occur with movies than with video, since the temporal sampling rate is lower.

Problems can also occur when apparent motion imagery is converted from one medium to another. This is of particular concern when 24 Hz movies are transferred to video. Not only does a non-interlaced format need to be translated to an interlaced format, but there is no straightforward way to move from 24 frames per second to 50 or 60 fields per second. Some high-end display devices have the ability to partially compensate for the artifacts introduced when film is converted to video.

## 21.3 Spatial Vision

One of the critical operations performed by the visual system is the estimation of geometric properties of the visible environment, since these are central to determining information about objects, locations, and events. Vision has sometimes been described as *inverse optics*, to emphasize that one function of the visual system is to invert the image formation process in order to determine the geometry, materials, and lighting in the world that produced a particular pattern on light on the retina. The central problem for a vision system is that properties of the visible environment are confounded in the patterns of light imaged on the retina. Brightness is a function of both illumination and reflectance, and can depend on environmental properties across large regions of space due to the complexities of light transport. Image locations of a projected environmental location at best can be used to constrain the position of that location to a half-line. As a consequence, it is rarely possible to uniquely determine the nature of the world that produced a particular imaged pattern of light.

Determining *surface layout*—the location and orientation of visible surfaces in the environment—is thought to be a key step in human vision. Most discussions of how the vision system extracts information about surface layout from the patterns of light it receives divide the problem into a set of *visual cues*, with each cue describing a particular visual pattern which can be used to infer properties of surface layout along with the needed rules of inference. Since surface layout can rarely be determined accurately and unambiguously from vision alone, the process of inferring surface layout usually requires additional, non-visual information. This can come from other senses or assumptions about what is likely to occur in the real world.

Visual cues are typically categorized into four categories. *Ocularmotor cues* involve information about the position and focus of the eyes. *Disparity cues* involve information extracted from viewing the same surface point with two eyes, beyond that available just from the positioning of the eyes. *Motion cues* provide information about the world that arises from either the movement of the observer or the movement of objects. *Pictorial cues* result from the process of projecting 3D surface shapes onto a 2D pattern of light that falls on the retina. This section deals with the visual cues relevant to the extraction of geometric information about individual points on surfaces. More general extraction of location and shape information is covered in Section 21.4.

### 21.3.1 Frames of Reference and Measurement Scales

Descriptions of the location and orientation of points on a visible surface must be done within the context of a particular frame of references that specifies the origin, orientation, and scaling of the coordinate system used in representing the geometric information. The human vision system uses multiple frames of reference, partially because of the different sorts of information available from different visual cues and partly because of the different purposes to which the information is put (Klatzky, 1998). *Egocentric* representations are defined with respect to the viewer's body. They can be subdivided into coordinate systems fixed to the eyes, head, or body. *Allocentric* representations, also called *exocentric* representations, are defined with respect to something external to the viewer. Allocentric frames of reference can be local to some configuration of objects in the environment or can be globally defined in terms of distinctive locations, gravity, or geographic properties.

The distance from the viewer to a particular visible location in the environment, expressed in an egocentric representation, is often referred to as *depth* in the perception literature. Surface orientation can be represented in either egocentric or allocentric coordinates. In egocentric representations of orientation, the term *slant* is used to refer to the angle between the line of sight to the point and the surface normal at the point, while the term *tilt* refers to the orientation of the projection of the surface normal onto a plane perpendicular to the line of sight.

Distance and orientation can be expressed in a variety of *measurement scales*. *Absolute* descriptions are specified using a standard that is not part of the sensed information itself. These can be culturally defined standards (e.g, meters), or standards relative to the viewer's body (e.g., eye height, the width of one's shoulders). *Relative* descriptions relate one perceived geometric property to another (e.g., point a is twice as far away as point b). *Ordinal* descriptions are a special

| *Cue* | a | r | o | *Requirements for absolute depth* |
|---|---|---|---|---|
| Accommodation | x | x | x | very limited range |
| Binocular convergence | x | x | x | limited range |
| Binocular disparity | - | x | x | limited range |
| Linear perspective, height in picture, horizon ratio | x | x | x | requires viewpoint height |
| Familiar size | x | x | x | |
| Relative size | - | x | x | |
| Aerial perspective | ? | x | x | adaptation to local conditions |
| Absolute motion parallax | ? | x | x | requires viewpoint velocity |
| Relative motion parallax | - | - | x | |
| Texture gradients | - | x | - | |
| Shading | - | x | - | |
| Occlusion | - | - | x | |

**Figure 21.17.** Common visual cues for absolute (a), relative (r), and ordinal (o) depth.

case of relative measure in which the sign, but not the magnitude, of the relation is all that is represented. Figure 21.17 provides a list of the most commonly considered visual cues, along with a characterization of the sorts of information they can potentially provide.

### 21.3.2 Ocularmotor Cues

Ocularmotor information about depth results directly from the muscular control of the eyes. There are two distinct types of ocularmotor information. *Accommodation* is the process by which the eye optically focuses at a particular distance. *Convergence* (often referred to as *vergence*) is the process by which the two eyes are pointed towards the same point in three-dimensional space. Both accommodation and convergence have the potential to provide absolute information about depth.

Physiologically, focusing in the human eye is accomplished by distorting the shape of the lens at the front of the eye. The vision system can infer depth from the amount of this distortion. Accommodation is a relatively weak cue to distance and is ineffective beyond about 2 m. Most people have increasing difficultly in focusing over a range of distances as they get beyond about 45 years old. For them, accommodation becomes even less effective.

Those not familiar with the specifics of visual perception sometimes confuse depth estimation from accommodation with depth information arising out of the

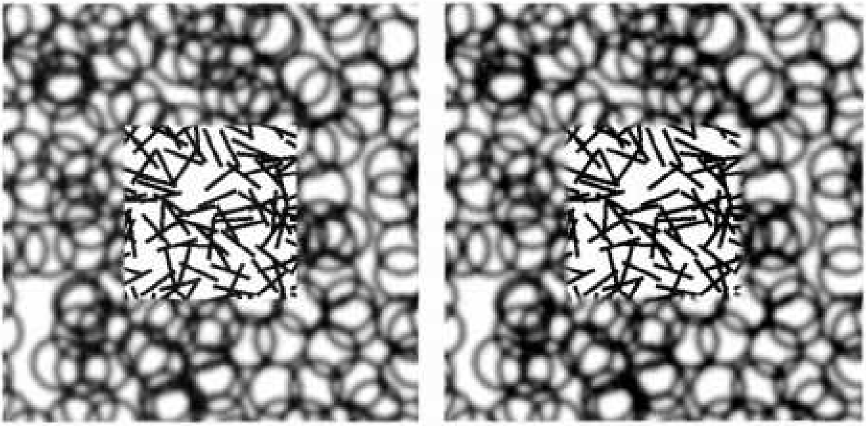


**Figure 21.18.** Does the central square appear in front of the pattern of circles or is it seen as appearing through a square hole in the pattern of circles? The only difference in the two images is the sharpness of the edge between the line and circle patterns (Marshall, Burbeck, Arely, Rolland, and Martin (1999), used by permission).

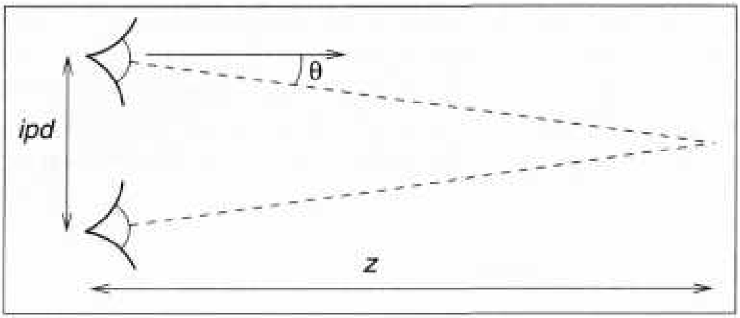


**Figure 21.19.** The *vergence* of the two eyes provides information about the distance to the point on which the eyes are fixated.

blur associated with limited depth-of-field in the eye. The accommodation depth cue provides information about the distance to that portion of the visual field that it is in focus. It does not depend on the degree to which other portions of the visual field are out of focus, other than that blur is used by the visual system to adjust focus. Depth-of-field does seem to provide a degree of ordinal depth information (Figure 21.18), though this effect has received only limited investigation.

If two eyes fixate on the same point in space, trigonometry can be used to determine the distance from the viewer to the viewed location (Figure 21.19). For the simplest case, in which the point of interest is directly in front of the viewer,

$$z = \frac{ipd/2}{\tan\theta}, \tag{21.5}$$

where $z$ is the distance to a point in the world, *ipd* is the *interpupillary distance* indicating the distance between the eyes, and $\theta$ is the *vergence angle* indicating the orientation of the eyes relative to straight ahead. For small $\theta$, which is the case for the geometric configuration of human eyes, $\tan\theta \approx \theta$ when $\theta$ is expressed in radians. Thus, differences in vergence angle specify differences in depth by the following relationship:

$$\Delta\theta \approx \frac{ipd}{2} \cdot \frac{1}{\Delta z}. \tag{21.6}$$

As $\theta \to 0$ in uniform steps, $\Delta z$ gets increasingly larger. This means that stereo vision is less sensitive to changes in depth as the overall depth increases. Convergence in fact only provides information on absolute depth for distances out to a few meters. Beyond that, changes in distance produce changes in vergence angle that are too small to be useful.

There is an interaction between accommodation and convergence in the human visual system: accommodation is used to help determine the appropriate vergence angle, while vergence angle is used to help set the focus distance. Normally, this helps the visual system when there is uncertainty is setting either accommodation or vergence. However, stereographic computer displays break the relationship between focus and convergence that occurs in the real world, leading to a number of perceptual difficulties (Wann, Rushton, & Mon-Williams, 1995) .

### 21.3.3 Binocular Disparity

The vergence angle of the eyes when fixated on a common point in space is only one of the ways that the visual system is able to determine depth from binocular stereo. A second mechanism involves a comparison of the retinal images in the two eyes and does not require information about where the eyes are pointed. A simple example demonstrates the effect. Hold your arm straight out in front of you, with your thumb pointed up. Stare at your thumb and then close one eye. Now, simultaneously open the closed eye and close the open eye. Your thumb will appear to be more or less stationary, while the more distant surfaces seen behind your thumb will appear to move from side to side (Figure 21.20). The change in retinal position of points in the scene between the left and right eyes is called *disparity*.

The binocular disparity cue requires that the vision system be able to match the image of points in the world in one eye with the imaged locations of those points in the other eye, a process referred to as the *correspondence problem*. This is a relatively complicated process and is only partially understood. Once correspondences have been established, the relative positions on which particular

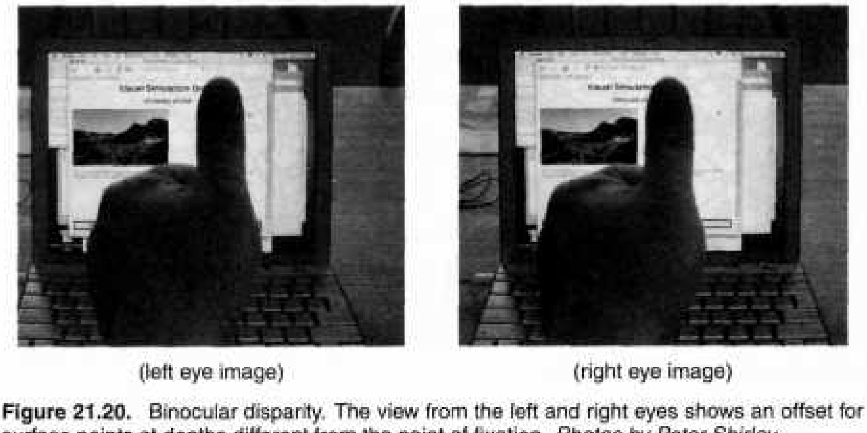

(left eye image) (right eye image)

**Figure 21.20.** Binocular disparity. The view from the left and right eyes shows an offset for surface points at depths different from the point of fixation. *Photos by Peter Shirley.*

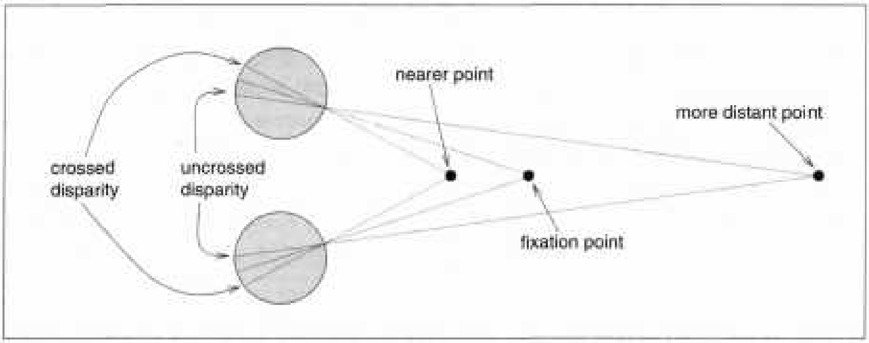


**Figure 21.21.** Near the line of sight, surface points nearer than the fixation point produce disparities in the opposite direction from those associated with surface points more distant than the fixation point.

points in the world project onto the left and right retinas indicate whether the points are closer than or farther away than the point of fixation. *Crossed disparity* occurs when the corresponding points are displaced outward relative to the fovea and indicates that the surface point is closer than the point of fixation. *Uncrossed disparity* occurs when the corresponding points are displaced inward relative to the fovea and indicates that the surface point is farther away than the point of fixation (Figure 21.21).[4] Binocular disparity is a relative depth cue, but it can provide information about absolute depth when scaled by convergence. Equation 21.5 applies to binocular disparity as well as binocular convergence. As with convergence, the sensitivity of binocular disparity to changes in depth decreases with depth.

### 21.3.4 Motion Cues

Relative motion between the eyes and visible surfaces will produce changes in the image of those surfaces on the retina. Three-dimensional relative motion between the eye and a surface point produces two-dimensional motion of the projection of the surface point on the retina. This retinal motion is given the name *optic flow*. Optic flow serves as the basis for several types of depth cues. In addition, optic flow can be used to determine information about how a person is moving in the world and whether or not a collision is imminent (Section 21.4.3).

If a person moves to the side while continuing to fixate on some surface point, then optic flow provides information about depth similar to stereo disparity. This

[4] Technically, crossed and uncrossed disparities indicate that the surface point generating the disparity is closer to or farther away from the *horopter*. The horopter is not a fixed distance away from the eyes but rather it is a curved surface passing through the point of fixation.

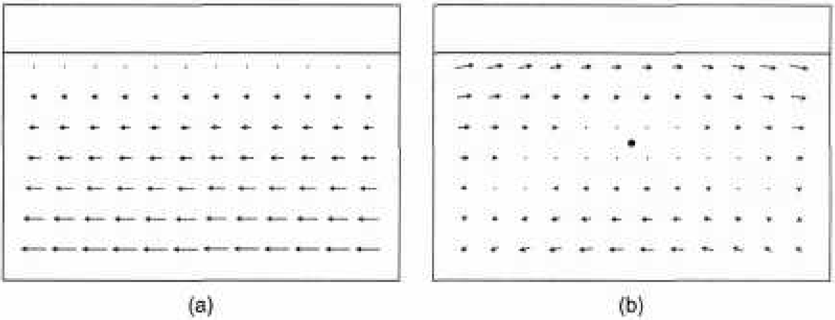


**Figure 21.22.** (a) Motion parallax generated by sideways movement to the right while looking at an extended ground plane. (b) The same motion, with eye tracking of the fixation point.

is referred to as *motion parallax*. For other surface points that project to retinal locations near the fixation point, zero optic flow indicates a depth equivalent to the fixation point; flow in the opposite direction to head translation indicates nearer points, equivalent to crossed disparity; and flow in the same direction as head translation indicates farther points, equivalent to uncrossed disparity (Figure 21.22). Motion parallax is a powerful cue to relative depth. In principle, motion parallax can provide absolute depth information if the visual system has access to information about the velocity of head motion. In practice, motion parallax appears at best to be a weak cue for absolute depth.

In addition to egocentric depth information due to motion parallax, visual motion can also provide information about the three-dimensional shape of objects moving relative to the viewer. In the perception literature, this is known as the *kinetic depth effect*. In computer vision, it is referred to as *structure-from-motion*. The kinetic depth effect presumes that one component of object motion is *rotation in depth*, meaning that there is a component of rotation around an axis perpendicular to the line of sight.

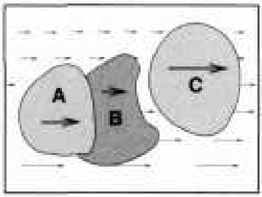


**Figure 21.23.** Discontinuities in optic flow signal surface boundaries. In many cases, the sign of the depth change (i.e., the ordinal depth) can be determined.

Optic flow can also provide information about the shape and location of surface boundaries, as shown in Figure 21.23. Spatial discontinuities in optic flow almost always either correspond to depth discontinuities or result from independently moving objects. Simple comparisons of the magnitude of optic flow are insufficient to determine the sign of depth changes, except in the special case of a viewer moving through an otherwise static world. Even when independently moving objects are present, however, the sign of the change in depth across surface boundaries can often be determined by other means. Motion often changes the portion of the more distant surface visible at surface boundaries. The appearance (*accretion*) or disappearance (*deletion*) of surface texture occurs because the nearer, occlud*ing* surface progressively uncovers or covers portions of the more distant, occlud*ed* surface. Comparisons of the motion of surface texture to either side of a boundary can also be used to infer ordinal depth, even in the absence of accretion or deletion of the texture. Discontinuities in optic flow and accretion/deletion of surface texture are referred to as *dynamic occlusion* cues and are another powerful source of visual information about the spatial structure of the environment.

The speed that a viewer is traveling relative to points in the world cannot be determined from visual motion alone (see Section 21.4.3). Despite this limitation, it is possible to use visual information to determine the time it will take to reach a visible point in the world even when speed cannot be determined. When velocity is constant, *time-to-contact* (often referred to as *time-to-collision*) is given by the retinal size of an entity towards which the observer is moving, divided by the rate at which that image size is increasing.[5] In the biological vision literature, this is often called the $\tau$ *function* (Lee & Reddish, 1981). If distance information to the structure in the world on which the time-to-collision estimate is based is available, then this can be used to determine speed.

### 21.3.5 Pictorial Cues

An image can contain much information about the spatial structure of the world from which it arose, even in the absence of binocular stereo or motion. As evidence for this, note that the world still appears three-dimensional even if we close one eye, hold our head stationary, and nothing moves in the environment. (As discussed in Section 21.5, the situation is more complicated in the case of photographs and other displayed images.) There are three classes of such *pictorial depth cues*. The best known of these involve *linear perspective*. There are also a number of *occlusion cues* that provide information about ordinal depth even in the absence of perspective. Finally, *illumination cues* involving shading, shadows and interreflections, and aerial perspective also provide visual information about spatial layout.

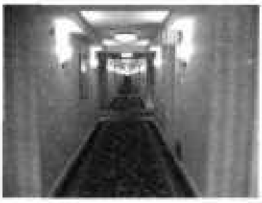


**Figure 21.24.** The classical linear perspective effects include object size scaled by distance, the convergence of parallel lines, the ground plane extending to a visible horizon, and position on the ground plane relative to the horizon. *Image courtesy Sam Pullara.*

The term *linear perspective* is often used to refer to properties of images involving object size in the image scaled by distance, the convergence of parallel lines, the ground plane extending to a visible horizon, and the relationship between the distance to objects on the ground plane and the image location of those objects relative to the horizon (Figure 21.24). More formally, linear perspective cues are those visual cues which exploit the fact that under perspective projection, the image location onto which points in the world are projected is scaled by $\frac{1}{z}$,

[5] The terms time-to-collision and time-to-contact are misleading, since contact will only occur if the viewer's trajectory actually passes through or near the entity under view.

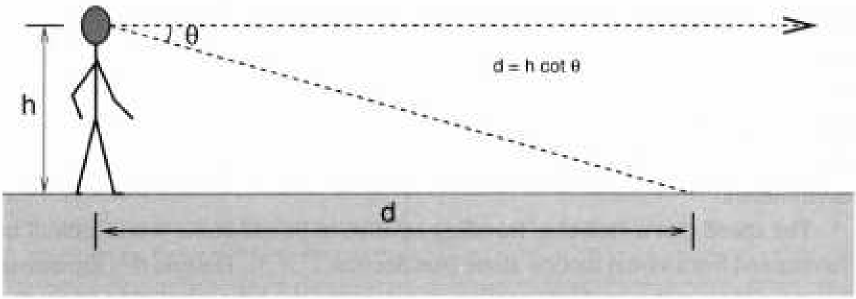


**Figure 21.25.** Absolute distance to locations on the ground plane can be determined based on declination angle from the horizon and eye height.

where $z$ is the distance from the point of projection to the point in the environment. Direct consequences of this relationship are that points that are farther away are projected to points closer to the center of the image (convergence of parallel lines) and that the spacing between the image of points in the world decreases for more distant world points (object size in the image is scaled by distance).[6] The fact that the image of an infinite flat surface in the world ends at a finite horizon is explained by examining the perspective projection equation as $z \to \infty$.

With the exception of size-related effects described in Section 21.4.2, most pictorial depth cues involving linear perspective depend on objects of interest being in contact with a ground plane. In effect, these cues estimate not the distance to the objects but, instead, the distance to the contact point on the ground plane. Assuming observer and object are both on top of a horizontal ground plane, then locations on the ground plane lower in the view will be close. Figure 21.25 illustrates this effect quantitatively. For a viewpoint $h$ above the ground and an *angle of declination* $\theta$ between the horizon and a point of interest on the ground, the point in question is a distance $d = h \cot \theta$ from the point at which the observer is standing. The angle of declination provides relative depth information for arbitrary fixed viewpoints and can provide absolute depth when scaling by eye height $(h)$ is possible.

While the human visual system almost certainly makes use of angle of declination as a depth cue, the exact mechanisms used to acquire the needed information are not clear. The angle $\theta$ could be obtained relative to either gravity or the visible horizon. There is some evidence that both are used in human vision. Eye height $h$ could be based on posture, visually determined by looking at the ground at one's feet, or learned by experience and presumed to be constant. While a

[6] The actual mathematics for analyzing the specifics of biological vision are different, since eyes are not well approximated by the planar projection formulation used in computer graphics and most other imaging applications.

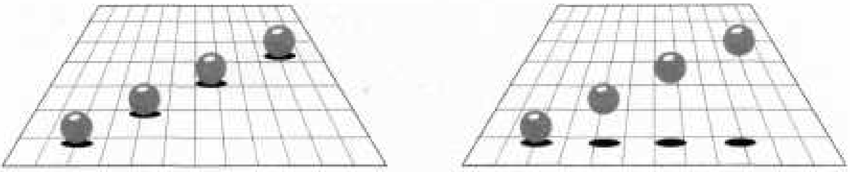


**Figure 21.26.** Shadows can indirectly function as a depth cue by associating the depth of an object with a location on the ground plane (after Kersten, Mamassian, and Knill (1997)).

number of researchers have investigated this issue, if and how these values are determined is not yet known with certainty.

Shadows provide a variety of types of information about three-dimensional spatial layout. *Attached shadows* indicate that an object is in contact with another surface, often consisting of the ground plane. *Detached shadows* indicate that an object is close to some surface, but not in contact with that surface. Shadows can serve as an indirect depth cue by causing an object to appear at the depth of the location of the shadow on the ground plane (Yonas, Goldsmith, & Hallstrom, 1978). When utilizing this cue, the visual system seems to make the assumption that light is coming from directly above (Figure 21.26).

Vision provides information about surface orientation as well as distance. It is convenient to represent visually determined surface orientation in terms of *tilt*, defined as the orientation in the image of the projection of the surface normal, and *slant*, defined as the angle between the surface normal and the line of sight.

A visible surface horizon can be used to find the orientation of an (effectively infinite) surface relative to the viewer. Determining tilt is straightforward, since the tilt of the surface is the orientation of the visible horizon. Slant can be recovered as well, since the lines of sight from the eye point to the horizon define a plane parallel to the surface. In many situations, either the surface horizon is not visible or the surface is small enough that its far edge does not correspond to an actual horizon. In such cases, visible texture can still be used to estimate orientation.

In the context of perception, the term *texture* refers to visual patterns consisting of sub-patterns replicated over a surface. The sub-patterns and their distribution can be fixed and regular, as for a checkerboard, or consistent in a more statistical sense, as in the view of a grassy field.[7] When a textured surface is viewed from an oblique angle, the projected view of the texture is distorted relative to the actual markings on the surface. Two quite distinct types of distortions occur (Knill, 1998), both affected by the amount of slant. The position and size

[7] In computer graphics, the term *texture* has a different meaning, referring to any image that is applied to a surface as part of the rendering process.

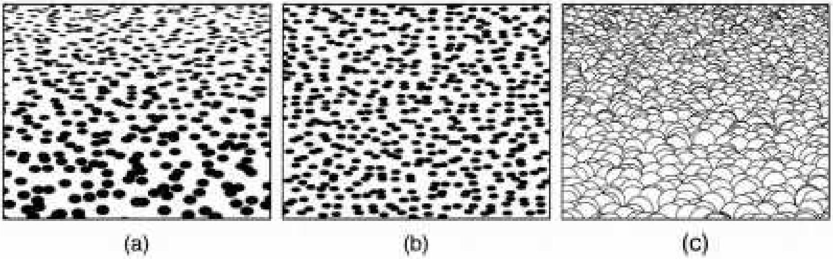


**Figure 21.27.** Texture cues for slant. (a) Near surface exhibiting compression and texture gradient; (b) distant surface exhibiting only compression; (c) variability in appearance of near surface with regular geometric variability.

of texture elements are subject to the linear perspective effects described above. This produces a *texture gradient* (Gibson, 1950) due to both element size and spacing decreasing with distance (Figure 21.27(a)). Both the image of individual texture elements and the distribution of elements are *foreshortened* under oblique viewing (Figure 21.27(b)). This produces a compression in the direction of tilt. For example, an obliquely viewed circle appears as an ellipse, with the ratio of the minor to major axes equal to the cosine of the slant. Note that foreshortening itself is not a result of linear perspective, though in practice both linear perspective and foreshortening provide information about slant.[8]

For texture gradients to serve as a cue to surface slant, the average size and spacing of texture elements must be constant over the textured surface. If spatial variability in size and spacing in the image is not due in its entirely to the projection process, then attempts to invert the effects of projection will produce incorrect inferences about surface orientation. Likewise, the foreshortening cue fails if the shape of texture elements is not isotropic, since then asymmetric texture element image shapes would occur in situations not associated with oblique viewing. These are examples of the assumptions often required in order for spatial visual cues to be effective. Such assumptions are reasonable to the degree that they reflect commonly occurring properties of the world.

Shading also provides information about surface shape (Figure 21.28). The brightness of viewed points on a surface depends on the surface reflectance and the orientation of the surface with respect to directional light sources and the observation point. When the relative position of an object, viewing direction, and illumination direction remain fixed, changes in brightness over a constant reflectance surface are indications of changes in the orientation of the surface of

[8] A third form of visual distortion occurs when surfaces with distinct 3D surface relief are viewed obliquely (Leung & Malik, 1997), as shown in Figure 21.27(c). Nothing is currently know about if or how this effect might be used by the human vision system to determine slant.

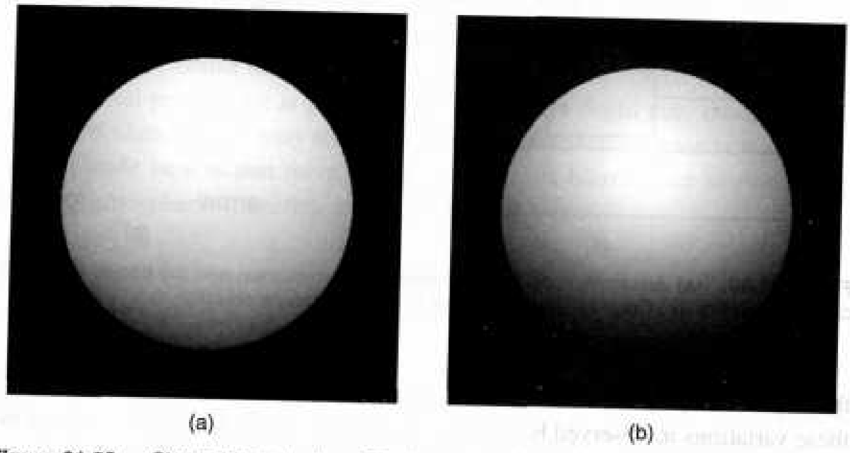


**Figure 21.28.** Shape-from-shading. The images in (a) and (b) appear to have different 3D shapes because of differences in the rate of change of brightness over their surfaces.

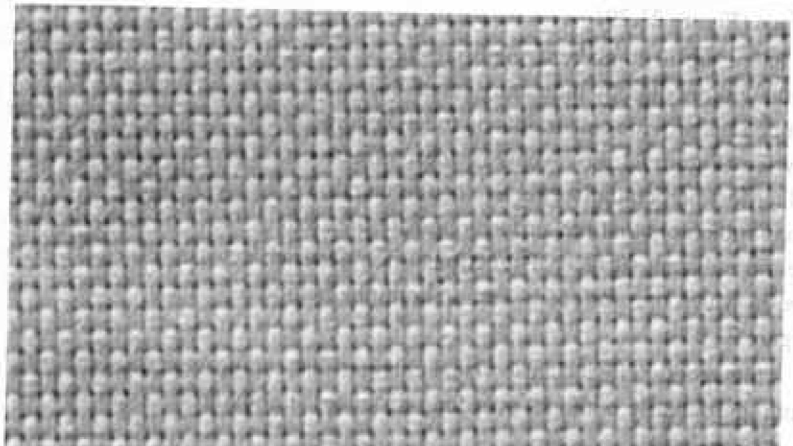


**Figure 21.29.** Shading can generate a strong perception of three-dimensional shape. In this figure, the effect is stronger if you view the image from several meters away using one eye. It becomes yet stronger if you place a piece of cardboard in front of the figure with a hole cut out slightly smaller than the picture (see Section 21.5). *Figure courtesy of Albert Yonas.* (See also Color Plate XIX.)

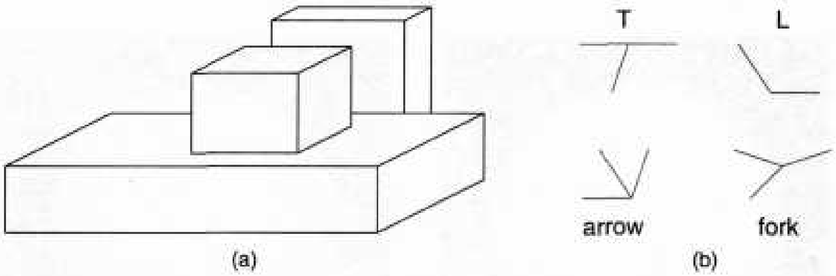


**Figure 21.30.** (a) Junctions provide information about occlusion and the convexity or concavity of corners. (b) Common junction types for planar surface objects.

the object. *Shape-from-shading* is the process of recovering surface shape from these variations in observed brightness. It is almost never possible to recover the actual orientation of surfaces from shading alone, though shading can often be combined with other cues to provide an effective indication of surface shape. For surfaces with fine-scale geometric variability, shading can provide a compelling three-dimensional appearance, even for an image rendered on a two-dimensional surface (Figure 21.29).

There are a number of pictorial cues that yield ordinal information about depth, without directly indicating actual distance. In line drawings, different types of junctions provide constraints on the 3D geometry that could have generated the drawing (Figure 21.30). Many of these effects occur in more natural images as well. Most perceptually effective of the junction cues are *T-junctions*, which are strong indicators that the surface opposite the stem of the T is occluding at least one more distant surface. T-junctions often generate a sense of *amodal completion*, in which one surface is seen to continue behind a nearer, occluding surface (Figure 21.31).

Atmospheric effects cause visual changes that can provide information about depth, particularly outdoors over long distances. Leonardo da Vinci was the first

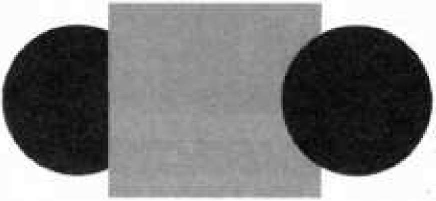


**Figure 21.31.** T-junctions cause the left disk to appear to be continuing behind the rectangle, while the right disk appears in front of the rectangle which is seen to continue behind the disk.

to describe *aerial perspective* (also called *atmospheric perspective*), in which scattering reduces the contrast of distant portions of the scene and causes them to appear more bluish than if they were nearer (da Vinci, 1970) (see Color Plate XX). Aerial perspective is predominately a relative depth cue, though there is some speculation that it may affect perception of absolute distance as well. While many people believe that more distant objects look blurrier due to atmospheric effects, atmospheric scattering actually causes little blur.

## 21.4 Objects, Locations, and Events

While there is fairly wide agreement among current vision scientists that the purpose of vision is to extract information about objects, locations, and events, there is little consensus on the key features of what information is extracted, how it is extracted, or how the information is used to perform tasks. Significant controversies exist about the nature of object recognition and the potential interactions between object recognition and other aspects of perception. Most of what we know about location involves low-level spatial vision, not issues associated with spatial relationships between complex objects or the visual processes required to navigate in complex environments. We know a fair amount about how people perceive their speed and heading as they move through the world, but have only a limited understanding of actual event perception. Visual attention involves aspects of the perception of objects, locations, and events. While there is much data about the phenomenology of visual attention for relatively simple and well controlled stimuli, we know much less about how visual attention serves high-level perceptual goals.

### 21.4.1 Object Recognition

Object recognition involves segregating an image into constituent parts corresponding to distinct physical entities and determining the identity of those entities. Figure 21.32 illustrates a few of the complexities associated with this process. We have little difficulty recognizing that the image on the left is some sort of vehicle, even though we have never before seen this particular view of a vehicle nor do most of us typically associate vehicles with this context. The image on the right is less easily recognizable until the page is turned upside down, indicating an orientational preference in human object recognition.

Object recognition is thought to involve two, fairly distinct steps. The first step organizes the visual field into *groupings* likely to correspond to objects and

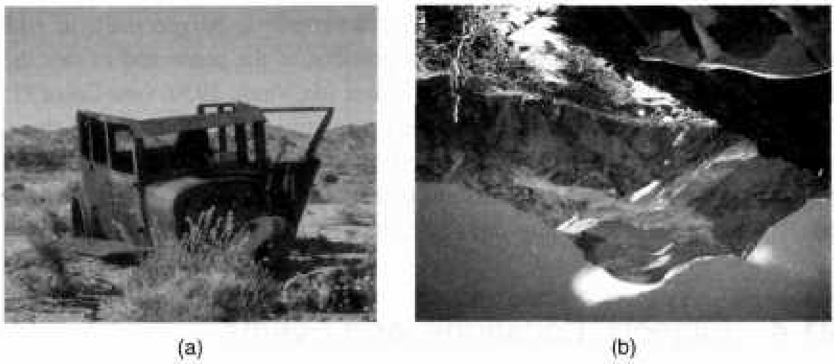


**Figure 21.32.** The complexities of object recognition. (a) We recognize a vehicle-like object even though we have likely never seen this particular view of a vehicle before. (b) The image is hard to recognize based on a quick view. It becomes much easier to recognize if the book is turned upside down.

surfaces. These grouping processes are very powerful (see Figure 21.33), though there is little or no conscious awareness of the low-level image features that generate the grouping effect.[9] Grouping is based on the complex interaction of proximity, similarities in the brightness, color, shape, and orientation of primitive structures in the image, common motion, and a variety of more complex relationships.

The second step in object recognition is to interpret groupings as identified objects. A computational analysis suggests that there are a number of distinctly

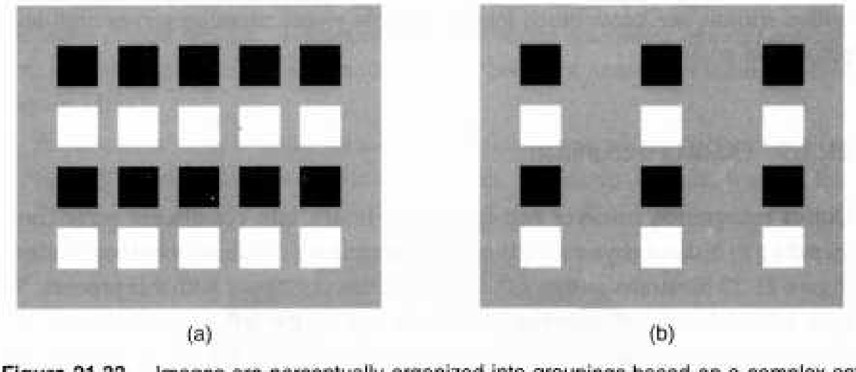


**Figure 21.33.** Images are perceptually organized into groupings based on a complex set of similarity and organizational criteria. (a) Similarity in brightness results in four horizontal groupings. (b) Proximity resulting in three vertical groupings.

[9] The most common form of visual camouflage involves adding visual textures that fool the perceptual grouping processes so that the view of the world cannot be organized in a way that separates out the object being camouflaged.

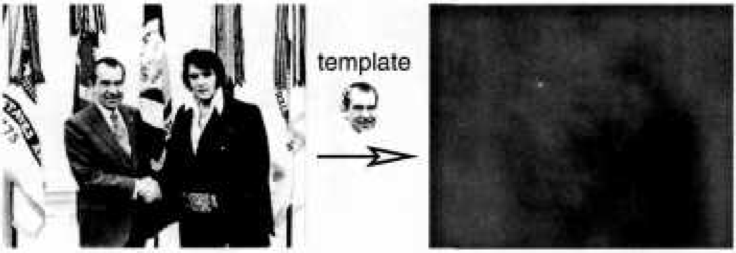


**Figure 21.34.** Template matching. The bright spot in the right image indicates the best match location to the template in the left image. *Image courtesy National Archives and Records Administration.*

different ways in which an object can be identified. The perceptual data is unclear as to which of these are actually used in human vision. Object recognition requires that the vision system have available to it descriptions of each class of object sufficient to discriminate each class from all others. Theories of object recognition differ in the nature of the information describing each class and the mechanisms used to match these descriptions to actual views of the world.

Three general types of descriptions are possible. *Templates* represent object classes in terms of prototypical views of objects in each class. Figure 21.34 shows a simple example. *Structural descriptions* represent object classes in terms of distinctive features of each class likely to be easily detected in views of the object, along with information about the geometric relationships between the features. Structural descriptions can either be represented in 2D or 3D. For 2D models of objects types, there must be a separate description for each distinctly different potential view of the object. For 3D models, two distinct forms of matching strategies are possible. In one, the three-dimensional structure of the viewed object is determined prior to classification using whatever spatial cues are available and then this 3D description of the view is matched to 3D prototypes of known objects. The other possibility is that some mechanism allows the determination of the orientation of the yet-to-be identified object under view. This orientation information is used to rotate and project potential 3D descriptions in a way that allows a 2D matching of the description and the viewed object. Finally, the last option for describing the properties of object classes involves *invariant features* which describe classes of objects in terms of more generic geometric properties, particularly those that are likely be be insensitive to different views of the object.

### 21.4.2 Size and Distance

In the absence of more definitive information about depth, objects which project onto a larger area of the retina are seen as closer compared with objects which

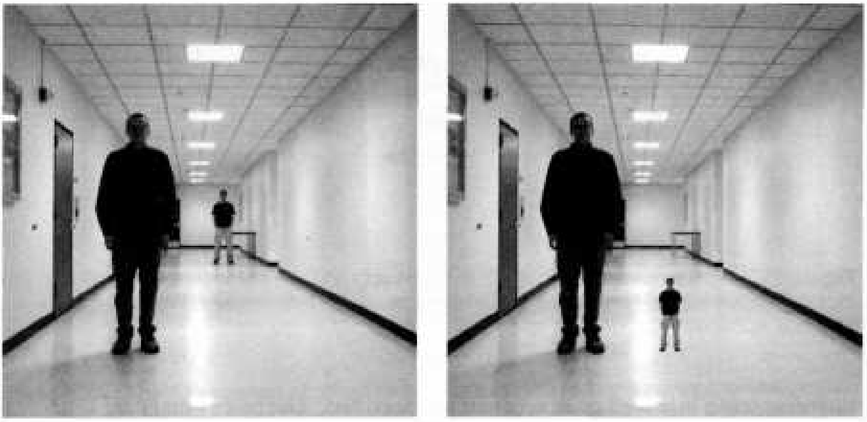


**Figure 21.35.** Left: perspective and familiar size cues are consistent. Right: perspective and familiar size cues are inconsistent. *Figure by Peter Shirley, Scott Kuhl, and J. Dylan Lacewell.*

project to a smaller retinal area, an effect called *relative size*. A more powerful cue involves *familiar size*, which can provide information for absolute distance to recognizable objects of known size. The strength of familiar size as a depth cue can be seen in illusions such as Figure 21.35, in which it is put in conflict with ground-plane, perspective-based depth cues. Familiar size is one part of the *size-distance* relationship, relating the physical size of an object, the optical size of the same object projected onto the retina, and the distance of the object from the eye (Figure 21.36).

When objects are sitting on top of a flat ground plane, additional sources for depth information become available, particularly when the horizon is either vis-

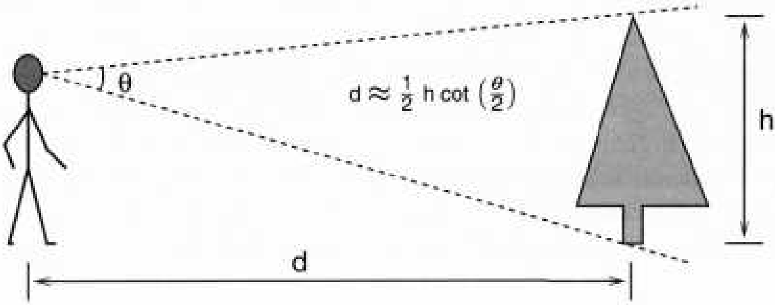
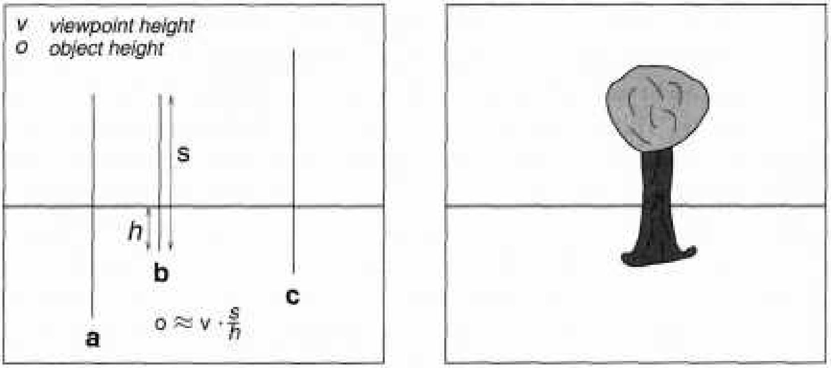

θ
$d \approx \frac{1}{2}\, h \cot\left(\frac{\theta}{2}\right)$
h
d

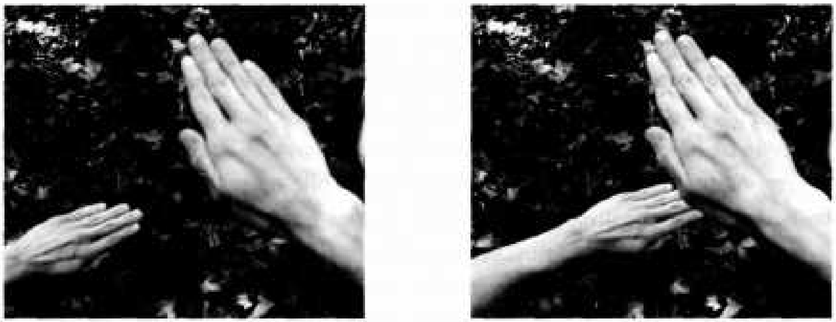
**Figure 21.36.** The *size-distance relationship* allows the distance to objects of known size to be determined based on the visual angle subtended by the object. Likewise, the size of an object at a know distance can be determined based on the visual angle subtended by the object.



(a) (b)

**Figure 21.37.** (a) The *horizon ratio* can be used to determine depth by comparing the visible portion of an object below the horizon to the total vertical visible extent of the object. (b) A real-world example.

ible or can be derived from other perspective information. The angle of declination to the contact point on the ground is a relative depth cue and provides absolute egocentric distance when scaled by eye height, as previously shown in Figure 21.25. The *horizon ratio*, in which the total visible height of an object is compared with the visible extent of that portion of the object appearing below the horizon, can be used to determine the actual size of objects, even when the distance to the objects is not known (Figure 21.37). Underlying the horizon ratio is the fact that for a flat ground plane, the line of sight to the horizon intersects objects at a position that is exactly an eye height above the ground.



(a) (b)

**Figure 21.38.** (a) Size constancy makes hands positioned at different distances from the eye appear to be nearly the same size for real-world viewing, even though the retinal sizes are quite different. (b) The effect is less strong when one hand is partially occluded by the other, particularly when one eye is closed. *Figure by Peter Shirley and Pat Moulis.*

The human visual system is sufficiently able to determine the absolute size of most viewed objects; our perception of size is dominated by the the actual physical size, and we have almost no conscious awareness of the corresponding retinal size of objects. This is similar to lightness constancy, discussed earlier, in that our perception is dominated by inferred properties of the world, not the low level features actually sensed by photoreceptors in the retina. Gregory (1997) describes a simple example of *size constancy*. Hold your two hands out in front of you, one at arms length and the other at half that distance away from you (Figure 21.38(a)). Your two hands will look almost the same size, even though the retinal sizes differ by a factor of two. The effect is much less strong if the nearer hand partially occludes the more distant hand, particularly if you close one eye (Figure 21.38(b)). The visual system also exhibits *shape constancy*, where the perception of geometric structure is close to actual object geometry than might be expected given the distortions of the retinal image due to perspective (Figure 21.39).

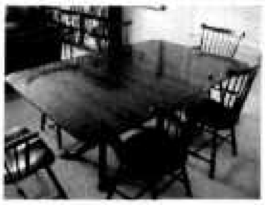


**Figure 21.39.** Shape constancy—the table looks rectangular even though its shape in the image is an irregular four sided polygon.

### 21.4.3 Events

Most aspects of event perception are beyond the scope of this chapter, since they involve complex non-visual cognitive processes. Three types of event perception are primarily visual, however, and are also of clear relevance to computer graphics. Vision is capable of providing information about how a person is moving in the world, the existence of independently moving objects in the world, and the potential for collisions either due to observer motion or due to objects moving towards the observer.

Vision can be used to determine rotation and the direction of translation relative to the environment. The simplest case involves movement towards a flat surface oriented perpendicularly to the line of sight. Presuming that there is sufficient surface texture to enable the recovery of optic flow, the flow field will form a symmetric pattern as shown in Figure 21.40(a). The location in the field of view of the *focus of expansion* of the flow field will have an associated line of sight corresponding to the direction of translation. While optic flow can be used to visually determine the direction of motion, it does not contain enough information to determine speed. To see this, consider the situation in which the world is made twice as large and the viewer moves twice as fast. The decrease in the magnitude of flow values due to the doubling of distances is exactly compensated for by the increase in the magnitude of flow values due to the doubling of velocity, resulting in an identical flow field.

Figure 21.40(b) shows the optic flow field resulting from the viewer (or more accurately, the viewer's eyes) rotating around the vertical axis. Unlike the situa-

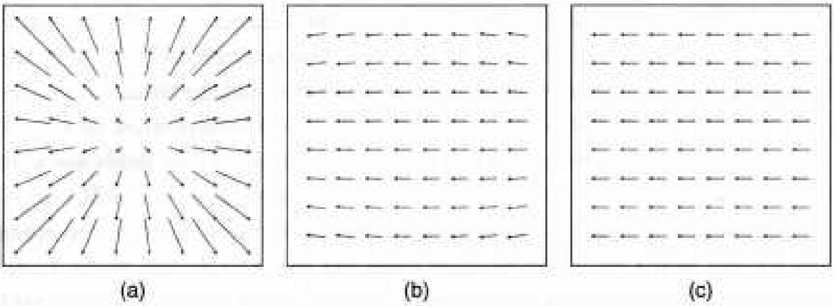


**Figure 21.40.** (a) Movement towards a flat, textured surface produces an expanding flow field, with the *focus of expansion* indicating the line of sight corresponding to the direction of motion. (b) The flow field resulting from rotation around the vertical axis while viewing a flat surface oriented perpendicularly to the line of sight. (c) The flow field resulting from translation parallel to a flat, textured surface.

tion with respect to translational motion, optic flow provides sufficient information to determine both the axis of rotation and the (angular) speed of rotation. The practical problem in exploiting this is that the flow resulting from pure rotational motion around an axis perpendicular to the line of sight is quite similar to the flow resulting from pure translation in the direction that is perpendicular to both the line of sight and this rotational axis, making it difficult to visually discriminate between the two very different types of motion (Figure 21.40(c)). Figure 21.41 shows the optical flow patterns generated by movement through a more realistic environment.

If a viewer is completely stationary, visual detection of moving objects is easy, since such objects will be associated with the only non-zero optic flow in the field

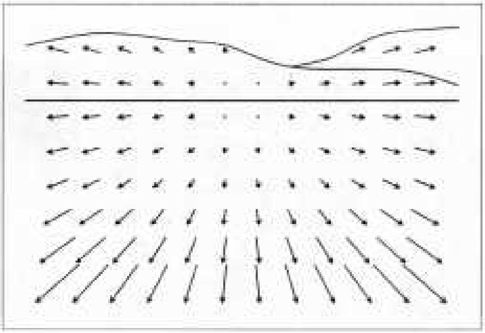


**Figure 21.41.** The optic flow generated by moving through an otherwise static environment provides information about both the motion relative to the environment and the distances to points in the environment. In this case, the direction of view is depressed from the horizon, but as indicated by the focus of expansion, the motion is parallel to the ground plane.

of view. The situation is considerably more complicated when the observer is moving, since the visual field will be dominated by non-zero flow, most or all of which is due to relative motion between the observer and the static environment (Thompson & Pong, 1990). In such cases, the visual system must be sensitive to patterns in the optic flow field that are inconsistent with flow fields associated with observer movement relative to a static environment (Figure 21.42).

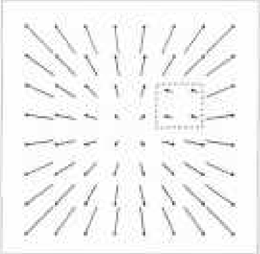


**Figure 21.42.** Visual detection of moving objects from a moving observation point requires recognizing patterns in the optic flow that cannot be associated with motion through a static environment.

Section 21.3.4 described how vision can be used to determine time to contact with a point in the environment even when the speed of motion is not known. Assuming a viewer moving with a straight, constant-speed trajectory and no independently moving objects in the world, contact will be made with whatever surface is in the direction of the line of sight corresponding to the focus of expansion at a time indicated by the $\tau$ relationship. An independently moving object complicate the matter of determining if a collision will in fact occur. Sailors use a method for detecting potential collisions that may also be employed in the human visual system: for non-accelerating straight-line motion, collisions will occur with objects that are visually expanding but otherwise remain visually stationary in the egocentric frame of reference.

One form of more complex event perception merits discussion here, since it is so important in interactive computer graphics. People are particularly sensitive to motion corresponding to human movement. Locomotion can be recognized when the only features visible are lights on the walker's joints (Johansson, 1973). Such *moving light displays* are often even sufficient to recognize properties such as the sex of the walker and the weight of the load that the walker may be carrying. In computer graphics renderings, viewers will notice even small inaccuracies in animated characters, particularly if they are intended to mimic human motion.

The term *visual attention* covers a range of phenomenon from where we point our eyes to cognitive effects involving what we notice in a complex scene and how

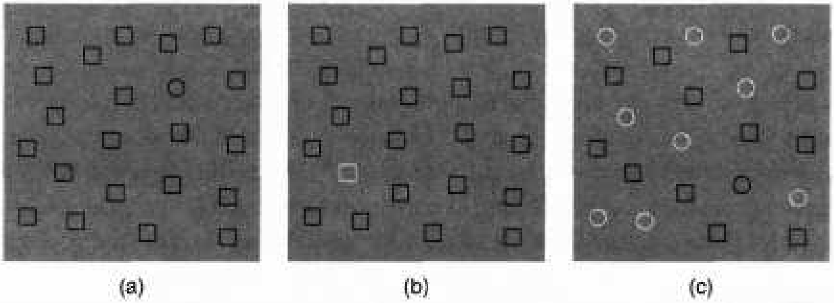


**Figure 21.43.** In (a) and (b), visual attention is quickly drawn to the item of different shape or color. In (c), sequential search appears to be necessary in order to find the one item that differs in both shape and color.

we interpret what we notice (Pashler, 1998). Figure 21.43 provides an example of how attentional processes affect vision, even for very simple images. In the left two panels, the one pattern differing in shape or color from the rest immediately "pops out" and is easily noticed. In the panel on the right, the one pattern differing in both shape and color is harder to find. The reason for this is that the visual system can do a parallel search for items distinguished by individual properties, but requires more cognitive, sequential search when looking for items that are indicated by the simultaneous presence of two distinguishing features. Graphically based human-computer interfaces should be (but often are not!) designed with an understanding of how to take advantage of visual attention processes in people so as to communicate important information quickly and effectively.

## 21.5 Picture Perception

So far, this chapter has dealt with the visual perception that occurs when the world is directly imaged by the human eye. When we view the results of computer graphics, of course, we are looking at rendered images and not the real world. This has important perceptual implications. In principle, it should be possible to generate computer graphics that appears indistinguishable from the real world, at least for monocular viewing without either object or observer motion. Imagine looking out at the world through a glass window. Now, consider coloring each point on the window to exactly match the color of the world originally seen at that point.[10] The light reaching the eye is unchanged by this operation, meaning that perception should be the same whether the painted glass is viewed or the real world is viewed through the window. The goal of computer graphics can be thought of as producing the colored window without actually having the equivalent real-world view available.

The problem for computer graphics and other visual arts is that we can't in practice match a view of the real world by coloring a flat surface. The brightness and dynamic range of light in the real world is impossible to recreate using any current display technology. Resolution of rendered images is also often less that the finest detail perceivable by human vision. Lightness and color constancy are much less apparent in pictures than in the real world, likely because the visual system attempts to compensate for variability in the brightness and color of the illumination based on the ambient illumination in the viewing environment rather than the illumination associated with the rendered image. This is why the real-

[10] This idea was first described by the painter Leon Battista Alberti in 1435 and is now known as *Alberti's Window*. It is closely related to the *camera obscura*.

istic appearance of color in photographs depends on film color balanced for the nature of the light source present when the photograph was taken and why realistic color in video requires a white-balancing step. While much is known about how limitations in resolution, brightness, and dynamic range affect the detectability of simple patterns, almost nothing is known about how these display properties affect spatial vision or object identification.

We have a better understanding of other aspects of this problem, which psychologists refer to as the perception of *pictorial space* (S. Rogers, 1995). One difference between viewing images and viewing the real world is that accommodation, binocular stereo, motion parallax, and perhaps other depth cues may indicate that the surface under view is much different that the distances in the world that it is intended to represent. The depths that are seen in such a situation tend to be somewhere between the depths indicated by the pictorial cues in the image and the distance to the image itself. When looking at a photograph or computer display, this often results in a sense of scale smaller than intended. On the other hand, seeing a movie in a big-screen theater produces a more compelling sense of spaciousness than does seeing the same movie on television, even if the distance to the TV is such that the visual angles are the same, since the movie screen is farther away.

Computer graphics rendered using perspective projection has a viewpoint, specified as a position and direction in model space, and a view frustum, which specifies the horizontal and vertical field of view and several other aspects of the viewing transform. If the rendered image is not viewed from the correct location, the visual angles to the borders of the image will not match the frustum used in creating the image. All visual angles within the image will be distorted as well, causing a distortion in all of the pictorial depth and orientation cues based on linear perspective. This effect occurs frequently in practice, when a viewer is positioned either too close or too far away from a photograph or display surface. If the viewer is too close, the perspective cues for depth will be compressed, and the cues for surface slant will indicate that the surface is closer to perpendicular to the line of sight than is actually the case. The situation is reversed if the viewer is too far from the photograph or screen. The situation is even more complicated if the line of sight does not go through the center of the viewing area, as is commonly the case in a wide variety of viewing situations.

The human visual system is able to partially compensate for perspective distortions arising from viewing an image at the wrong location, which is why we are able to sit in different seats at a movie theater and experience a similar sense of the depicted space. When controlling viewing position is particularly important, *viewing tubes* can be used. These are appropriately sized tubes, mounted in a fixed position relative to the display, and through which the viewer sees the display. The viewing tube constrains the observation point to the (hopefully) correct position. Viewing tubes are also quite effective at reducing the conflict in depth information between the pictorial cues in the image and the actual display surface. They eliminate both stereo and motion parallax, which if present would correspond to the display surface, not the rendered view. If they are small enough in diameter, they also reduce other cues to the location of the display surface by hiding the picture frame or edge of the display device. Exotic visually immersive display devices such as head-mounted displays (HMDs) go further in attempting to hide visual cues to the position of the display surface while adding binocular stereo and motion parallax consistent with the geometry of the world being rendered.

Erik Reinhard

# 22

# Tone Reproduction

As discussed in Chapter 21, the human visual system adapts to a wide range of viewing conditions. Under normal viewing, we may discern a range of around 4 to 5 log units of illumination, i.e., the ratio between brightest and darkest areas where we can see detail may be as large as $100,000 : 1$. Through adaptation processes, we may adapt to an even larger range of illumination. We call images that are matched to the capabilities of the human visual system *high dynamic range*.

Visual simulations routinely produce images with a high dynamic range (Ward Larson & Shakespeare, 1998). Recent developments in image-capturing techniques allow multiple exposures to be aligned and recombined into a single high dynamic range image (Debevec & Malik, 1997). Multiple exposure techniques are also available for video. In addition, we expect future hardware to be able to photograph or film high dynamic range scenes directly. In general, we may think of each pixel as a triplet of three floating point numbers.

As it is becoming easier to create high dynamic range imagery, the need to display such data is rapidly increasing. Unfortunately, most current display devices, monitors and printers, are only capable of displaying around 2 log units of dynamic range. We consider such devices to be of low dynamic range. Most images in existence today are represented with a byte-per-pixel-per-color channel, which is matched to current display devices, rather than to the scenes they represent.

Typically, low dynamic range images are not able to represent scenes without loss of information. A common example is an indoor room with an out-

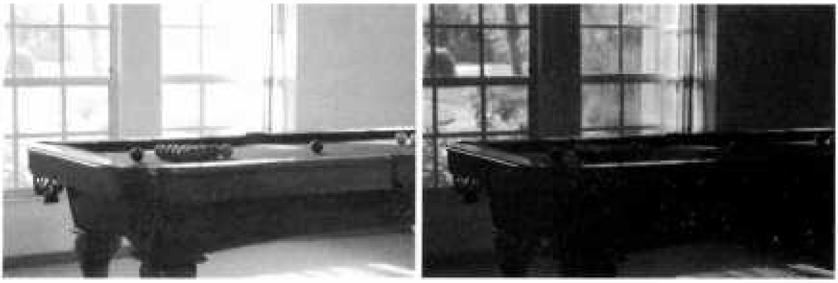


**Figure 22.1.** With conventional photography, some parts of the scene may be under- or over-exposed. To visualize the snooker table, the view through the window is burned out in the left image. On the other hand, the snooker table will be too dark if the outdoor part of this scene is properly exposed. Compare with Figure 22.2, which shows a high dynamic range image prepared for display using a tone reproduction algorithm.

door area visible through the window. Humans are easily able to see details of both the indoor part and the outside part. A conventional photograph typically does not capture this full range of information—the photographer has to choose whether the indoor or the outdoor part of the scene is properly exposed (see Figure 22.1). These decisions may be avoided by using high dynamic range imaging and preparing these images for display using techniques described in this chapter (see Figure 22.2).

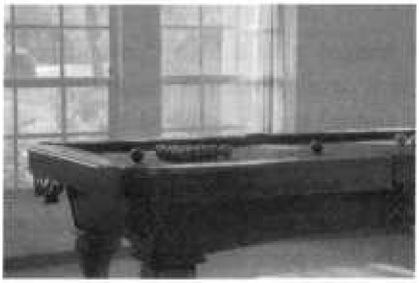


**Figure 22.2.** A high dynamic range image tonemapped for display using a recent tone reproduction operator (Reinhard & Devlin, 2005). In this image, both the indoor part and the view through the window are properly exposed.

There are two strategies available to display high dynamic range images. First, we may develop display devices which can directly accommodate a high dynamic range (Seetzen, Whitehead, & Ward, 2003; Seetzen et al., 2004). Second, we may prepare high dynamic range images for display on low dynamic range display devices (Upstill, 1985). This is currently the more common approach and the topic of this chapter. Although we foresee that high dynamic range display devices will become widely used in the (near) future, the need to compress the dynamic range of an image may diminish, but will not disappear. In particular, printed media such as this book are by their very nature low dynamic range.

Compressing the range of values of an image for the purpose of display on a low dynamic range display device is called tonemapping or tone reproduction.

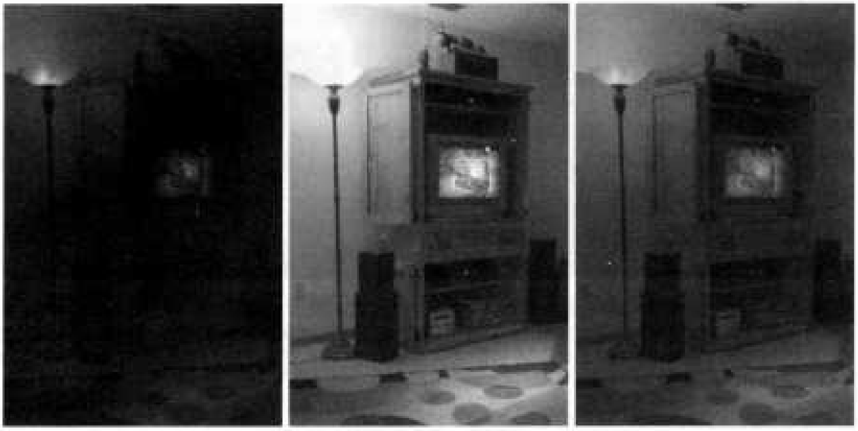


Figure 22.3. Linear scaling of high dynamic range images to fit a given display device may cause significant detail to be lost (left and middle). The left image is linearly scaled. In the middle image high values are clamped. For comparison, the right image is tonemapped, allowing details in both bright and dark regions to be visible.

A simple compression function would be to normalize an image (see Figure 22.3 (left)). This constitutes a linear scaling which tends to be sufficient only if the dynamic range of the image is only marginally higher than the dynamic range of the display device. For images with a higher dynamic range, small intensity differences will be quantized to the same display value such that visible details are lost. In Figure 22.3 (middle) all pixel values larger than a user-specified maximum are set to this maximum (i.e., they are clamped). This makes the normalization less dependent on noisy outliers, but here we lose information in the bright areas of the image. For comparison, Figure 22.3(right) is a tonemapped version showing detail in both the dark and the bright regions.

In general linear scaling will not be appropriate for tone reproduction. The key issue in tone reproduction is then to compress an image while at the same time preserving one or more attributes of the image. Different tone reproduction algorithms focus on different attributes such as contrast, visible detail, brightness or appearance.

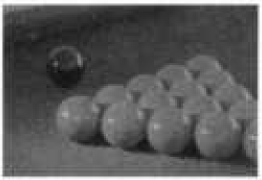


Figure 22.4. Image used for demonstrating the goal of tone reproduction in Figure 22.5.

Ideally, displaying a tonemapped image on a low dynamic range display device would create the same visual response in the observer as the original scene. Given the limitations of display devices, this will not be achievable, although we could aim for approximating this goal as closely as possible.

As an example, we created the high dynamic range image shown in Figure 22.4. This image was then tonemapped and displayed on a display device. The display device itself was then placed in the scene such that it displays its own background (Figure 22.5). In the ideal case, the display should appear transpar-

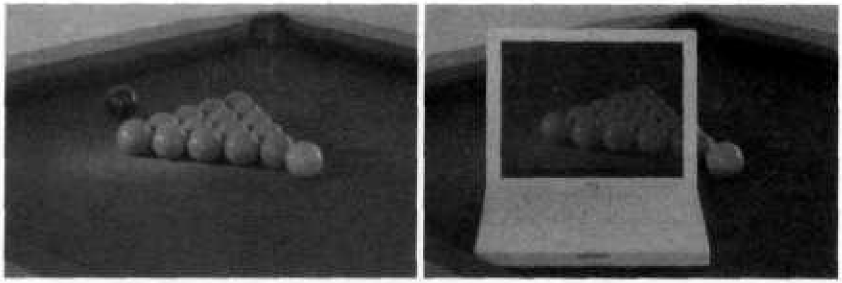


**Figure 22.5.** After tonemapping the image in Figure 22.4 and displaying it on a monitor, the monitor is placed in the scene approximately at the location where the image was taken. Dependent on the quality of the tone reproduction operator, the result should appear as if the monitor is transparent.

ent. Dependent on the quality of the tone reproduction operator, as well as the nature of the scene being depicted, this goal may be more or less achievable.

## 22.1 Classification

Although it would be possible to classify tone reproduction operators by which attribute they aim to preserve, or for which task they were developed, we classify algorithms according to their general technique. This will enable us to show the differences and similarities between a significant number of different operators, and so, hopefully, contribute to the meaningful selection of specific operators for given tone reproduction tasks.

The main classification scheme we follow hinges upon the realization that tone reproduction operators are based on insights gained from various disciplines. In particular, several operators are based on knowledge of human visual perception.

The human visual system detects light using photoreceptors located in the retina. Light is converted to an electrical signal which is partially processed in the retina and then transmitted to the brain. Except for the first few layers of cells in the retina, the signal derived from detected light is transmitted using impulse trains. The information-carrying quantity is the frequency with which these electrical pulses occur.

The range of light that the human visual system can detect is much larger than the range of frequencies employed by the human brain to transmit information. Thus, the human visual system effortlessly solves the tone reproduction problem—a large range of luminances is transformed into a small range of frequencies of impulse trains. Emulating relevant aspects of the human visual system is therefore a worthwhile approach to tone reproduction; this approach is explained in more detail in Section 22.7.

A second class of operators is grounded in physics. Light interacts with surfaces and volumes before being absorbed by the photoreceptors. In computer graphics, light interaction is generally modelled by the rendering equation. For purely diffuse surfaces, this equation may be simplified to the product between light incident upon a surface (illuminance), and this surface's ability to reflect light (reflectance) (Oppenheim, Schafer, & Stockham, 1968).

Since reflectance is a passive property of surfaces, for diffuse surfaces it is, by definition, low dynamic range—typically between 0.005 and 1 (Stockham, 1972). The reflectance of a surface cannot be larger than 1, since then it would reflect more light than was incident upon the surface. Illuminance, on the other hand, can produce arbitrarily large values and is limited only by the intensity and proximity of the light sources.

The dynamic range of an image is thus predominantly governed by the illuminance component. In the face of diffuse scenes, a viable approach to tone reproduction may therefore be to separate reflectance from illuminance, compress the illuminance component, and then recombine the image.

However, the assumption that all surfaces in a scene are diffuse is generally incorrect. Many high dynamic range images depict highlights and/or directly visible light sources (Figure 22.3). The luminance reflected by a specular surface may be almost as high as the light source it reflects.

Various tone reproduction operators currently used split the image into a high dynamic range base layer and a low dynamic range detail layer. These layers would represent illuminance and reflectance if the depicted scene were entirely diffuse. For scenes containing directly visible light sources or specular highlights, separation into base and detail layers still allows the design of effective tone reproduction operators, although no direct meaning can be attached to the separate layers. Such operators are discussed in Section 22.5.

## 22.2 Dynamic Range

Conventional images are stored with one byte per pixel for each of the red, green and blue components. The dynamic range afforded by such an encoding depends on the ratio between smallest and largest representable value, as well as the step size between successive values. Thus, for low dynamic range images, there are only 256 different values per color channel.

High dynamic range images encode a significantly larger set of possible values; the maximum representable value may be much larger and the step size between successive values may be much smaller. The file size of high dynamic range images is therefore generally larger as well, although at least one standard (the OpenEXR high dynamic range file format (Kainz, Bogart, & Hess, 2003)) includes a very capable compression scheme.

A different approach to limit file sizes is to apply a tone reproduction operator to the high dynamic data. The result may then be encoded in JPEG format. In addition, the input image may be divided pixel-wise by the tonemapped image. The result of this division can then be subsampled and stored as a small amount of data in the header of the same JPEG image (G. Ward & Simmons, 2004). The file size of such sub-band encoded images is of the same order as conventional JPEG encoded images. Display programs can display the JPEG image directly or may reconstruct the high dynamic range image by multiplying the tonemapped image with the data stored in the header.

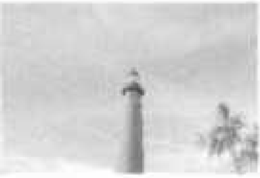


**Figure 22.6.** Dynamic range of 2.65 $\log_2$ units.

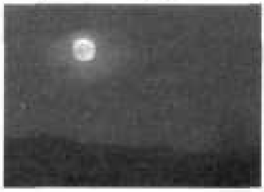


**Figure 22.7.** Dynamic range of 3.96 $\log_2$ units.

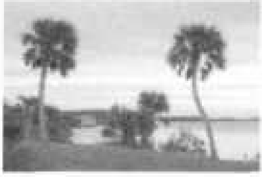


**Figure 22.8.** Dynamic range of 4.22 $\log_2$ units.

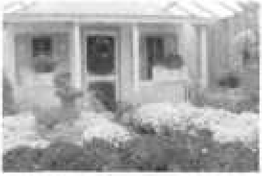


**Figure 22.9.** Dynamic range of 5.01 $\log_2$ units.

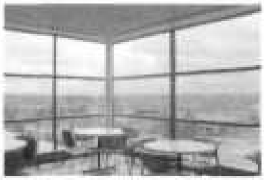


**Figure 22.10.** Dynamic range of 6.56 $\log_2$ units.

In general, the combination of smallest step size and ratio of the smallest and largest representable values determines the dynamic range that an image encoding scheme affords. For computer-generated imagery, an image is typically stored as a triplet of floating point values before it is written to file or displayed on screen, although more efficient encoding schemes are possible (Reinhard, Ward, Debevec, & Pattanaik, 2005). Since most display devices are still fitted with eight-bit D/A converters, we may think of tone reproduction as the mapping of floating point numbers to bytes such that the result is displayable on a low dynamic range display device.

The dynamic range of individual images is generally smaller, and is determined by the smallest and largest luminances found in the scene. A simplistic approach to measure the dynamic range of an image may therefore compute the ratio between the largest and smallest pixel value of an image. Sensitivity to outliers may be reduced by ignoring a small percentage of the darkest and brightest pixels.

Alternatively, the same ratio may be expressed as a difference in the logarithmic domain. This measure is less sensitive to outliers. The images shown in the margin on this page are examples of images with different dynamic ranges. Note that the night scene in this case does not have a smaller dynamic range than the day scene. While all the values in the night scene are smaller, the ratio between largest and smallest values is not.

However, the recording device or rendering algorithm may introduce noise which will lower the useful dynamic range. Thus, a measurement of the dynamic range of an image should factor in noise. A better measure of dynamic range would therefore be a signal-to-noise ratio, expressed in decibels, as used in signal processing.

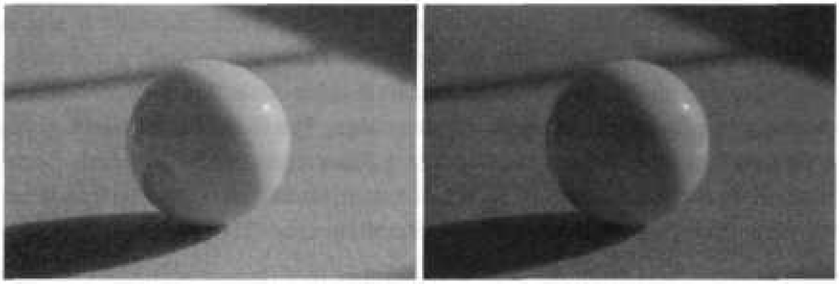


**Figure 22.11.** Per-channel gamma correction may desaturate the image. The left image was desaturated with a value of $s = 0.5$. The right image was not desaturated ($s = 1$). (See also Plate X)

## 22.3 Color

Tone reproduction operators normally compress luminance values, rather than work directly on the red, green, and blue components of a color image. After these luminance values have been compressed into display values $L_d(x, y)$, a color image may be reconstructed by keeping the ratios between color channels the same as they were before compression (using $s = 1$) (Schlick, 1994b):

$$\begin{aligned} I_{r,d}(x,y) &= \left(\frac{I_r(x,y)}{L_v(x,y)}\right)^s L_d(x,y), \\ I_{g,d}(x,y) &= \left(\frac{I_g(x,y)}{L_v(x,y)}\right)^s L_d(x,y), \\ I_{b,d}(x,y) &= \left(\frac{I_b(x,y)}{L_v(x,y)}\right)^s L_d(x,y). \end{aligned}$$

The results frequently appear over-saturated, because human color perception is non-linear with respect to overall luminance level. This means that if we view an image of a bright outdoor scene on a monitor in a dim environment, our eyes are adapted to the dim environment rather than the outdoor lighting. By keeping color ratios constant, we do not take this effect into account.

Alternatively, the saturation constant $s$ may be chosen smaller than one. Such per-channel gamma correction may desaturate the results to an appropriate level, as shown in Figure 22.11 and Color Plate X (Fattal, Lischinski, & Werman, 2002). A more comprehensive solution is to incorporate ideas from the field of color appearance modeling into tone reproduction operators (Pattanaik, Ferwerda, Fairchild, & Greenberg, 1998; Fairchild & Johnson, 2004; Reinhard & Devlin, 2005).

Finally, if an example image with a representative color scheme is already available, this color scheme may be applied to a new image.Such a mapping of colors between images may be used for subtle color correction such as saturation adjustment or for more creative color mappings. The mapping proceeds by converting both source and target images to a decorrelated color space. In such a color space, the pixel values in each color channel may be treated independently without introducing too many artifacts (Reinhard et al., 2001).

Mapping colors from one image to another in a decorrelated color space is then straightforward: compute the mean and standard deviation of all pixels in the source and target images for the three color channels separately. Then, shift and scale the target image so that in each color channel the mean and standard deviation of the target image is the same as the source image. The resulting image is then obtained by converting from the decorrelated color space to RGB and clamping negative pixels to zero. The dynamic range of the image may have changed as a result of applying this algorithm. It is therefore recommended to apply this algorithm on high dynamic range images and apply a conventional tone reproduction algorithm afterwards. A suitable decorrelated color space is the opponent space from Section 20.8.

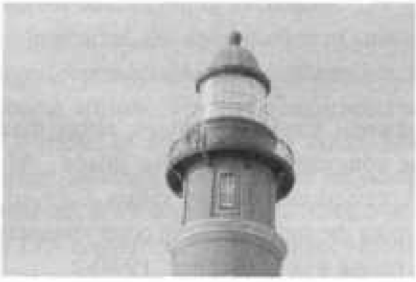


Figure 22.12. Image used for demonstrating the color transfer technique. Results are shown in Figures 22.13 and 22.31. (See also Color Plates XI, XII and XIV.)

The result of applying such a color transform to the image in Figure 22.12 is shown in Figure 22.13.

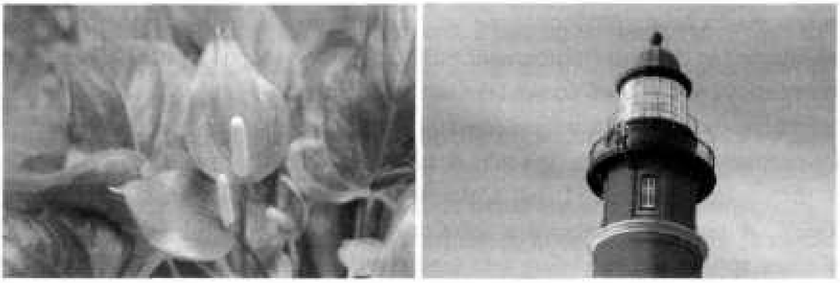


Figure 22.13. The image on the left is used to adjust the colors of the image shown in Figure 22.12. The result is shown on the right. (See also Color Plate XII.)

## 22.4 Image Formation

For now we assume that an image is formed as the result of light being diffusely reflected off of surfaces. Later in this chapter we relax this constraint to scenes directly depicting light sources and highlights. The luminance $L_v$ of each pixel is then approximated by the following product:

$$L_v(x, y) = r(x, y)\, E_v(x, y).$$

Here, $r$ denotes the reflectance of a surface, and $E_v$ denotes the illuminance. The subscript $v$ indicates that we are using photometrically weighted quantities. Alternatively, we may write this expression in the logarithmic domain (Oppenheim et al., 1968):

$$\begin{aligned} D(x, y) &= \log(L_v(x, y)) \\ &= \log(r(x, y)\, E_v(x, y)) \\ &= \log(r(x, y)) + \log(E_v(x, y)). \end{aligned}$$

Photographic transparencies record images by varying the density of the material. In traditional photography, this variation has a logarithmic relation with luminance. Thus, in analogy with common practice in photography, we will use the term *density representation* ($D$) for log luminance. When represented in the log domain, reflectance and illuminance become additive. This facilitates separation of these two components, despite the fact that isolating either reflectance or illuminance is an under-constrained problem. In practice, separation is possible only to a certain degree and depends on the composition of the image. Nonetheless, tone reproduction could be based on disentangling these two components of image formation, as shown in the following two sections.

## 22.5 Frequency-Based Operators

For typical diffuse scenes, the reflectance component tends to exhibit high spatial frequencies due to textured surfaces as well as the presence of surface edges. On the other hand, illuminance tends to be a slowly varying function over space.

Since reflectance is low dynamic range and illuminance is high dynamic range, we may try to separate the two components. The frequency-dependence of both reflectance and illuminance provides a solution. We may for instance compute the Fourier transform of an image and attenuate only the low frequencies. This compresses the illuminance component while leaving the reflectance component

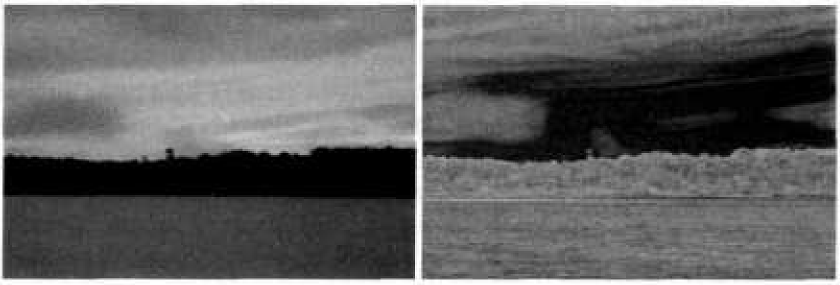


Figure 22.14. Bilateral filtering removes small details but preserves sharp gradients (left). The associated detail layer is shown on the right.

largely unaffected—the very first digital tone reproduction operator known to us takes this approach (Oppenheim et al., 1968).

More recently, other operators have also followed this line of reasoning. In particular, bilateral and trilateral filters were used to separate an image into base and detail layers (Durand & Dorsey, 2002; Choudhury & Tumblin, 2003). Both filters are edge-preserving smoothing operators which may be used in a variety of different ways. Applying an edge-preserving smoothing operator to a density image results in a blurred image in which sharp edges remain present (Figure 22.14 (left)). We may view such an image as a base layer. If we then pixel-wise divide the high dynamic range image by the base layer, we obtain a detail layer which contains all the high frequency detail (Figure 22.14 (right)).

For diffuse scenes, base and detail layers are similar to representations of illuminance and reflectance. For images depicting highlights and light sources, this parallel does not hold. However, separation of an image into base and detail layers is possible regardless of the image's content. By compressing the base layer before recombining into a compressed density image, a low dynamic range density image may be created (Figure 22.15). After exponentiation, a displayable image is obtained.

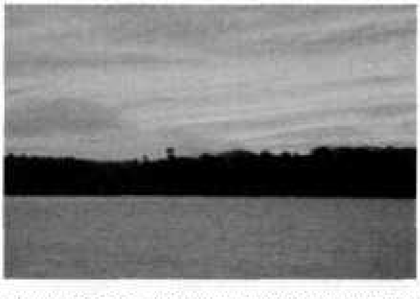


Figure 22.15. An image tonemapped using bilateral filtering. The base and detail layers shown in Figure 22.14 are recombined after compressing the base layer.

Edge-preserving smoothing operators may also be used to compute a local adaptation level for each pixel, which may be used in a spatially varying or local tone reproduction operator. We describe this use of bilateral and trilateral filters in Section 22.7.

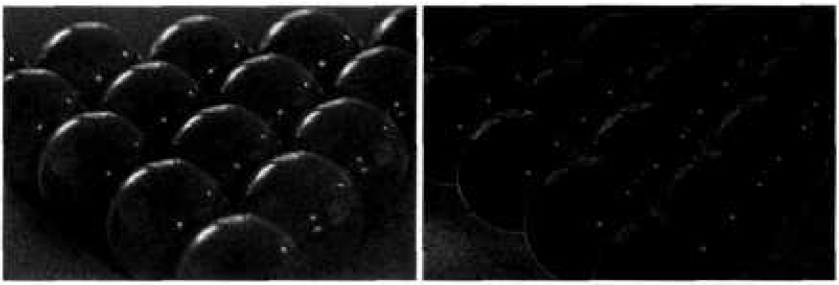


**Figure 22.16.** The image on the left (tonemapped using gradient domain compression) shows a scene with highlights. These highlights show up as large gradients on the right, where the magnitude of the gradients is mapped to a grayscale (black is a gradient of 0, white is the maximum gradient in the image).

## 22.6 Gradient-Domain Operators

The arguments made for the frequency-based operators in the preceding section also hold for the gradient field. Assuming that no light sources are directly visible, the reflectance component will be a constant function with sharp spikes in the gradient field. Similarly, the illuminance component will cause small gradients everywhere.

Humans are generally able to separate illuminance from reflectance in typical scenes. The perception of surface reflectance after discounting the illuminant is called *lightness.* To assess the lightness of an image depicting only diffuse surfaces, B. K. P. Horn was the first to separate reflectance and illuminance using a gradient field (Horn, 1974). He used simple thresholding to remove all small gradients and then integrated the image, which involves solving a Poisson equation using the Full Multigrid Method (Press, Teukolsky, Vetterling, & Flannery, 1992).

The result is similar to an edge-preserving smoothing filter. This is according to expectation since Oppenheim's frequency-based operator works under the same assumptions of scene reflectivity and image formation. In particular, Horn's work was directly aimed at "mini-worlds of Mondrians," which are simplified versions of diffuse scenes which resemble the abstract paintings by the famous Dutch painter Piet Mondrian.

Horn's work cannot be employed directly as a tone reproduction operator, since most high dynamic range images depict light sources. However, a relatively small variation will turn this work into a suitable tone reproduction operator. If light sources or specular surfaces are depicted in the image, then large gradients will be associated with the edges of light sources and highlights. These cause the image to have a high dynamic range. An example is shown in Figure 22.16, where the highlights on the snooker balls cause sharp gradients.

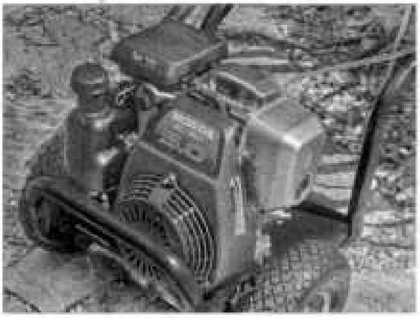


**Figure 22.17.** An image tonemapped using gradient domain compression.

We could therefore compress a high dynamic range image by attenuating large gradients, rather than thresholding the gradient field. This approach was taken by Fattal et al. who showed that high dynamic range imagery may be successfully compressed by integrating a compressed gradient field (Figure 22.17) (Fattal et al., 2002). Fattal's gradient domain compression is not limited to diffuse scenes.

## 22.7 Spatial Operators

In the following sections, we discuss tone reproduction operators which apply compression directly on pixels without transformation to other domains. Often global and local operators are distinguished. Tone reproduction operators in the former class change each pixel's luminance values according to a compressive function which is the same for each pixel. The term global stems from the fact that many such functions need to be anchored to some values determined by analyzing the full image. In practice, most operators use the geometric average $\bar{L}_v$ to steer the compression:

$$\bar{L}_v = \exp\left(\frac{1}{N}\sum_{x,y}\log(\delta + L_v(x,y))\right). \tag{22.1}$$

In Equation 22.1, a small constant $\delta$ is introduced to prevent the average to become zero in the presence of black pixels. The geometric average is normally mapped to a predefined display value. The effect of mapping the geometric average to different display values is shown in Figure 22.18. Alternatively, sometimes the minimum or maximum image luminance is used. The main challenge faced in the design of a global operator lies in the choice of the compressive function.

On the other hand, local operators compress each pixel according to a specific compression function which is modulated by information derived from a selection of neighboring pixels, rather than the full image. The rationale is that a bright pixel in a bright neighborhood may be perceived differently than a bright pixel in a dim neighborhood. Design challenges in the development of a local operator involves choosing the compressive function, the size of the local neighborhood

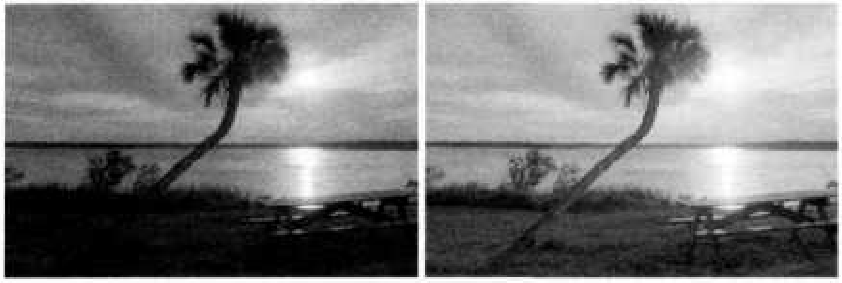


**Figure 22.18.** Spatial tonemapping operator applied after mapping the geometric average to different display values (left: 0.12, right: 0.38).

for each pixel, and the manner in which local pixel values are used. In general, local operators achieve better compression than global operators (Figure 22.19), albeit at a higher computational cost.

Both global and local operators are often inspired by the human visual system. Most operators employ one of two distinct compressive functions, which is orthogonal to the distinction between local and global operators. Display values $L_d(x, y)$ are most commonly derived from image luminances $L_v(x, y)$ by the

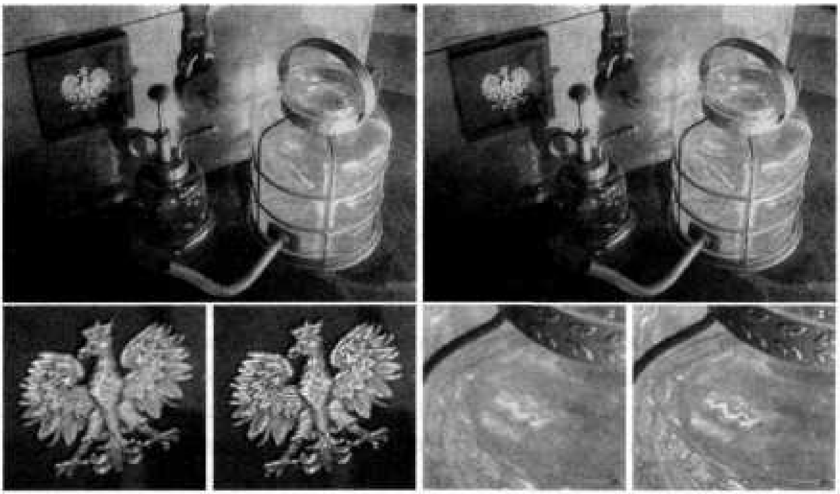


**Figure 22.19.** A global tone reproduction operator (left) and a local tone reproduction operator (right) (Reinhard, Stark, Shirley, & Ferwerda, 2002) of each image. The local operator shows more detail; for example the metal badge on the right shows better contrast and the highlights are crisper.

following two functional forms:

$$L_d(x,y) = \frac{L_v(x,y)}{f(x,y)}, \tag{22.2}$$

$$L_d(x,y) = \frac{L_v(x,y)}{L_v(x,y) + f^n(x,y)}. \tag{22.3}$$

In these equations, $f(x, y)$ may either be a constant or a function which varies per pixel. In the former case, we have a global operator, whereas a spatially varying function $f(x, y)$ results in a local operator. The exponent $n$ is usually a constant which is fixed for a particular operator.

Equation 22.2 divides each pixel's luminance by a value derived from either the full image or a local neighborhood. Equation 22.3 has an S-shaped curve on a log-linear plot and is called a sigmoid for that reason. This functional form fits data obtained from measuring the electrical response of photoreceptors to flashes of light in various species. In the following sections, we discuss both functional forms.

## 22.8 Division

Each pixel may be divided by a constant to bring the high dynamic range image within a displayable range. Such a division essentially constitutes linear scaling, as shown in Figure 22.3. While Figure 22.3 shows ad-hoc linear scaling, this approach may be refined by employing psychophysical data to derive the scaling constant $f(x, y) = k$ in Equation 22.2 (G. J. Ward, 1994; Ferwerda, Pattanaik, Shirley, & Greenberg, 1996).

Alternatively, several approaches exist that compute a spatially varying divisor. In each of these cases, $f(x, y)$ is a blurred version of the image, i.e., $f(x, y) = L_v^{\text{blur}}(x, y)$. The blur is achieved by convolving the image with a Gaussian filter (Chiu et al., 1993; Rahman, Jobson, & Woodell, 1996). In addition, the computation of $f(x, y)$ by blurring the image may be combined with a shift in white point for the purpose of color appearance modeling (Fairchild & Johnson, 2002; Johnson & Fairchild, 2003; Fairchild & Johnson, 2004).

The size and the weight of the Gaussian filter has a profound impact on the resulting displayable image. The Gaussian filter has the effect of selecting a weighted local average. Tone reproduction is then a matter of dividing each pixel by its associated weighted local average. If the size of the filter kernel is chosen too small, then haloing artifacts will occur (Figure 22.20 (left)). Haloing is a common problem with local operators and is particularly evident when tone mapping relies on division.

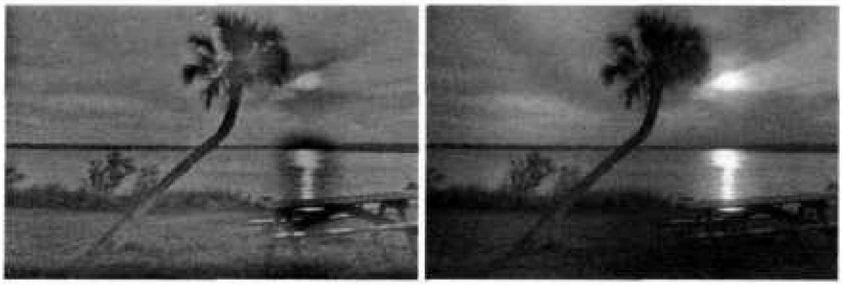


**Figure 22.20.** Images tonemapped by dividing by Gaussian blurred versions. The size of the filter kernel is 64 pixels for the left image and 512 pixels for the right image. For division-based algorithms, halo artifacts are minimized by choosing large filter kernels.

In general, haloing artifacts may be minimized in this approach by making the filter kernel large (Figure 22.20 (right)). Reasonable results may be obtained by choosing a filter size of at least one quarter of the image. Sometimes even larger filter kernels are desirable to minimize artifacts. Note, that in the limit, the filter size becomes as large as the image itself. In that case the local operator becomes global, and the extra compression normally afforded by a local approach is lost.

The functional form whereby each pixel is divided by a Gaussian blurred pixel at the same spatial position thus requires an undesirable tradeoff between amount of compression and severity of artifacts.

## 22.9 Sigmoids

Equation 22.3 follows a different functional form from simple division, and, therefore, affords a different tradeoff between amount of compression, presence of artifacts, and speed of computation.

Sigmoids have several desirable properties. For very small luminance values, the mapping is approximately linear, so that contrast is preserved in dark areas of the image. The function has an asymptote at one, which means that the output mapping is always bounded between 0 and 1.

In Equation 22.3, the function $f(x, y)$ may be computed as a global constant or as a spatially varying function. Following common practice in electrophysiology, we call $f(x, y)$ the *semi-saturation* constant. Its value determines which values in the input image are optimally visible after tonemapping. In particular, if we assume that the exponent $n$ equals 1, then luminance values equal to the semi-saturation constant will be mapped to 0.5. The effect of choosing different semi-saturation constants is shown in Figure 22.21.

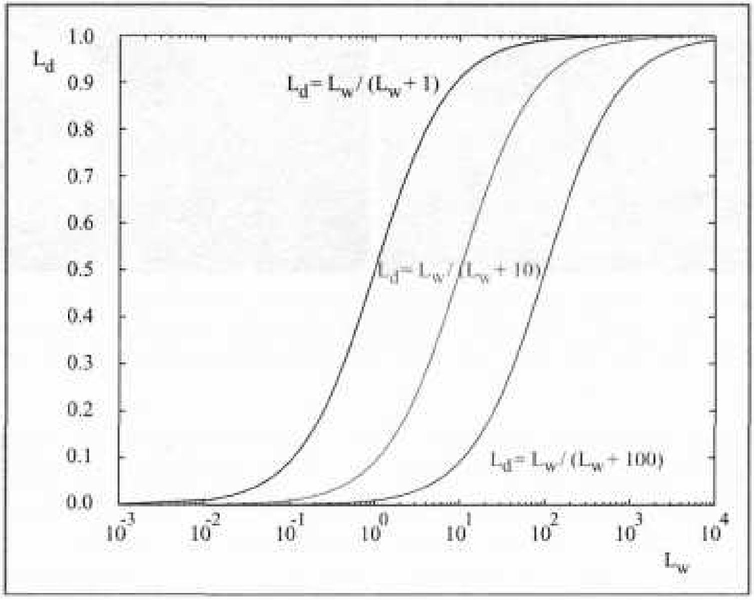


**Figure 22.21.** The choice of semi-saturation constant determines how input values are mapped to display values.

The function $f(x, y)$ may be computed in several different ways (Reinhard et al., 2005). In its simplest form, $f(x, y)$ is set to $\bar{L}_v/k$, so that the geometric average is mapped to user parameter $k$ (Figure 22.22) (Reinhard et al., 2002). In this case, a good initial value for $k$ is 0.18, although for particularly bright or dark scenes this value may be raised or lowered. Its value may be estimated from the image itself (Reinhard, 2003). The exponent $n$ in Equation 22.3 may be set to 1.

In this approach, the semi-saturation constant is a function of the geometric average, and the operator is therefore global. A variation of this global opera-

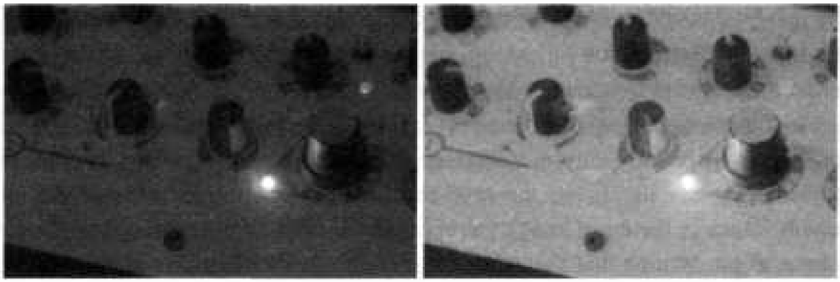


**Figure 22.22.** A linearly scaled image (left) and an image tonemapped using sigmoidal compression (right).

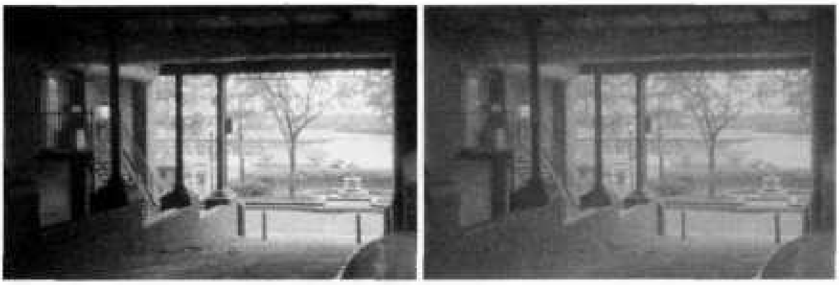


**Figure 22.23.** Linear interpolation varies contrast in the tonemapped image. The parameter *a* is set to 0.0 in the left image, and to 1.0 in the right image.

tor computes the semi-saturation constant by linearly interpolating between the geometric average and each pixel's luminance:

$$f(x,y) = a\, L_v(x,y) + (1-a)\, \bar{L}_v.$$

The interpolation is governed by user parameter $a$ which has the effect of varying the amount of contrast in the displayable image (Figure 22.23) (Reinhard & Devlin, 2005). More contrast means less visible detail in the light and dark areas and vice versa. This interpolation may be viewed as a half-way house between a fully global and a fully local operator by interpolating between the two extremes without resorting to expensive blurring operations.

Although operators typically compress luminance values, this particular operator may be extended to include a simple form of chromatic adaptation. It thus presents an opportunity to adjust the level of saturation normally associated with tonemapping, as discussed at the beginning of this chapter.

Rather than compress the luminance channel only, sigmoidal compression is applied to each of the three color channels:

$$\begin{aligned} I_{r,d}(x,y) &= \frac{I_r(x,y)}{I_r(x,y) + f^n(x,y)}, \\ I_{g,d}(x,y) &= \frac{I_g(x,y)}{I_g(x,y) + f^n(x,y)}, \\ I_{b,d}(x,y) &= \frac{I_b(x,y)}{I_b(x,y) + f^n(x,y)}. \end{aligned}$$

The computation of $f(x,y)$ is also modified to bilinearly interpolate between the geometric average luminance and pixel luminance and between each independent color channel and the pixel's luminance value. We therefore compute the geometric average luminance value $\bar{L}_v$, as well as the geometric average of the red, green and blue channels ($\bar{I}_r$, $\bar{I}_g$, and $\bar{I}_b$). From these values, we compute $f(x,y)$

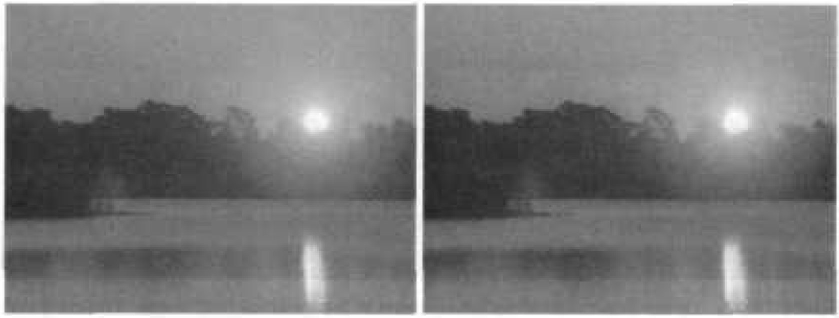


**Figure 22.24.** Linear interpolation for color correction. The parameter $c$ is set to 0.0 in the left image, and to 1.0 in the right image. (See also Color Plate XIII.)

for each pixel and for each color channel independently. We show the equation for the red channel ($f_r(x,y)$):

$$\begin{aligned} G_r(x,y) &= c\, I_r(x,y) + (1-c)\, L_v(x,y), \\ \bar{G}_r(x,y) &= c\, \bar{I}_r + (1-c)\, \bar{L}_v, \\ f_r(x,y) &= a\, G_r(x,y) + (1-a)\, \bar{G}_r(x,y). \end{aligned}$$

The interpolation parameter $a$ steers the amount of contrast as before, and the new interpolation parameter $c$ allows a simple form of color correction (Figure 22.24 and Color Plate XIII).

So far we have not discussed the value of the exponent $n$ in Equation 22.3. Studies in electrophysiology report values between $n = 0.2$ and $n = 0.9$ (Hood, Finkelstein, & Buckingham, 1979). While the exponent may be user-specified, for a wide variety of images we may estimate a reasonable value from the geometric average luminance $\bar{L}_v$ and the minimum and maximum luminance in the image ($L_{\min}$ and $L_{\max}$) with the following empirical equation:

$$n = 0.3 + 0.7 \left( \frac{L_{\max} - \bar{L}_v}{L_{\max} - L_{\min}} \right)^{1.4}.$$

The several variants of sigmoidal compression shown so far are all global in nature. This has the advantage that they are fast to compute, and they are very suitable for medium to high dynamic range images. For very high dynamic range images, it may be necessary to resort to a local operator, since this may give some extra compression. A straightforward method to extend sigmoidal compression replaces the global semi-saturation constant by a spatially varying function, which may be computed in several different ways.

In other words, the function $f(x,y)$ is so far assumed to be constant, but may also be computed as a spatially localized average. Perhaps the simplest way to accomplish this is to once more use a Gaussian blurred image. Each pixel in a blurred image represents a locally averaged value which may be viewed as a suitable choice for the semi-saturation constant[1].

As with division-based operators discussed in the previous section, we have to consider haloing artifacts. However, when an image is divided by a Gaussian blurred version of itself, the size of the Gaussian filter kernel needs to be large in order to minimize halos. If sigmoids are used with a spatially variant semi-saturation constant, the Gaussian filter kernel needs to be made small in order to minimize artifacts. This is a significant improvement, since small amounts of Gaussian blur may be efficiently computed directly in the spatial domain. In other words, there is no need to resort to expensive Fourier transforms. In practice, filter kernels of only a few pixels width are sufficient to suppress significant artifacts while at the same time producing more local contrast in the tonemapped images.

One potential issue with Gaussian blur is that the filter blurs across sharp contrast edges in the same way that it blurs small details. In practice, if there is a large contrast gradient in the neighborhood of the pixel under consideration, this causes the Gaussian-blurred pixel to be significantly different from the pixel itself. This is the direct cause for halos. By using a very large filter kernel in a division-based approach, such large contrasts are averaged out.

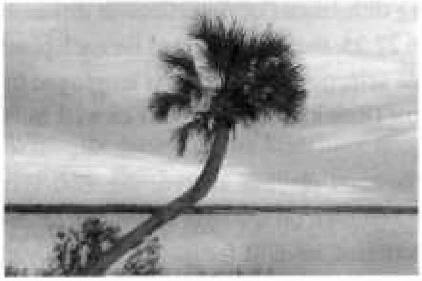


**Figure 22.25.** Example image used to demonstrate the scale selection mechanism shown in Figure 22.26.

In sigmoidal compression schemes, a small Gaussian filter minimizes the chances of overlapping with a sharp contrast gradient. In that case, halos still occur, but their size is such that they usually go unnoticed and instead are perceived as enhancing contrast.

Another way to blur an image, while minimizing the negative effects of nearby large contrast steps, is to avoid blurring over such edges. A simple, but computationally expensive way, is to compute a stack of Gaussian-blurred images with different kernel sizes. For each pixel, we may choose the largest Gaussian that does not overlap with a significant gradient.

In a relatively uniform neighborhood, the value of a Gaussian-blurred pixel should be the same regardless of the filter kernel size. Thus, the difference between a pixel filtered with two different Gaussians should be approximately zero.

[1] Although $f(x, y)$ is now no longer a constant, we continue to refer to it as the semi-saturation constant.

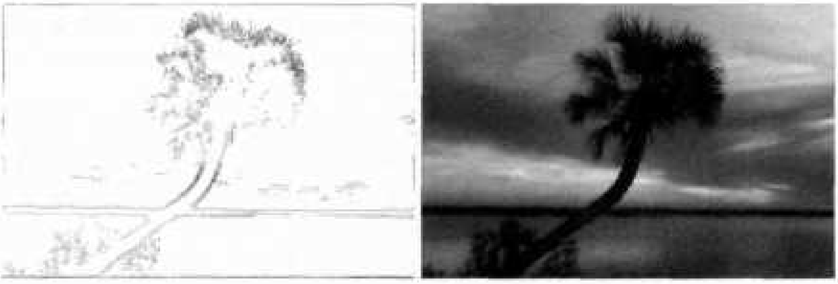


**Figure 22.26.** Scale selection mechanism: the left image shows the scale selected for each pixel of the image shown in Figure 22.25; the darker the pixel, the smaller the scale. A total of eight different scales were used to compute this image. The right image shows the local average computed for each pixel on the basis of the neighborhood selection mechanism.

This difference will only change significantly if the wider filter kernel overlaps with a neighborhood containing a sharp contrast step, whereas the smaller filter kernel does not.

It is possible, therefore, to find the largest neighborhood around a pixel that does not contain sharp edges by examining differences of Gaussians at different kernel sizes. For the image shown in Figure 22.25, the scale selected for each pixel is shown in Figure 22.26 (left). Such a scale selection mechanism is employed by the photographic tone reproduction operator (Reinhard et al., 2002) as well as in Ashikhmin's operator (Ashikhmin, 2002).

Once the appropriate neighborhood for each pixel is known, the Gaussian blurred average $L_{\mathrm{blur}}$ for this neighborhood (shown on the right of Figure 22.26) may be used to steer the semi-saturation constant, such as for instance employed by the photographic tone reproduction operator:

$$L_{\mathrm{d}} = \frac{L_{\mathrm{w}}}{1. + L_{\mathrm{blur}}}.$$

An alternative, and arguably better, approach is to employ edge-preserving smoothing operators, which are designed specifically for removing small details while keeping sharp contrasts in tact. Several such filters, such as the bilateral filter (Figure 22.27), trilateral filter, Susan filter, the LCIS algorithm and the mean shift algorithm are suitable, although some of them are expensive to compute (Durand & Dorsey, 2002; Choudhury & Tumblin, 2003; Pattanaik & Yee, 2002; Tumblin & Turk, 1999; Comaniciu & Meer, 2002).

## 22.10 Other Approaches

Although the previous sections together discuss most tone reproduction operators to date, there are one or two operators that do not directly fit into the above cate-

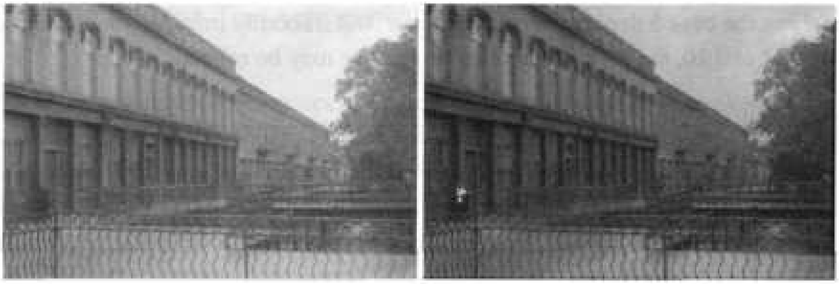


**Figure 22.27.** Sigmoidal compression (left) and sigmoidal compression using bilateral filtering to compute the semi-saturation constant (right). Note the improved contrast in the sky in the right image.

gories. The simplest of these are variations of logarithmic compression, and the other is a histogram-based approach.

Dynamic range reduction may be accomplished by taking the logarithm, provided that this number is greater than 1. Any positive number may then be nonlinearly scaled between 0 and 1 using the following equation:

$$L_d(x,y) = \frac{\log_b(1 + L_v(x,y))}{\log_b(1 + L_{\max})}$$

While the base $b$ of the logarithm above is not specified, any choice of base will do. This freedom to choose the base of the logarithm may be used to vary the base with input luminance, and thus achieve an operator that is better matched to the image being compressed (Drago, Myszkowski, Annen, & Chiba, 2003). This method uses Perlin and Hoffert's bias function which takes user parameter $p$ (Perlin & Hoffert, 1989):

$$\text{bias}_p(x) = x^{\log_{10}(p)/\log_{10}(1/2)}.$$

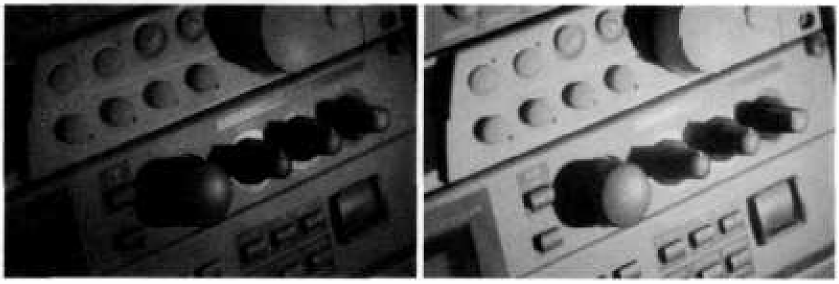


**Figure 22.28.** Logarithmic compression using base 10 logarithms (left) and logarithmic compression with varying base (right).

Making the base $b$ dependent on luminance and smoothly interpolating bases between 2 and 10, the logarithmic mapping above may be refined:

$$L_d(x,y) = \frac{\log_{10}(1 + L_v(x,y))}{\log_{10}(1 + L_{\max})} \cdot \frac{1}{\log_{10}\left(2 + 8\left(\left(\frac{L_v(x,y)}{L_{\max}}\right)^{\log_{10}(p)/\log_{10}(1/2)}\right)\right)}.$$

For user parameter $p$, an initial value of around 0.85 tends to yield plausible results (Figure 22.28 (right)).

Alternatively, tone reproduction may be based on histogram equalization. Traditional histogram equalization aims to give each luminance value equal probability of occurrence in the output image. Greg Ward refines this method in a manner that preserves contrast (Ward Larson, Rushmeier, & Piatko, 1997).

First, a histogram is computed from the luminances in the high dynamic range image. From this histogram, a cumulative histogram is computed such that each bin contains the number of pixels that have a luminance value less than or equal to the luminance value that the bin represents. The cumulative histogram is a monotonically increasing function. Plotting the values in each bin against the luminance values represented by each bin therefore yields a function which may be viewed as a luminance mapping function. Scaling this function, such that the vertical axis spans the range of the display device, yields a tone reproduction operator. This technique is called histogram equalization.

Ward further refined this method by ensuring that the gradient of this function never exceeds 1. This means, that if the difference between neighboring values in the cumulative histogram is too large, this difference is clamped to 1. This avoids the problem that small changes in luminance in the input may yield large differences in the output image. In other words, by limiting the gradient of the cumulative histogram to 1, contrast is never exaggerated. The resulting algorithm is called histogram adjustment (see Figure 22.29).

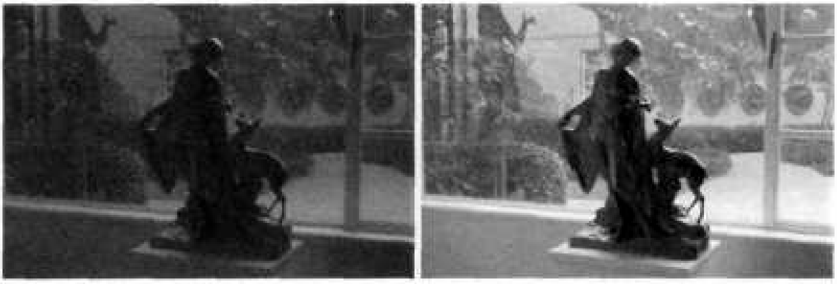


**Figure 22.29.** A linearly scaled image (left) and a histogram adjusted image (right). *Image created with the kind permission of the Albin Polasek museum, Winter Park, Florida.*

## 22.11 Night Tonemapping

The tone reproduction operators discussed so far nearly all assume that the image represents a scene under *photopic* viewing conditions, i.e., as seen at normal light levels. For *scotopic* scenes, i.e., very dark scenes, the human visual system exhibits distinctly different behavior. In particular, perceived contrast is lower, visual acuity (i.e., the smallest detail that we can distinguish) is lower, and everything has a slightly blue appearance.

To allow such images to be viewed correctly on monitors placed in photopic lighting conditions, we may preprocess the image such that it appears as if we were adapted to a very dark viewing environment. Such preprocessing frequently takes the form of a reduction in brightness and contrast, desaturation of the image, blue shift, and a reduction in visual acuity (Thompson, Shirley, & Ferwerda, 2002).

A typical approach starts by converting the image from RGB to XYZ. Then, scotopic luminance $V$ may be computed for each pixel:

$$V = Y \left[ 1.33 \left( 1 + \frac{Y + Z}{X} \right) - 1.68 \right].$$

This single channel image may then be scaled and multiplied by an empirically chosen bluish grey. An example is shown in Figure 22.30. If some pixels are in the photopic range, then the night image may be created by linearly blending the bluish grey image with the input image. The fraction to use for each pixel depends on $V$.

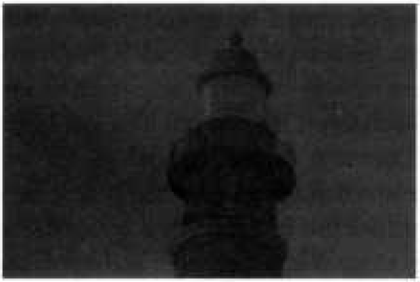


**Figure 22.30.** Simulated night scene using the image shown in Figure 22.12. (See also Color Plates XI and XV.)

Loss of visual acuity may be modelled by low-pass filtering the night image, although this would give an incorrect sense of blurriness. A better approach is to apply a bilateral filter to retain sharp edges while blurring smaller details (Tomasi & Manduchi, 1998).

Finally, the color transfer technique outlined in Section 22.3 may also be used to transform a day-lit image into a night scene. The effectiveness of this approach depends on the availability of a suitable night image from which to transfer colors. As an example, the image in Figure 22.12 is transformed into a night image in Figure 22.31.

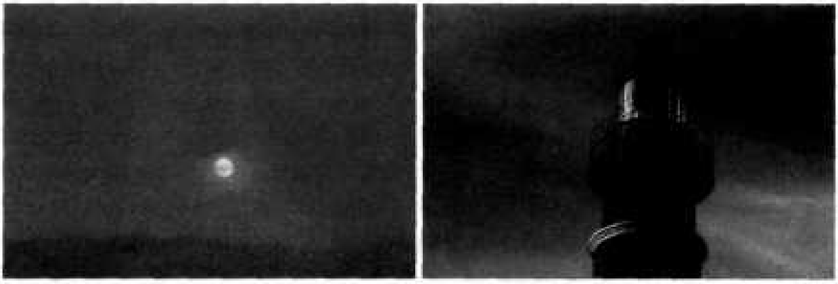


**Figure 22.31.** The image on the left is used to transform the image of Figure 22.12 into a *night scene, shown here on the right.* (*See also Color Plate XIV.*)

## 22.12 Discussion

Since global illumination algorithms naturally produce high dynamic range images, direct display of the resulting images is not possible. Rather than resort to linear scaling or clamping, a tone reproduction operator should be used. Any tone reproduction operator is better than using no tone reproduction. Dependent on the requirements of the application, one of several operators may be suitable.

For instance, real-time rendering applications should probably resort to a simple sigmoidal compression, since these are fast enough to also run in real time. In addition, their visual quality is often good enough. The histogram adjustment technique (Ward Larson et al., 1997) may also be fast enough for real-time operation.

For scenes containing a very high dynamic range, better compression may be achieved with a local operator. However, the computational cost is frequently substantially higher, leaving these operators suitable only for non-interactive applications. Among the fastest of the local operators is the bilateral filter due to the optimizations afforded by this technique (Durand & Dorsey, 2002).

This filter is interesting as a tone reproduction operator by itself, or it may be used to compute a local adaptation level for use in a sigmoidal compression function. In either case, the filter respects sharp contrast changes and smoothes over smaller contrasts. This is an important feature that helps minimize halo artifacts which are a common problem with local operators.

An alternative approach to minimize halo artifacts is the scale selection mechanism used in the photographic tone reproduction operator (Reinhard et al., 2002), although this technique is slower to compute.

In summary, while a large number of tone reproduction operators is currently available, only a small number of fundamentally different approaches exist. Fourier-domain and gradient-domain operators are both rooted in knowledge of image formation. Spatial-domain operators are either spatially variant (local) or global in nature. These operators are usually based on insights gained from studying the human visual system (and the visual system of many other species).

# 23

# Global Illumination

Many surfaces in the real world receive most or all of their incident light from other reflective surfaces. This is often called *indirect lighting* or *mutual illumination*. For example, the ceilings of most rooms receive little or no illumination directly from luminaires (light emitting objects). The direct and indirect components of illumination are shown in Figure 23.1.

Although accounting for the interreflection of light between surfaces is straightforward, it is potentially costly because all surfaces may reflect any given surface, resulting in as many as $O(N^2)$ interactions for $N$ surfaces. Because the entire global database of objects may illuminate any given object, accounting for indirect illumination is often called the *global illumination* problem.

There is a rich and complex literature on solving the global illumination problem (e.g., (Appel, 1968; Goral, Torrance, Greenberg, & Battaile, 1984; Cook et

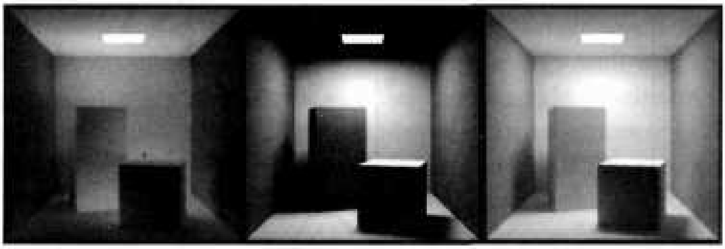


**Figure 23.1.** In the left and middle images, the indirect and direct lighting, respectively, are separated out. On the right, the sum of both components is shown. Global illumination algorithms account for both the direct and the indirect lighting.

al., 1984; Immel et al., 1986; Kajiya, 1986; Malley, 1988)). In this chapter we discuss two algorithms as examples: particle tracing and path tracing. The first is useful for walkthrough applications such as maze games, and as a component of batch rendering. The second is useful for realistic batch rendering. Then we discuss separating out "direct" lighting where light takes exactly once bounce between luminaire and camera.

## 23.1 Particle Tracing for Lambertian Scenes

Recall the transport equation from Section 19.2:

$$L_s(\mathbf{k}_o) = \int_{\text{all } \mathbf{k}_i} \rho(\mathbf{k}_i, \mathbf{k}_o) L_f(\mathbf{k}_i) \cos\theta_i d\sigma_i.$$

The geometry for this equation is shown Figure 23.2. When the illuminated point is Lambertian, this equation reduces to:

$$L_s = \frac{R}{\pi} \int_{\text{all } \mathbf{k}_i} L_f(\mathbf{k}_i) \cos\theta_i d\sigma_i,$$

where $R$ is the diffuse reflectance. One way to approximate the solution to this equation is to use finite element methods. First, we break the scene into $N$ surfaces each with unknown surface radiance $L_i$, reflectance $R_i$, and emitted radiance $E_i$. This results in the set of $N$ simultaneous linear equations

$$L_i = E_i + \frac{R_i}{\pi} \sum_{j=1}^{N} k_{ij} L_j,$$

where $k_{ij}$ is a constant related to the original integral representation. We then solve this set of linear equations, and we can render $N$ constant-colored polygons. This finite element approach is often called *radiosity*.

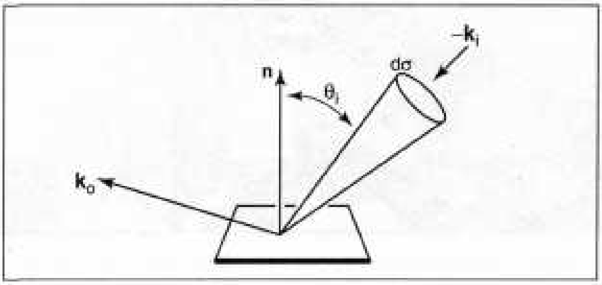


**Figure 23.2.** The geometry for the transport equation in its directional form.

An alternative method to radiosity is to use a statistical simulation approach by randomly following light "particles" from the luminaire though the environment. This is a type of *particle tracing*. There are many algorithms that use some form of particle tracing; we will discuss a form of particle tracing that deposits light in the textures on triangles. First, we review some basic radiometric relations. The radiance $L$ of a Lambertian surface with area $A$ is directly proportional to the incident power per unit area:

$$L = \frac{\Phi}{\pi A}, \tag{23.1}$$

where $\Phi$ is the outgoing power from the surface. Note that in this discussion, all radiometric quantities are either spectral or RGB depending on the implementation. If the surface has emitted power $\Phi_e$, incident power $\Phi_i$, and reflectance $R$, then this equation becomes

$$L = \frac{\Phi_e + R\Phi_i}{\pi A}.$$

If we are given a model with $\Phi_e$ and $R$ specified for each triangle, we can proceed luminaire by luminaire, firing power in the form of particles from each luminaire. We associate a texture map with each triangle to store accumulated radiance, with all texels initialized to

$$L = \frac{\Phi_e}{\pi A}.$$

If a given triangle has area $A$ and $n_t$ texels, and it is hit by a particle carrying power $\phi$, then the radiance of that texel is incremented by

$$\Delta L = \frac{n_t \phi}{\pi A}.$$

Once a particle hits a surface, we increment the radiance of the texel it hits, probabilistically decide whether to reflect the particle, and if we reflect it we choose a direction and adjust its power.

Note that we want the particle to terminate at some point. For each surface we can assign a reflection probability $p$ to each surface interaction. A natural choice would be to let $p = R$ as it is with light in nature. The particle would then scatter around the environment not losing or gaining any energy until it is absorbed. This approach works well when the particles carry a single wavelength (Walter, Hubbard, Shirley, & Greenberg, 1997). However, when a spectrum or RGB triple is carried by the ray as is often implemented (Jensen, 2001), there is no single $R$ and some compromise for the value of $p$ should be chosen. The power $\phi'$ for reflected particles should be adjusted to account for the possible extinction of the particles:

$$\phi' = \frac{R\phi}{p}$$

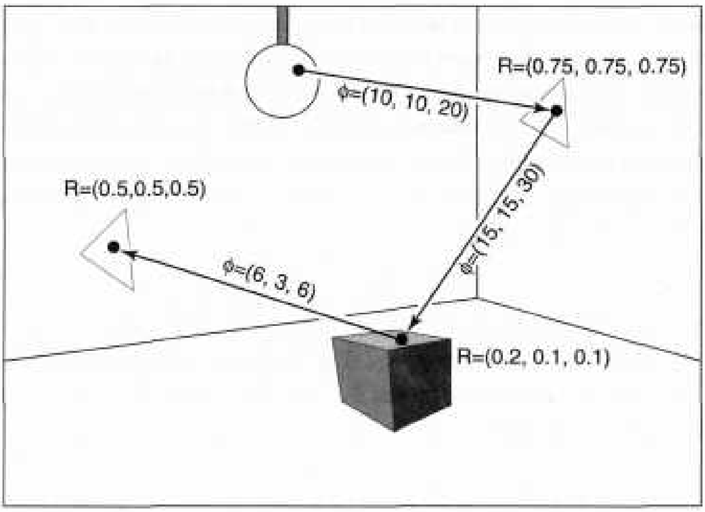


**Figure 23.3.** The path of a particle that survives with probability 0.5 and is absorbed at the last intersection. The RGB power is shown for each path segment.

Note that $p$ can be set to any positive constant less than one, and that this constant can be different for each interaction. When $p > R$ for a given wavelength, the particle will gain power at that wavelength, and when $p < R$ it will lose power at that wavelength. The case where it gains power will not interfere with convergence because the particle will stop scattering and be terminated at some point as long as $p < 1$. For the remainder of this discussion we set $p = 0.5$. The path of a single particle in such a system is shown in Figure 23.3.

A key part to this algorithm is that we scatter the light with an appropriate distribution for Lambertian surfaces. As discussed in Section 14.4.1, we can find a vector with a cosine (Lambertian) distribution by transforming two canonical random numbers $(\xi_1, \xi_2)$ as follows:

$$\mathbf{a} = \left(\cos(2\pi\xi_1)\sqrt{\xi_2}, \sin(2\pi\xi_1)\sqrt{\xi_2}, \sqrt{1-\xi_2}\right). \qquad (23.2)$$

Note that this assumes the normal vector is parallel to the $z$-axis. For a triangle, we must establish an orthonormal basis with $\mathbf{w}$ parallel to the normal vector. We can accomplish this as follows:

$$\mathbf{w} = \frac{\mathbf{n}}{\|\mathbf{n}\|},$$
$$\mathbf{u} = \frac{\mathbf{p}_1 - \mathbf{p}_0}{\|\mathbf{p}_1 - \mathbf{p}_0\|},$$
$$\mathbf{v} = \mathbf{w} \times \mathbf{u},$$

where $\mathbf{p}_i$ are the vertices of the triangle. Then, by definition, our vector in the appropriate coordinates is

$$\mathbf{a} = \cos(2\pi\xi_1)\sqrt{\xi_2}\mathbf{u} + \sin(2\pi\xi_1)\sqrt{\xi_2}\mathbf{v} + \sqrt{1-\xi_2}\mathbf{w}. \qquad (23.3)$$

In pseudocode our algorithm for $p = 0.5$ and one luminaire is:

```
for (Each of $n$ particles) do
   RGB $phi = \Phi/n$
   compute uniform random point $\mathbf{a}$ on luminaire
   compute random direction $\mathbf{b}$ with cosine density
   done = false
   while not done do
      if (ray $\mathbf{a} + t\mathbf{b}$ hits at some point $\mathbf{c}$ ) then
         add $n_t R\phi/(\pi A)$ to appropriate texel
         if ($\xi_1 > 0.5$) then
            $\phi = 2R\phi$
            $\mathbf{a} = \mathbf{c}$
            $\mathbf{b}$ = random direction with cosine density
      else
         done = true
```

Here $\xi_i$ are canonical random numbers. Once this code has run, the texture maps store the radiance of each triangle and can be rendered directly for any viewpoint with no additional computation.

## 23.2 Path Tracing

While particle tracing is well suited to precomputation of the radiances of diffuse scenes, it is problematic for creating images of scenes with general BRDFs or scenes that contain many objects. The most straightforward way to create images of such scenes is to use *path tracing* (Kajiya, 1986). This is a probabilistic method that sends rays from the eye and traces them back to the light. Often path tracing is used only to compute the indirect lighting. Here we will present it in a way that captures all lighting, which can be inefficient. This is sometimes called *brute force* path tracing. In Section 23.3, more efficient techniques for direct lighting can be added.

In path tracing, we start with the full transport equation:

$$L_s(\mathbf{k}_o) = L_e(\mathbf{k}_o) + \int_{\text{all } \mathbf{k}_i} \rho(\mathbf{k}_i, \mathbf{k}_o) L_f(\mathbf{k}_i) \cos\theta_i d\sigma_i.$$

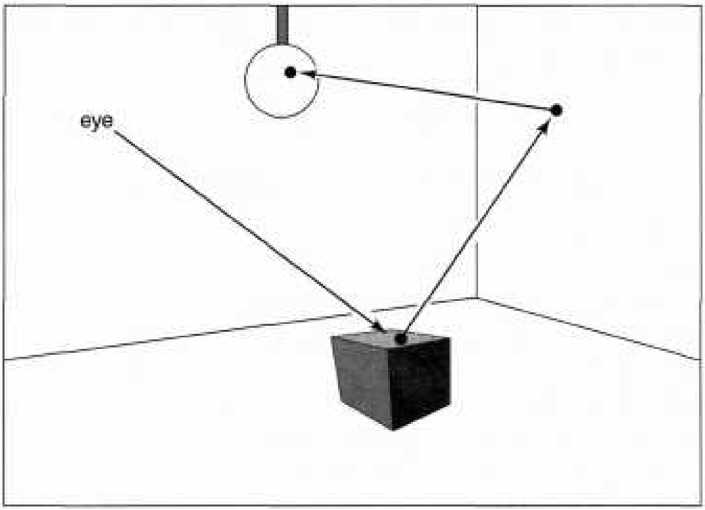


**Figure 23.4.** In path tracing, a ray is followed through a pixel from the eye and scattered through the scene until it hits a luminaire.

We use Monte Carlo integration to approximate the solution to this equation for each viewing ray. Recall from Section 14.3, that we can use random samples to approximate an integral:

$$\int_{x \in S} g(x) d\mu \approx \frac{1}{N} \sum_{i=1}^{N} \frac{g(x_i)}{p(x_i)},$$

where the $x_i$ are random points with probability density function $p$. If we apply this directly to the transport equation with $N = 1$ we get

$$L_s(\mathbf{k}_o) \approx L_e(\mathbf{k}_o) + \frac{\rho(\mathbf{k}_i, \mathbf{k}_o) L_f(\mathbf{k}_i) \cos\theta_i d\sigma_i}{p(\mathbf{k}_i)}.$$

So if we have a way to select random directions $\mathbf{k}_i$ with a known density $p$, we can get an estimate. The catch is that $L_f(\mathbf{k}_i)$ is itself an unknown. Fortunately we can apply recursion and use a statistical estimate for $L_f(\mathbf{k}_i)$ by sending a ray in that direction to find the surface seen in that direction. We end when we hit a luminaire and $L_e$ is non-zero (Figure 23.4). This method assumes lights have zero reflectance, or we would continue to recurse.

In the case of a Lambertian BRDF ($\rho = R/\pi$), we can use a cosine density function:

$$p(\mathbf{k}_i) = \frac{\cos\theta_i}{\pi}.$$

A direction with this density can be chosen according to Equation 23.3. This allows some cancellation of cosine terms in our estimate:

$$L_o(\mathbf{k}_o) \approx L_e(\mathbf{k}_o) - R L_f(\mathbf{k}_i).$$

In pseudocode such a path tracer for Lambertian surfaces would operate just like the ray tracers described in Chapter 10, but the *raycolor* function would be modified:

```
RGB raycolor(ray $\mathbf{a} + t\mathbf{b}$, int depth)
if (ray hits at some point $\mathbf{c}$) then
  RGB $c = L_e(-\mathbf{b})$
  if (depth < maxdepth) then
    compute random direction $\mathbf{d}$
    return $c + R$ raycolor($\mathbf{c} + s\mathbf{d}$, depth+1)
else
  return background color
```

This will result in a very noisy image unless either large luminaires or very large numbers of samples are used. Note the color of the luminaires must be well above one (sometimes thousands or tens of thousands) to make the surfaces have final colors near one, because only those rays that hit a luminaire by chance will make a contribution, and most rays will contribute only a color near zero. To generate the random direction **d**, we use the same technique as we do in particle tracing (see Equation 23.2).

In the general case we might want to use spectral colors or use a more general BRDF. In practice, we should have the material class contain member functions to compute a random direction as well as compute the $p$ associated with that direction. This way materials can be added transparently to an implementation.

## 23.3 Accurate Direct Lighting

This section presents a more physically-based method of direct lighting than Chapter 9. These methods will be useful in making global illumination algorithms more efficient. The key idea is to send shadow rays to the luminaires as described in Chapter 10, but to do so with careful bookkeeping based on the transport equation from the previous chapter. The global illumination algorithms can be adjusted to make sure they compute the direct component exactly once. For example, in particle tracing, particles coming directly from the luminaire would not be logged, so the particles would only encode indirect lighting. This makes nice looking shadows much more efficiently than computing direct lighting in the context of global illumination.

### 23.3.1 Mathematical Framework

To calculate the direct light from one *luminaire* (light emitting object) onto a non-emitting surface, we solve a form of the transport equation from Section 19.2:

$$L_s(\mathbf{x},\mathbf{k}_o) = \int_{\text{all } \mathbf{x}'} \frac{\rho(\mathbf{k}_i,\mathbf{k}_o) L_e(\mathbf{x}',-\mathbf{k}_i) v(\mathbf{x},\mathbf{x}') \cos\theta_i \cos\theta'}{\|\mathbf{x}-\mathbf{x}'\|^2} dA'. \qquad (23.4)$$

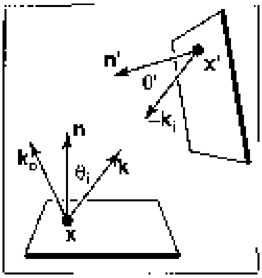


**Figure 23.5.** The direct lighting terms for Equation 23.4.

Recall that $L_e$ is the emitted radiance of the source, $v$ is a visibility function that is equal to 1 if $\mathbf{x}$ "sees" $\mathbf{x}'$ and zero otherwise, and the other variables are as illustrated in Figure 23.5.

If we are to sample Equation 23.4 using Monte Carlo integration, we need to pick a random point $\mathbf{x}'$ on the surface of the luminaire with density function $p$ (so $\mathbf{x}' \sim p$). Just plugging into Equation 14.5 with one sample yields

$$L_s(\mathbf{x},\mathbf{k}_o) \approx \frac{\rho(\mathbf{k}_i,\mathbf{k}_o) L_e(\mathbf{x}',-\mathbf{k}_i) v(\mathbf{x},\mathbf{x}') \cos\theta_i \cos\theta'}{p(\mathbf{x}')\|\mathbf{x}-\mathbf{x}'\|^2}. \qquad (23.5)$$

If we pick a uniform random point on the luminaire, then $p = 1/A$, where $A$ is the area of the luminaire. This gives

$$L_s(\mathbf{x},\mathbf{k}_o) \approx \frac{\rho(\mathbf{k}_i,\mathbf{k}_o) L_e(\mathbf{x}',-\mathbf{k}_i) v(\mathbf{x},\mathbf{x}') A \cos\theta_i \cos\theta'}{\|\mathbf{x}-\mathbf{x}'\|^2}. \qquad (23.6)$$

We can use Equation 23.6 to sample planar (e.g., rectangular) luminaires in a straightforward fashion. We simply pick a random point on each luminaire.

The code for one luminaire is:

```
color directLight( $\mathbf{x}$, $\mathbf{k}_o$, $\mathbf{n}$)
pick random point $\mathbf{x}'$ with normal vector $\mathbf{n}'$ on light
$\mathbf{d} = \mathbf{x}' - \mathbf{x}$
$\mathbf{k}_i = \mathbf{d}/\|d\|$
if (ray $\mathbf{x} + t\mathbf{d}$ has no hits for $t < 1 - \epsilon$) then
  return $\rho(\mathbf{k}_i,\mathbf{k}_o) L_e(\mathbf{x}',-\mathbf{k}_i)(\mathbf{n}\cdot\mathbf{d})(-\mathbf{n}'\cdot\mathbf{d})/\|\mathbf{d}\|^4$
else
  return 0
```

The above code needs some extra tests such as clamping the cosines to zero if they are negative. Note that the term $\|\mathbf{d}\|^4$ comes from the distance squared term

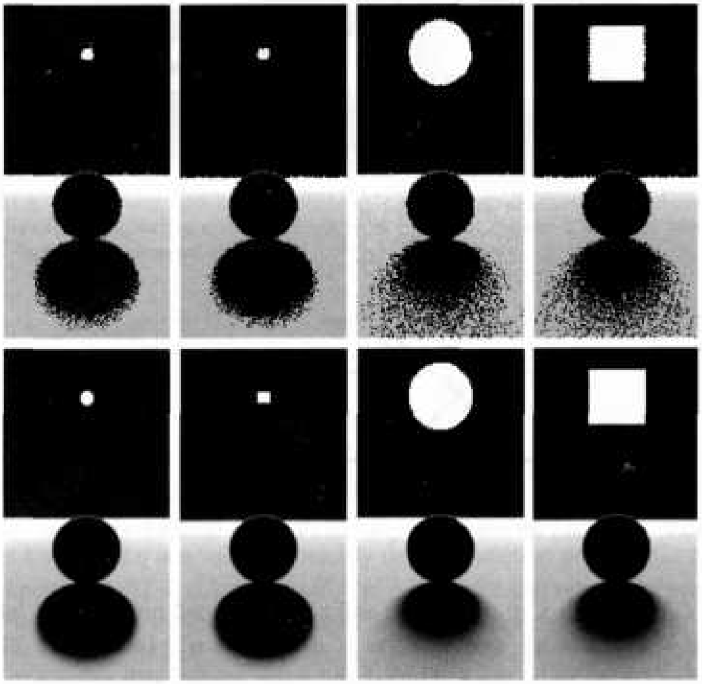


**Figure 23.6.** Various soft shadows on a backlit sphere with a square and an area light source. Top: 1 sample. Bottom: 100 samples. Note that the shape of the light source is less important than its size in determining shadow appearance.

and the two cosines, e.g., $\mathbf{n} \cdot \mathbf{d} = \|\mathbf{d}\| \cos\theta$ because $\mathbf{d}$ is not necessarily a unit vector.

Several examples of soft shadows are shown in Figure 23.6.

### 23.3.2 Sampling a Spherical Luminaire

Although a sphere with center $\mathbf{c}$ and radius $R$ can be sampled using Equation 23.6, this sampling will yield a very noisy image because many samples will be on the back of the sphere, and the $\cos\theta'$ term varies so much. Instead, we can use a more complex $p(\mathbf{x}')$ to reduce noise.

The first non-uniform density we might try is $p(\mathbf{x}') \propto \cos\theta'$. This turns out to be just as complicated as sampling with $p(\mathbf{x}') \propto \cos\theta'/\|\mathbf{x}' - \mathbf{x}\|^2$, so we instead discuss that here. We observe that sampling on the luminaire this way is the same as using a constant density function $q(\mathbf{k}_i) = \text{const}$ defined in the space of directions subtended by the luminaire as seen from $\mathbf{x}$. We now use a coordinate

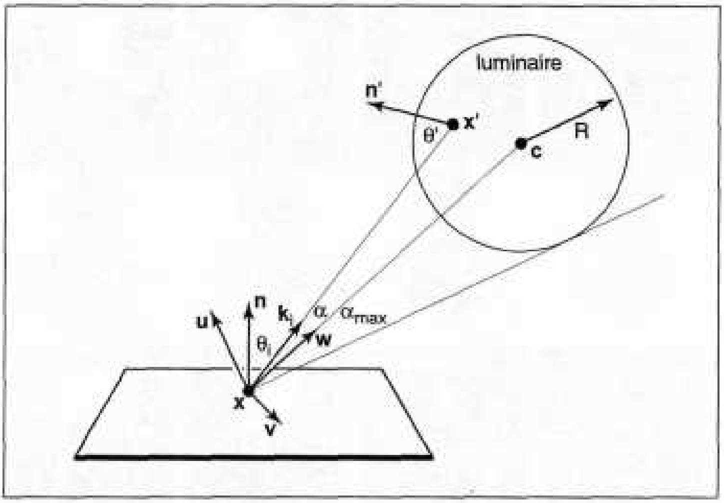


**Figure 23.7.** Geometry for direct lighting at point **x** from a spherical luminaire.

system defined with $\mathbf{x}$ at the origin, and a right-handed orthonormal basis with $\mathbf{w} = (\mathbf{c} - \mathbf{x})/\|\mathbf{c} - \mathbf{x}\|$, and $\mathbf{v} = (\mathbf{w} \times \mathbf{n})/\|(\mathbf{w} \times \mathbf{n})\|$ (see Figure 23.7). We also define $(\alpha, \phi)$ to be the azimuthal and polar angles with respect to the *uvw* coordinate system.

The maximum $\alpha$ that includes the spherical luminaire is given by

$$\alpha_{\max} = \arcsin\left(\frac{R}{\|\mathbf{x} - \mathbf{c}\|}\right) = \arccos\sqrt{1 - \left(\frac{R}{\|\mathbf{x} - \mathbf{c}\|}\right)^2}.$$

Thus, a uniform density (with respect to solid angle) within the cone of directions subtended by the sphere is just the reciprocal of the solid angle $2\pi(1 - \cos\alpha_{\max})$ subtended by the sphere:

$$q(\mathbf{k}_i) = \frac{1}{2\pi\left(1 - \sqrt{1 - \left(\frac{R}{\|\mathbf{x}-\mathbf{c}\|}\right)^2}\right)}.$$

And we get

$$\begin{bmatrix} \cos\alpha \\ \phi \end{bmatrix} = \begin{bmatrix} 1 - \xi_1 + \xi_1\sqrt{1 - \left(\frac{R}{\|\mathbf{x}-\mathbf{c}\|}\right)^2} \\ 2\pi\xi_2 \end{bmatrix}.$$

This gives us the direction $\mathbf{k}_i$. To find the actual point, we need to find the first point on the sphere in that direction. The ray in that direction is just $(\mathbf{x} + t\mathbf{k}_i)$,

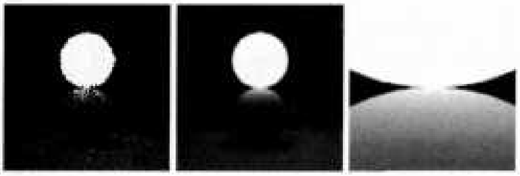


**Figure 23.8.** A sphere with $L_e$ = 1 touching a sphere of reflectance 1. Where the two spheres touch, the reflective sphere should have $L(\mathbf{x}')$ = 1 . Left: 1 sample. Middle: 100 samples. Right: 100 samples, close-up.

where $\mathbf{k}_i$ is given by

$$\mathbf{k}_i = \begin{bmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{bmatrix} \begin{bmatrix} \cos\phi\sin\alpha \\ \sin\phi\sin\alpha \\ \cos\alpha \end{bmatrix}.$$

We must also calculate $p(\mathbf{x}')$, the probability density function with respect to the area measure (recall that the density function $q$ is defined in solid angle space). Since we know that $q$ is a valid probability density function using the $\omega$ measure, and we know that $d\Omega = dA(\mathbf{x}')\cos\theta'/\|\mathbf{x}' - \mathbf{x}\|^2$, we can relate any probability density function $q(\mathbf{k}_i)$ with its associated probability density function $p(\mathbf{x}')$:

$$q(\mathbf{k}_i) = \frac{p(\mathbf{x}')\cos\theta'}{\|\mathbf{x}' - \mathbf{x}\|^2}. \tag{23.7}$$

So we can solve for $p(\mathbf{x}')$:

$$p(\mathbf{x}') = \frac{\cos\theta'}{2\pi\|\mathbf{x}' - \mathbf{x}\|^2 \left(1 - \sqrt{1 - \left(\frac{R}{\|\mathbf{x} - \mathbf{c}\|}\right)^2}\right)}.$$

A good debugging case for this is shown in Figure 23.8.

### 23.3.3 Non-Diffuse Luminaries

There is no reason the luminance of the luminaire cannot vary with both direction and position. For example, it can vary with position if the luminaire is a television. It can vary with direction for car headlights and other directional sources. Little in our analysis need change from the previous sections, except that $L_e(\mathbf{x}')$ must change to $L_e(\mathbf{x}', -\mathbf{k}_i)$. The simplest way to vary the intensity with direction is to use a Phong-like pattern with respect to the normal vector $\mathbf{n}'$. To avoid using an exponent in the term for the total light output, we can use the form

$$L_e(\mathbf{x}', -\mathbf{k}_i) = \frac{(n+1)E(\mathbf{x}')}{2\pi} cos^{(n-1)}\theta',$$

where $E(\mathbf{x}')$ is the *radiant exitance* (power per unit area) at point $\mathbf{x}'$, and $n$ is the Phong exponent. You get a diffuse light for $n = 1$. If the light is non-uniform across its area, e.g., as a television set is, then $E$ will not be a constant.

## Frequently Asked Questions

- My pixel values are no longer in some sensible zero-to-one range. What should I display?

You should use one of the *tone reproduction* techniques described in Chapter 22.

- What global illumination techniques are used in practice?

For batch rendering of complex scenes, path tracing with one level of reflection is often used. Path tracing is often augmented with a particle tracing preprocess as described in Jensen's book in the chapter notes. For walkthrough games, some form of world-space preprocess is often used, such as the particle tracing described in this chapter. For scenes with very complicated specular transport, an elegant but involved method, Metropolis Light Transport (Veach & Guibas, 1997) may be the best choice.

- How does the ambient component relate to global illumination?

For diffuse scenes, the radiance of a surface is proportional to the product of the irradiance at the surface and the reflectance of the surface. The ambient component is just an approximation to the irradiance scaled by the inverse of $\pi$. So although it is a crude approximation, there can be some methodology to guessing it (M. F. Cohen, Chen, Wallace, & Greenberg, 1988), and it is probably more accurate than doing nothing, i.e., using zero for the ambient term. Because the indirect irradiance can vary widely within a scene, using a different constant for each surface can be used for better results rather than using a global ambient term.

- Why do most algorithms compute direct lighting using traditional ray tracing?

Although global illumination algorithms automatically compute direct lighting, and it is in fact slightly more complicated to make them compute only indirect lighting, it is usually faster to compute direct lighting separately. There are three reasons for this. First, indirect lighting tends to be smooth compared to

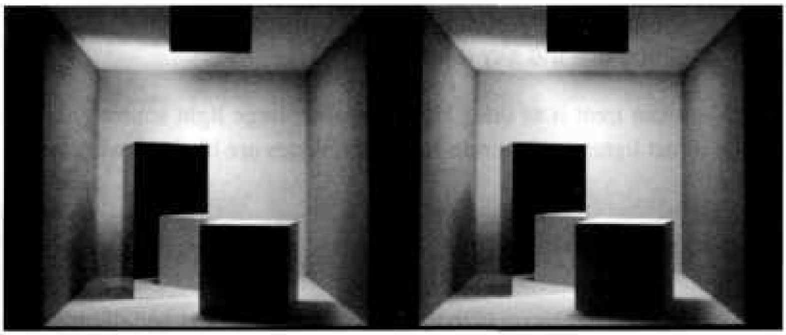


**Figure 23.9.** A comparison between a rendering and a photo. *Figure courtesy Sumant Pattanaik and the Cornell Program of Computer Graphics.* (See also Plate XXI.)

direct lighting (see Figure 23.1) so coarser representations can be used, e.g., low-resolution texture maps for particle tracing. The second reason is that light sources tend to be small, and it is rare to hit them by chance in a "from the eye" method such as path tracing, while direct shadow rays are efficient. The third reason is that direct lighting allows stratified sampling so it converges rapidly compared to unstratified sampling. The issue of stratification is the reason that shadow rays are used in Metropolis Light Transport despite the stability of its default technique for dealing with direct lighting as just one type of path to handle.

- How artificial is it to assume ideal diffuse and specular behavior?

For environments that have only matte and mirrored surfaces, the Lambertian/specular assumption works well. A comparison between a rendering using that assumption and a photograph is shown in Figure 23.9.

- How many shadow rays are needed per pixel?

Typically between 16 and 400. Using narrow penumbra, a large ambient term (or a large indirect component), and a masking texture (Ferwerda, Shirley, Pattanaik, & Greenberg, 1997) can reduce the number needed.

- How do I sample something like a filament with a metal reflector where much of the light is reflected from the filament?

Typically the whole light is replaced by a simple source that approximates its aggregate behavior. For viewing rays, the complicated source is used. So a car headlight would look complex to the viewer, but the lighting code might see simple disk-shaped lights.

- Isn't something like the sky a luminaire?

Yes, and you can treat it as one. However, such large light sources may not be helped by direct lighting; the brute-force techniques are likely to work better.

## Notes

Global illumination has its roots in the fields of heat transfer and illumination engineering as documented in *Radiosity: A Programmer's Perspective* (Ashdown, 1994). Other good books related to global illumination include *Radiosity and Global Illumination* (M. F. Cohen & Wallace, 1993), *Radiosity and Realistic Image Synthesis* (Sillion & Puech, 1994), *Principles of Digital Image Synthesis* (Glassner, 1995), *Realistic Image Synthesis Using Photon Mapping* (Jensen, 2001), *Advanced Global Illumination* (Dutré, Bala, & Bekaert, 2002), and *Physically Based Rendering* (Pharr & Humphreys, 2004). The probabilistic methods discussed in this chapter are from *Monte Carlo Techniques for Direct Lighting Calculations* (Shirley, Wang, & Zimmerman, 1996).

## Exercises

1. For a closed environment, where every surface is a diffuse reflector and emittor with reflectance $R$ and emitted radiance $E$, what is the total radiance at each point? *Hint: for* $R = 0.5$ *and* $E = 0.25$ *the answer is* $0.5$. This is an excellent debugging case.

2. Using the definitions from Chapter 19, verify Equation 23.1.

3. If we want to render a typically-sized room with textures at centimeter-square resolution, approximately how many particles should we send to get an average of about 1000 hits per texel?

4. Develop a method to take random samples with uniform density from a disk.

5. Develop a method to take random samples with uniform density from a triangle.

6. Develop a method to take uniform random samples on a "sky dome" (the inside of a hemisphere).

# 24

# Reflection Models

As we discussed in Chapter 19, the reflective properties of a surface can be summarized using the BRDF (Nicodemus, Richmond, Hsia, Ginsberg, & Limperis, 1977; Cook & Torrance, 1982). In this chapter, we discuss some of the most visually important aspects of material properties and a few fairly simple models that are useful in capturing these properties. There are many BRDF models in use in graphics, and the models presented here are meant to give just an idea of non-diffuse BRDFs.

## 24.1 Real-World Materials

Many real materials have a visible structure at normal viewing distances. For example, most carpets have easily visible pile that contributes to appearance. For our purposes, such structure is not part of the material property but is, instead, part of the geometric model. Structure whose details are invisible at normal viewing distances, but which do determine macroscopic material appearance, are part of the material property. For example, the fibers in paper have a complex appearance under magnification, but they are blurred together into an homogeneous appearance when viewed at arm's length. This distinction between microstructure that is folded into BRDF is somewhat arbitrary and depends on what one defines as "normal" viewing distance and visual acuity, but the distinction has proven quite useful in practice.

In this section we define some categories of materials. Later in the chapter, we present reflection models that target each type of material. In the notes at the end of the chapter some models that account for more exotic materials are also discussed.

### 24.1.1 Smooth Dielectrics and Metals

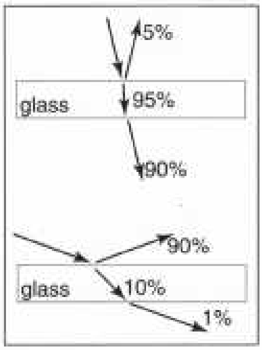


**Figure 24.1.** The amount of light reflected and transmitted by glass varies with the angle.

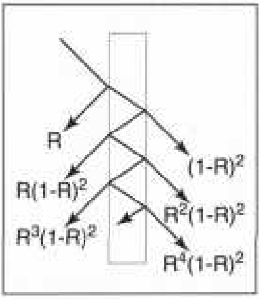


**Figure 24.2.** Light is repeatedly reflected and refracted by glass, with the fractions of energy shown.

Dielectrics are clear materials that refract light; their basic properties were summarized in Chapter 10. Metals reflect and refract light much like dielectrics, but they absorb light very, very quickly. Thus, only very thin metal sheets are transparent at all, e.g., the thin gold plating on some glass objects. For a smooth material, there are only two important properties:

1. How much light is reflected at each incident angle and wavelength;
2. What fraction of light is absorbed as it travels through the material for a given distance and wavelength.

The amount of light transmitted is whatever is not reflected (a result of energy conservation). For a metal, in practice, we can assume all the light is immediately absorbed. For a dielectric, the fraction is determined by the constant used in Beer's Law as discussed in Chapter 10.

The amount of light reflected is determined by the *Fresnel equations* as discussed in Chapter 10. These equations are straightforward, but cumbersome. The main effect of the Fresnel Equations is to increase the reflectance as the incident angle increases, particularly near grazing angles. This effect works for transmitted light as well. These ideas are shown diagrammatically in Figure 24.1. Note that the light is repeatedly reflected and refracted as shown in Figure 24.2. Usually only one or two of the reflected images is easily visible.

### 24.1.2 Rough Surfaces

If a metal or dielectric is roughened to a small degree, but not so small that diffraction occurs, then we can think of it as a surface with *microfacets* (Cook & Torrance, 1982). Such surfaces behave specularly at a closer distance, but viewed at a further distance seem to spread the light out in a distribution. For a metal, an example of this rough surface might be brushed steel, or the "cloudy" side of most aluminum foil.

For dielectrics, such as a sheet of glass, scratches or other irregular surface features make the glass blur the reflected and transmitted images that we can normally see clearly. If the surface is heavily scratched, we call it *translucent* rather than transparent. This is a somewhat arbitrary distinction, but it is usually clear whether we would consider a glass translucent or transparent.

### 24.1.3 Diffuse Materials

A material is *diffuse* if it is matte, i.e., not shiny. Many surfaces we see are diffuse, such as most stones, paper, and unfinished wood. To a first approximation, diffuse surfaces can be approximated with a Lambertian (constant) BRDF. Real diffuse materials usually become somewhat specular for grazing angles. This is a subtle effect, but can be important for realism.

### 24.1.4 Translucent Materials

Many thin objects, such as leaves and paper, both transmit and reflect light diffusely. For all practical purposes no clear image is transmitted by these objects. These surfaces can add a hue shift to the transmitted light. For example, red paper is red because it filters out non-red light for light that penetrates a short distance into the paper, and then scatters back out. The paper also transmits light with a red hue because the same mechanisms apply, but the transmitted light makes it all the way through the paper. One implication of this property is that the transmitted coefficient should be the same in both directions.

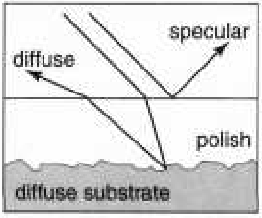


**Figure 24.3.** Light hitting a layered surface can be reflected specularly, or it can be transmitted and then scatter diffusely off the substrate.

### 24.1.5 Layered Materials

Many surfaces are composed of "layers" or are dielectrics with embedded particles that give the surface a diffuse property (Phong, 1975). The surface of such materials reflects specularly as shown in Figure 24.3, and thus obeys the Fresnel equations. The light that is transmitted is either absorbed or scattered back up to the dielectric surface where it may or may not be transmitted. That light that is transmitted, scattered, and then retransmitted in the opposite direction forms a diffuse "reflection" component.

Note that the diffuse component also is attenuated with the degree of the angle, because the Fresnel equations cause reflection back into the surface as the angle increases as shown in Figure 24.4. Thus instead of a constant diffuse BRDF, one that vanishes near the grazing angle is more appropriate.

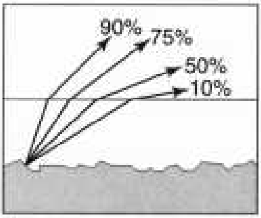


**Figure 24.4.** The light scattered by the substrate is less and less likely to make it out of the surface as the angle relative to the surface normal increases.

## 24.2 Implementing Reflection Models

A BRDF model, as described in Section 19.1.6, will produce a rendering which is more physically based than the rendering we get from point light sources and Phong-like models. Unfortunately, real BRDFs are typically quite complicated and cannot be deduced from first principles. Instead, they must either be measured and directly approximated from raw data, or they must be crudely approximated in an empirical fashion. The latter empirical strategy is what is usually done, and the development of such approximate models is still an area of research. This section discusses several desirable properties of such empirical models.

First, physical constraints imply two properties of a BRDF model. The first constraint is energy conservation:

$$\text{for all } \mathbf{k}_i,\ R(\mathbf{k}_i) = \int_{\text{all } \mathbf{k}_o} \rho(\mathbf{k}_i, \mathbf{k}_o) \cos\theta_o \, d\sigma_o \leq 1.$$

If you send a beam of light at a surface from any direction $\mathbf{k}_i$, then the total amount of light reflected over all directions will be at most the incident amount. The second physical property we expect all BRDFs to have is reciprocity:

$$\text{for all } \mathbf{k}_i, \mathbf{k}_o,\ \rho(\mathbf{k}_i, \mathbf{k}_o) = \rho(\mathbf{k}_o, \mathbf{k}_i).$$

Second, we want a clear separation between diffuse and specular components. The reason for this is that, although there is a mathematically-clean delta function formulation for ideal specular components, delta functions must be implemented as special cases in practice. Such special cases are only practical if the BRDF model clearly indicates what is specular and what is diffuse.

Third, we would like intuitive parameters. For example, one reason the Phong model has enjoyed such longevity is that its diffuse constant and exponent are both clearly related to the intuitive properties of the surface, namely surface color and highlight size.

Finally, we would like the BRDF function to be amenable to Monte Carlo sampling. Recall from Chapter 14 that an integral can be sampled by $N$ random points $x_i \sim p$ where $p$ is defined with the same measure as the integral:

$$\int f(x) d\mu \approx \frac{1}{N} \sum_{j=1}^{N} \frac{f(x_j)}{p(x_j)}.$$

Recall from Section 19.2 that the surface radiance in direction $\mathbf{k}_o$ is given by a transport equation:

$$L_s(\mathbf{k}_o) = \int_{\text{all } \mathbf{k}_i} \rho(\mathbf{k}_i, \mathbf{k}_o) L_f(\mathbf{k}_i) \cos\theta_i d\sigma_i.$$

If we sample directions with pdf $p(\mathbf{k}_i)$ as discussed in Chapter 23, then we can approximate the surface radiance with samples:

$$L_s(\mathbf{k}_o) \approx \frac{1}{N} \sum_{j=1}^{N} \frac{\rho(\mathbf{k}_j, \mathbf{k}_o) L_f(\mathbf{k}_j) \cos\theta_j}{p(\mathbf{k}_j)}.$$

This approximation will converge for any $p$ that is non-zero where the integrand is non-zero. However, it will only converge well if the integrand is not very large relative to $p$. Ideally, $p(\mathbf{k})$ should be approximately shaped like the integrand $\rho(\mathbf{k}_j, \mathbf{k}_o)L_f(\mathbf{k}_j)\cos\theta_j$. In practice, $L_f$ is complicated, and the best we can accomplish is to have $p(\mathbf{k})$ shaped somewhat like $\rho(\mathbf{k}, \mathbf{k}_o)L_f(\mathbf{k})\cos\theta$.

For example, if the BRDF is Lambertian, then it is constant and the "ideal" $p(\mathbf{k})$ is proportional to $\cos\theta$. Because the integral of $p$ must be one, we can deduce the leading constant:

$$\int_{\text{all } \mathbf{k} \text{ with } \theta < \pi/2} C \cos\theta d\sigma = 1.$$

This implies that $C = 1/\pi$, so we have

$$p(\mathbf{k}) = \frac{1}{\pi}\cos\theta.$$

An acceptably efficient implementation will result as long as $p$ doesn't get too small when the integrand is non-zero. Thus, the constant pdf will also suffice:

$$p(\mathbf{k}) = \frac{1}{2\pi}.$$

This emphasizes that many pdfs may be acceptable for a given BRDF model.

## 24.3 Specular Reflection Models

For a metal, we typically specify the reflectance at normal incidence $R_0(\lambda)$. The reflectance should vary according to the Fresnel equations, and a good approximation is given by (Schlick, 1994a)

$$R(\theta, \lambda) = R_0(\lambda) + (1 - R_0(\lambda))(1 - \cos\theta)^5.$$

This approximation allows us to just set the normal reflectance of the metal either from data or by eye.

For a dielectric, the same formula works for reflectance. However, we can set $R_0(\lambda)$ in terms of the refractive index $n(\lambda)$:

$$R_0(\lambda) = \left(\frac{n(\lambda) - 1}{n(\lambda) + 1}\right)^2.$$

Typically, $n$ does not vary with wavelength, but for applications where dispersion is important, $n$ can vary. The refractive indices that are often useful include water ($n = 1.33$), glass ($n = 1.4$ to $n = 1.7$), and diamond ($n = 2.4$).

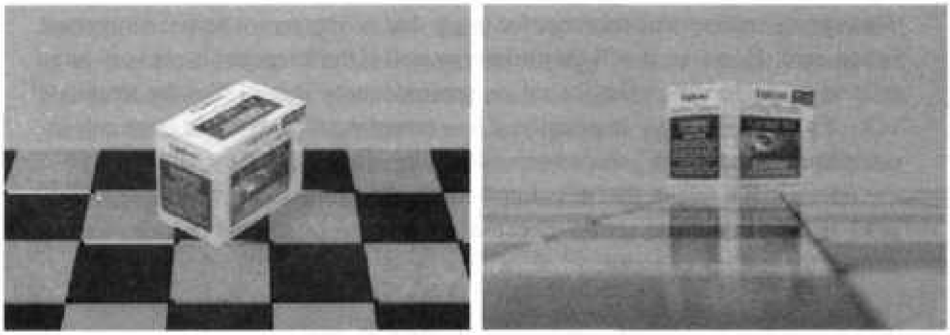


**Figure 24.5.** Renderings of polished tiles using coupled model. These images were produced using a Monte Carlo path tracer. The sampling distribution for the diffuse term is $\cos\theta/\pi$.

## 24.4 Smooth Layered Model

Reflection in matte/specular materials, such as plastics or polished woods, is governed by Fresnel equations at the surface and by scattering within the subsurface. An example of this reflection can be seen in the tiles in the renderings in Figure 24.5. Note that the blurring in the specular reflection is mostly vertical due to the compression of apparent bump spacing in the view direction. This effect causes the vertically-streaked reflections seen on lakes on windy days; it can either be modeled using explicit micro-geometry and a simple smooth-surface reflection model or by a more general model that accounts for this asymmetry.

We could use the traditional Lambertian-specular model for the tiles, which linearly mixes specular and Lambertian terms. In standard radiometric terms, this can be expressed as

$$\rho(\theta,\phi,\theta',\phi'\lambda) = \frac{R_d(\lambda)}{\pi} + R_s\rho_s(\theta,\phi,\theta',\phi'),$$

where $R_d(\lambda)$ is the hemispherical reflectance of the matte term, $R_s$ is the specular reflectance, and $\rho_s$ is the normalized specular BRDF (a weighted Dirac delta function on the sphere). This equation is a simplified version of the BRDF where $R_s$ is independent of wavelength. The independence of wavelength causes a highlight that is the color of the luminaire, so a polished rather than a metal appearance will be achieved. Ward (G. J. Ward, 1992) suggests to set $R_d(\lambda) + R_s \leq 1$ in order to conserve energy. However, such models with constant $R_s$ fail to show the increase in specularity for steep viewing angles. This is the key point: in the real world the relative proportions of matte and specular appearance change with the viewing angle.

One way to simulate the change in the matte appearance is to explicitly dampen $R_d(\lambda)$ as $R_s$ increases (Shirley, 1991):

$$\rho(\theta, \phi, \theta', \phi', \lambda) = R_f(\theta)\rho_s(\theta, \phi, \theta', \phi') + \frac{R_d(\lambda)(1 - R_f(\theta))}{\pi},$$

where $R_f(\theta)$ is the Fresnel reflectance for a polish-air interface. The problem with this equation is that it is not reciprocal, as can been seen by exchanging $\theta$ and $\theta'$; this changes the value of the matte damping factor because of the multiplication by $(1 - R_f(\theta))$. The specular term, a scaled Dirac delta function, is reciprocal, but this does not make up for the non-reciprocity of the matte term. Although this BRDF works well, its lack of reciprocity can cause some rendering methods to have ill-defined solutions.

We now present a model that produces the matte/specular tradeoff while remaining reciprocal and energy conserving. Because the key feature of the new model is that it couples the matte and specular scaling coefficients, it is called a *coupled* model (Shirley, Smits, Hu, & Lafortune, 1997).

Surfaces which have a glossy appearance are often a clear dielectric, such as polyurethane or oil, with some subsurface structure. The specular (mirror-like) component of the reflection is caused by the smooth dielectric surface and is independent of the structure below this surface. The magnitude of this specular term is governed by the Fresnel equations.

The light that is not reflected specularly at the surface is transmitted through the surface. There, either it is absorbed by the subsurface, or it is reflected from a pigment or a subsurface and transmitted back through the surface of the polish. This transmitted light forms the matte component of reflection. Since the matte component can only consist of the light that is transmitted, it will naturally decrease in total magnitude for increasing angle.

To avoid choosing between physically plausible models and models with good qualitative behavior over a range of incident angles, note that the Fresnel equations that account for the specular term, $R_f(\theta)$, are derived directly from the physics of the dielectric-air interface. Therefore, the problem must lie in the matte term. We could use a full-blown simulation of subsurface scattering as implemented, but this technique is both costly and requires detailed knowledge of subsurface structure, which is usually neither known nor easily measurable. Instead, we can modify the matte term to be a simple approximation that captures the important qualitative angular behavior shown in Figure 24.4.

Let us assume that the matte term is not Lambertian, but instead is some other function that depends only on $\theta$, $\theta'$ and $\lambda$: $\rho_m(\theta, \theta', \lambda)$. We discard behavior that depends on $\phi$ or $\phi'$ in the interest of simplicity. We try to keep the formulas reasonably simple because the physics of the matte term is complicated and sometimes requires unknown parameters. We expect the matte term to be close to constant, and roughly rotationally symmetric (He et al., 1992).

An obvious candidate for the matte component $\rho_m(\theta, \theta', \lambda)$ that will be reciprocal is the *separable* form $kR_m(\lambda)f(\theta)f(\theta')$ for some constant $k$ and matte reflectance parameter $R_m(\lambda)$. We could merge $k$ and $R_m(\lambda)$ into a single term, but we choose to keep them separated because this makes it more intuitive to set $R_m(\lambda)$—which must be between 0 and 1 for all wavelengths. Separable BRDFs have been shown to have several computational advantages, thus we use the separable model:

$$\rho(\theta, \phi, \theta', \phi', \lambda) = R_f(\theta)\rho_s(\theta, \phi, \theta', \phi') + kR_m(\lambda)f(\theta)f(\theta').$$

We know that the matte component can only contain energy not reflected in the surface (specular) component. This means that for $R_m(\lambda) = 1$, the incident and reflected energy are the same, which suggests the following constraint on the BRDF for each incident $\theta$ and $\lambda$:

$$R_f(\theta) + 2\pi k f(\theta)\int_0^{\frac{\pi}{2}} f(\theta')\cos\theta'\sin\theta' d\theta' = 1. \tag{24.1}$$

We can see that $f(\theta)$ must be proportional to $(1 - R_f(\theta))$. If we assume that matte components that absorb some energy have the same directional pattern as this ideal, we get a BRDF of the form

$$\rho(\theta, \phi, \theta', \phi', \lambda) = R_f(\theta)\rho_s(\theta, \phi, \theta', \phi') + kR_m(\lambda)[1 - R_f(\theta)][1 - R_f(\theta')].$$

We could now insert the full form of the Fresnel equations to get $R_f(\theta)$, and then use energy conservation to solve for constraints on $k$. Instead, we will use the approximation discussed in Section 24.1.1 We find that

$$f(\theta) \propto (1 - (1 - \cos\theta)^5).$$

Applying Equation 24.1 yields

$$k = \frac{21}{20\pi(1 - R_0)}. \tag{24.2}$$

The full coupled BRDF is then

$$\begin{aligned}\rho(\theta, \phi, \theta', \phi', \lambda) = &\\ &\left[R_0 + (1-\cos\theta)^5(1-R_0)\right]\rho_s(\theta, \phi, \theta', \phi') + \\ &kR_m(\lambda)\left[1 - (1 - \cos\theta)^5\right]\left[1 - (1 - \cos\theta')^5\right].\end{aligned} \tag{24.3}$$

The results of running the coupled model is shown in Figure 24.5. Note that for the high viewpoint, the specular reflection is almost invisible, but it is clearly visible in the low-angle photograph image, where the matte behavior is less obvious.

For reasonable values of refractive indices, $R_0$ is limited to approximately the range 0.03 to 0.06 (the value $R_0 = 0.05$ was used for Figure 24.5). The value of $R_s$ in a traditional Phong model is harder to choose, because it typically must be tuned for viewpoint in static images and tuned for a particular camera sequence for animations. Thus, the coupled model is easier to use in a "hands-off" mode.

## 24.5 Rough Layered Model

The previous model is fine if the surface is smooth. However, if the surface is not ideal, some spread is needed in the specular component. An extension of the coupled model to this case is presented here (Ashikhmin & Shirley, 2000). At a given point on a surface, the BRDF is a function of two directions, one in the direction towards the light and one in the direction towards the viewer. We would like to have a BRDF model that works for "common" surfaces, such as metal and plastic, and has the following characteristics:

1. **Plausible:** as defined by Lewis (Lewis, 1994), this refers to the BRDF obeying energy conservation and reciprocity.

2. **Anisotropy:** the material should model simple anisotropy, such as seen on brushed metals.

3. **Intuitive parameters:** for material, such as plastics, there should be parameters $R_d$ for the substrate and $R_s$ for the normal specular reflectance as well as two roughness parameters $n_u$ and $n_v$.

4. **Fresnel behavior:** specularity should increase as the incident angle decreases.

5. **Non-Lambertian diffuse term:** the material should allow for a diffuse term, but the component should be non-Lambertian to assure energy conservation in the presence of Fresnel behavior.

6. **Monte Carlo friendliness:** there should be some reasonable probability density function that allows straightforward Monte Carlo sample generation for the BRDF.

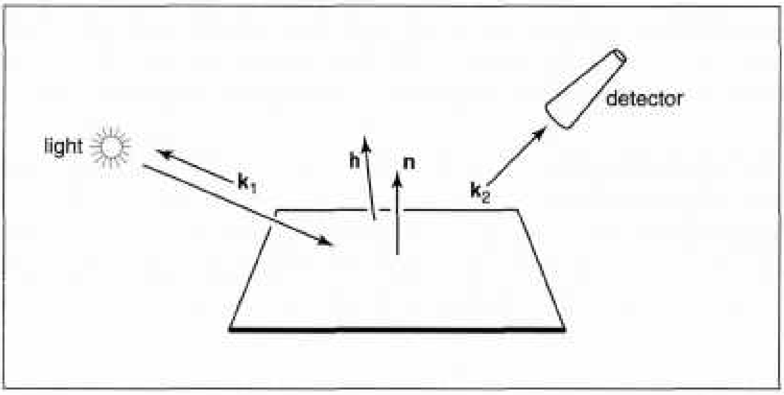


**Figure 24.6.** Geometry of reflection. Note that $\mathbf{k}_1$, $\mathbf{k}_2$, and $\mathbf{h}$ share a plane, which usually does not include $\mathbf{n}$.

A BRDF with these properties is a Fresnel-weighted Phong-style cosine lobe model that is anisotropic.

We again decompose the BRDF into a specular component and a diffuse component (Figure 24.6). Accordingly, we write our BRDF as the classical sum of two parts:

$$\rho(\mathbf{k}_1, \mathbf{k}_2) = \rho_s(\mathbf{k}_1, \mathbf{k}_2) + \rho_d(\mathbf{k}_1, \mathbf{k}_2), \qquad (24.4)$$

where the first term accounts for the specular reflection (this will be presented in the next section). While it is possible to use the Lambertian BRDF for the diffuse term $\rho_d(\mathbf{k}_1, \mathbf{k}_2)$ in our model, we will discuss a better solution in Section 24.5.2 and how to implement the model in Section 24.5.3. Readers who just want to implement the model should skip to that section.

### 24.5.1 Anisotropic Specular BRDF

To model the specular behavior, we use a Phong-style specular lobe but make this lobe anisotropic and incorporate Fresnel behavior while attempting to preserve the simplicity of the initial mode. This BRDF is

$$\rho(\mathbf{k}_1, \mathbf{k}_2) = \frac{\sqrt{(n_u + 1)(n_v + 1)}}{8\pi} \frac{(\mathbf{n} \cdot \mathbf{h})^{n_u \cos^2 \phi + n_v \sin^2 \phi}}{(\mathbf{h} \cdot \mathbf{k}_i)\max(\cos\theta_i, \cos\theta_o)} F(\mathbf{k}_i \cdot \mathbf{h}). \qquad (24.5)$$

Again we use Schlick's approximation to the Fresnel equation:

$$F(\mathbf{k}_i \cdot \mathbf{h}) = R_s + (1 - R_s)(1 - (\mathbf{k}_i \cdot \mathbf{h}))^5, \qquad (24.6)$$

where $R_s$ is the material's reflectance for the normal incidence. Because $\mathbf{k}_i \cdot \mathbf{h} = \mathbf{k}_o \cdot \mathbf{h}$, this form is reciprocal. We have an empirical model whose terms are

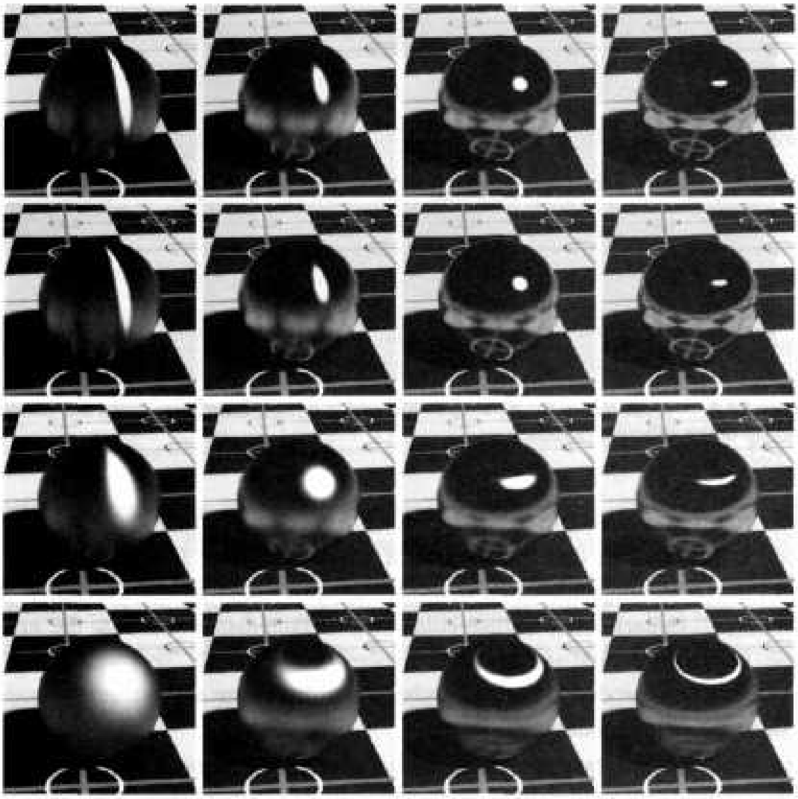


**Figure 24.7.** Metallic spheres for exponents 10, 100, 1000, 10000 increasing both left-to-right and top-to-bottom.

chosen to enforce energy conservation and reciprocity. A full rationalization for the terms is given in the paper by Ashikhmin, listed in the chapter notes.

The specular BRDF of Equation 24.5 is useful for representing metallic surfaces where the diffuse component of reflection is very small. Figure 24.7 shows a set of metal spheres on a texture-mapped Lambertian plane. As the values of parameters $n_u$ and $n_v$ change, the appearance of the spheres shift from rough metal to almost perfect mirror, and from highly anisotropic to the more familiar Phong-like behavior.

### 24.5.2 Diffuse Term for the Anisotropic Phong Model

It is possible to use a Lambertian BRDF together with the anisotropic specular term; this is done for most models, but it does not necessarily conserve energy. A

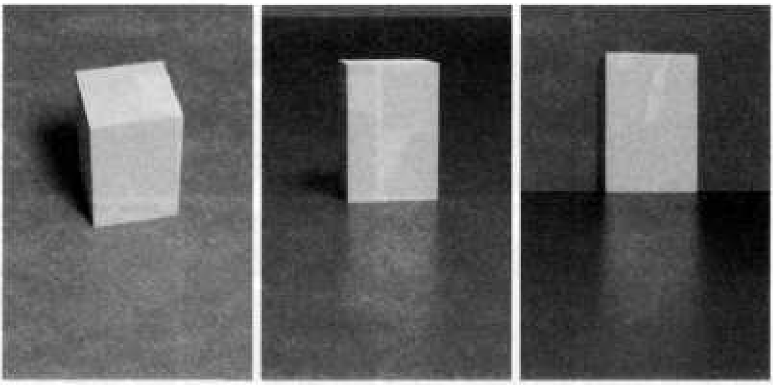


**Figure 24.8.** Three views for $n_u = n_v = 400$ and a diffuse substrate. Note the change in intensity of the specular reflection.

better approach is a simple angle-dependent form of the diffuse component which accounts for the fact that the amount of energy available for diffuse scattering varies due to the dependence of the specular term's total reflectance on the incident angle. In particular, diffuse color of a surface disappears near the grazing angle, because the total specular reflectance is close to one. This well-known effect cannot be reproduced with a Lambertian diffuse term and is therefore missed by most reflection models.

Following a similar approach to the coupled model, we can find a form of the diffuse term that is compatible with the anisotropic Phong lobe:

$$\rho_d(\mathbf{k}_1, \mathbf{k}_2) = \frac{28R_d}{23\pi}(1 - R_s)\left(1 - \left(1 - \frac{\cos\theta_i}{2}\right)^5\right)\left(1 - \left(1 - \frac{\cos\theta_o}{2}\right)^5\right). \tag{24.7}$$

Here $R_d$ is the diffuse reflectance for normal incidence, and $R_s$ is the Phong lobe coefficient. An example using this model is shown in Figure 24.8.

### 24.5.3 Implementing the Model

Recall that the BRDF is a combination of diffuse and specular components:

$$\rho(\mathbf{k}_1, \mathbf{k}_2) = \rho_s(\mathbf{k}_1, \mathbf{k}_2) + \rho_d(\mathbf{k}_1, \mathbf{k}_2). \tag{24.8}$$

The diffuse component is given in Equation 24.7; the specular component is given in Equation 24.5. It is not necessary to call trigonometric functions to compute the exponent, so the specular BRDF can be written:

$$\rho(\mathbf{k}_1, \mathbf{k}_2) = \frac{\sqrt{(n_u+1)(n_v+1)}}{8\pi} (\mathbf{n}\cdot\mathbf{h})^{\frac{(n_u(\mathbf{h}\cdot\mathbf{u})^2 + n_v(\mathbf{h}\cdot\mathbf{v})^2)/(1-(\mathbf{hn})^2)}{(\mathbf{h}\cdot\mathbf{k}_i)\max(\cos\theta_i, \cos\theta_o)}} F(\mathbf{k}_i \cdot \mathbf{h}). \tag{24.9}$$

In a Monte Carlo setting, we are interested in the following problem: given $\mathbf{k}_1$, generate samples of $\mathbf{k}_2$ with a distribution whose shape is similar to the cosine-weighted BRDF. Note that greatly undersampling a large value of the integrand is a serious error, while greatly oversampling a small value is acceptable in practice. The reader can verify that the densities suggested below have this property.

A suitable way to construct a pdf for sampling is to consider the distribution of half vectors that would give rise to our BRDF. Such a function is

$$p_h(\mathbf{h}) = \frac{\sqrt{(n_u+1)(n_v+1)}}{2\pi} (\mathbf{nh})^{n_u \cos^2\phi + n_v \sin^2\phi}, \tag{24.10}$$

where the constants are chosen to ensure it is a valid pdf.

We can just use the probability density function $p_h(\mathbf{h})$ of Equation 24.10 to generate a random $\mathbf{h}$. However, to evaluate the rendering equation, we need both a reflected vector $\mathbf{k}_o$ and a probability density function $p(\mathbf{k}_o)$. It is important to note that if you generate $\mathbf{h}$ according to $p_h(\mathbf{h})$ and then transform to the resulting $\mathbf{k}_o$:

$$\mathbf{k}_o = -\mathbf{k}_i + 2(\mathbf{k}_i \cdot \mathbf{h})\mathbf{h}, \tag{24.11}$$

the density of the resulting $\mathbf{k}_o$ is **not** $p_h(\mathbf{k}_o)$. This is because of the difference in measures in $\mathbf{h}$ and $\mathbf{k}_o$. So the actual density $p(\mathbf{k}_o)$ is

$$p(\mathbf{k}_o) = \frac{p_h(\mathbf{h})}{4(\mathbf{k}_i\mathbf{h})}. \tag{24.12}$$

Note that in an implementation where the BRDF is known to be this model, the estimate of the rendering equation is quite simple as many terms cancel out.

It is possible to generate an $\mathbf{h}$ vector whose corresponding vector $\mathbf{k}_o$ will point inside the surface, i.e., $\cos\theta_o < 0$. The weight of such a sample should be set to zero. This situation corresponds to the specular lobe going below the horizon and is the main source of energy loss in the model. Clearly, this problem becomes progressively less severe as $n_u$, $n_v$ become larger.

The only thing left now is to describe how to generate $\mathbf{h}$ vectors with the pdf of Equation 24.10. We will start by generating $\mathbf{h}$ with its spherical angles in the range $(\theta, \phi) \in [0, \frac{\pi}{2}] \times [0, \frac{\pi}{2}]$. Note that this is only the first quadrant of the hemisphere. Given two random numbers $(\xi_1, \xi_2)$ uniformly distributed in $[0, 1]$, we can choose

$$\phi = \arctan\left(\sqrt{\frac{n_u+1}{n_v+1}} \tan\left(\frac{\pi\xi_1}{2}\right)\right), \tag{24.13}$$

and then use this value of $\phi$ to obtain $\theta$ according to

$$\cos\theta = (1-\xi_2)^{1/(n_u \cos^2\phi + n_v \sin^2\phi + 1)}. \qquad (24.14)$$

To sample the entire hemisphere, we use the standard manipulation where $\xi_1$ is mapped to one of four possible functions depending on whether it is in $[0, 0.25)$, $[0.25, 0.5)$, $[0.5, 0.75)$, or $[0.75, 1.0)$. For example for $\xi_1 \in [0.25, 0.5)$, find $\phi(1 - 4(0.5 - \xi_1))$ via Equation 24.13, and then "flip" it about the $\phi = \pi/2$ axis. This ensures full coverage and stratification.

For the diffuse term, use a simpler approach and generate samples according to a cosine distribution. This is sufficiently close to the complete diffuse BRDF to substantially reduce variance of the Monte Carlo estimation.

## Frequently Asked Questions

- My images look too smooth, even with a complex BRDF. What am I doing wrong?

BRDFs only capture subpixel detail that is too small to be resolved by the eye. Most real surfaces also have some small variations, such as the wrinkles in skin, that can be seen. If you want true realism, some sort of texture or displacement map is needed.

- How do I integrate the BRDF with texture mapping?

Texture mapping can be used to control any parameter on a surface. So any kinds of colors or control parameters used by a BRDF should be programmable.

- I have very pretty code except for my material class. What am I doing wrong?

You are probably doing nothing wrong. Material classes tend to be the ugly thing in everybody's programs. If you find a nice way to deal with it, please let me know! My own code uses a shader architecture (Hanrahan & Lawson, 1990) which makes the material include much of the rendering algorithm.

## Notes

There are many BRDF models described in the literature, and only a few of them have been described here. Others include (Cook & Torrance, 1982; He et al., 1992; G. J. Ward, 1992; Oren & Nayar, 1994; Schlick, 1994a; Lafortune, Foo, Torrance, & Greenberg, 1997; Stam, 1999; Ashikhmin, Premože, & Shirley, 2000; Ershov, Kolchin, & Myszkowski, 2001; Matusik, Pfister, Brand, & McMillan, 2003; Lawrence, Rusinkiewicz, & Ramamoorthi, 2004; Stark, Arvo, & Smits, 2005). The desired characteristics of BRDF models is discussed in *Making Shaders More Physically Plausible* (Lewis, 1994).

## Exercises

1. Suppose that instead of the Lambertian BRDF we used a BRDF of the form $C \cos^a \theta_i$. What must $C$ be to conserve energy?

2. The BRDF in Exercise 1 is not reciprocal. Can you modify it to be reciprocal?

3. Something like a highway sign is a *retroreflector*. This means that the BRDF is large when $\mathbf{k}_i$ and $\mathbf{k}_o$ are near each other. Make a model inspired by the Phong model that captures retroreflection behavior while being reciprocal and conserving energy.

# 25

# Image-Based Rendering

A classic conflict in computer graphics is that between visual realism and the ability to interact. One attempt to deal with this problem is to use a set of captured or precomputed realistic images and to attempt to interpolate new images for novel viewpoints (Chen & Williams, 1993). This approach is called *image-based rendering*, abbreviated *IBR*.

The basic idea of IBR idea is illustrated in 2D for a database of two images in Figure 25.1. Given two images, we approximate an image as seen from a

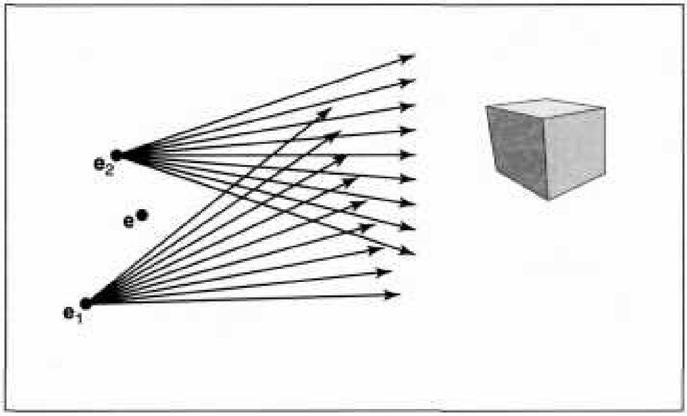


**Figure 25.1.** Given two images as seen from $\mathbf{e}_1$ and $\mathbf{e}_2$, image-based rendering can be used to make an approximation to the image that would be seen from a new viewpoint $\mathbf{e}$.

novel viewpoint. The quality of this approximation depends on the detail of the two source images, the underlying geometry of the object, and the relation of the three points. In this chapter, we discuss the most brute-force IBR method which uses very regular samples and straightforward interpolation.

## 25.1 The Light Field

For every point in space, light is passing through it in every direction. For a given point $\mathbf{a}$ and direction $\mathbf{d}$, the amount of light is quantified by the radiance (see Section 19.1.5). For the set of all points and all directions, we can describe the radiance at every location/direction pair as a function $L$ that we evaluate to get the radiance:

$$L(\mathbf{a}, \mathbf{d}) \equiv \text{the radiance at point } \mathbf{a} \text{ in direction } \mathbf{d}.$$

An image is just a set of evaluations of this $L$ function for a given eyepoint $\mathbf{a}$ and a structured set of directions $\mathbf{d}_i$. Technically, this function varies with wavelength (spectral radiance) and time. Usually we will think of RGB moments of the radiance and a steady-state in time. This $L$ function has no standard name, but the most common one used in graphics is the *light field*.

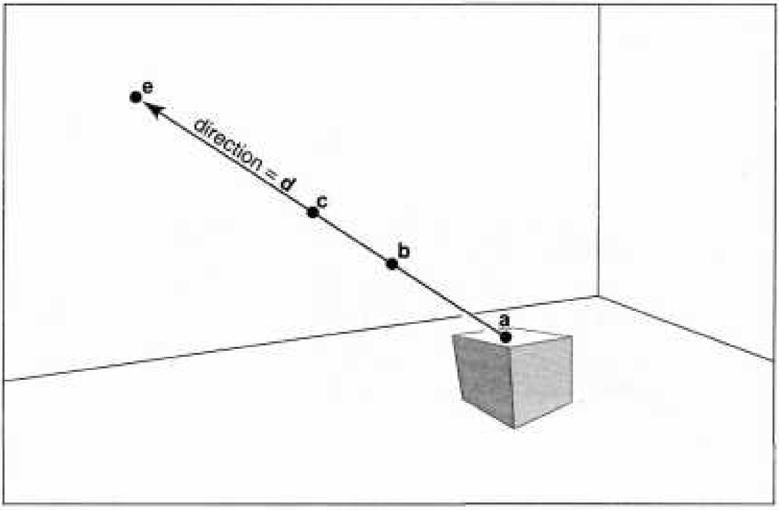


**Figure 25.2.** Several point/direction pairs in the light field. Because they lie along the same light ray, they have the same value.

There is a great deal of structure in the light field. Most importantly, $L$ does not vary along a line for a fixed direction. This is illustrated in Figure 25.2, where

$$L(\mathbf{a},\mathbf{d}) = L(\mathbf{b},\mathbf{d}) = L(\mathbf{c},\mathbf{d}) = L(\mathbf{e},\mathbf{d}).$$

Note that if there in a object along the line, then the light field may be different for points on either side of the object.

Before we can try to approximate values of the light field at novel viewpoints, we must establish the dimensionality of the light field. At first glance it is 5D, because it varies over 3D position and 2D direction. Indeed, this is the dimensionality of the function inside a participating medium, such as smoke. However, because the value of the function does not vary along a line, we can create a *line-space* and evaluate the function for a directed line in 3D space:

$$L(\mathbf{A}) \equiv \text{the radiance along a directed line } \mathbf{A}.$$

A line in 3D is a 4D entity (see Section 14.1.3). This means we should be able to store radiance samples along rays as points in a 4D space. A way to do this is explored in the next section.

## 25.2 Creating a Novel Image from a Set of Images

If we want to create an image from a novel viewpoint using only images from a precomputed set of images, then the key is to organize the data for this pur-

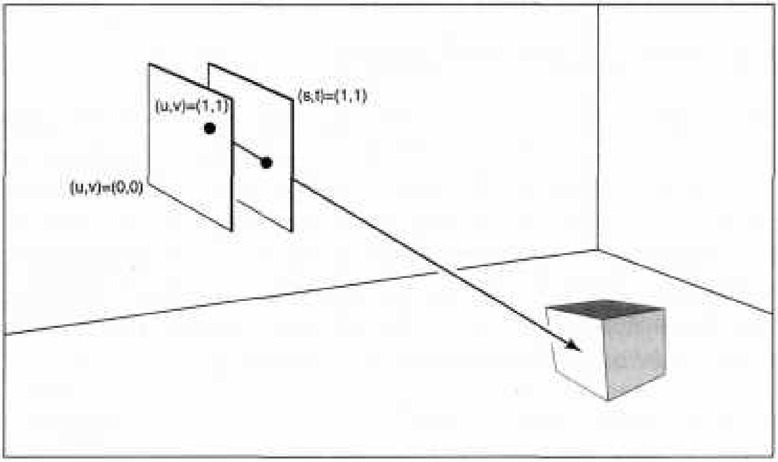


**Figure 25.3.** Any ray through the two rectangles can be parameterized by the two sets of texture coordinates, (u,v) and (s,t) of the hitpoints of the rays and the rectangle.

pose (Gortler, Grzeszczuk, Szeliski, & Cohen, 1996; Levoy & Hanrahan, 1996). To simplify things, we create a space with dense samples and do some simple form of interpolation. As in any graphics application, we first ask whether rectilinear samples and simple interpolation will work. Recall from Section 14.1.3 that one way to parameterize line space is to use two sets of 2D rectilinear coordinates on a pair of parallel planes. For a finite set of view directions, we can just use the rectilinear coordinates on a pair of parallel rectangles. This idea is illustrated in Figure 25.3, where a given ray is associated with a $(u, v, s, t)$ quadruple.

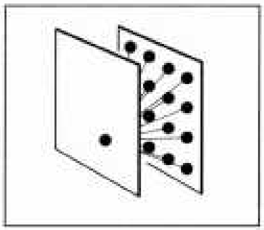


**Figure 25.4.** One (u,v) sample for all (s,t) values can be created with a single traditional rendering pass.

A nice thing about storing radiance samples this way is that we can assemble the database using a traditional renderer. For a given position $\mathbf{e}$ associated with a single $(u, v)$ sample, we can compute an array of samples (pixels) on the $(s, t)$ plane as shown in Figure 25.4. We can then render one image per $(u, v)$ position to complete the 4D database.

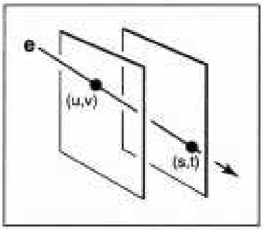


**Figure 25.5.** A ray from a new viewpoint that hits both planes will have a well defined (u,v,s,t) value.

Given a rectilinearly sampled database of radiances in $(u, v, s, t)$ space, and given a novel view position, all rays not behind or parallel to the two planes will have a well defined $(u, v, s, t)$ value (Figure 25.5). For rays through the two rectangles, we will have stored radiances near that 4D point, and we can do some form of interpolation to compute a new value.

The interpolation scheme we use will determine the quality of the image. We could use full quadralinear interpolation between the nearest sixteen data points in $(u, v, s, t)$ space. We could also use nearest-neighbor interpolation which will access less data, but will also result in blockier images.

## Frequently Asked Questions

- What is the best place to store images?

The basic conflict is between organizing images so they are convenient for interpolation versus storing them so they are compact and yield accurate results. For convenient interpolation, we want the simple data structures described in this chapter. For compactness, we want the images stored as near to the surfaces they show as is possible. In the limit, this just means putting the images directly on the surfaces, i.e., texture mapping. Once you do this, a traditional rendering algorithm is most appropriate. The process of using images to create a traditional geometry/texture model is called *image-based modeling*.

- What are the applications of IBR?

This is the million-dollar question. So far they have been limited in practice. However, virtual shopping and web education seem like natural applications for

IBR. In each of these, a set of photographs can be used to make compelling interactive experiences. For example, to browse homes for sale, the ability to move the viewpoint is essential to get a feel for the house. Another natural application is sports. IBR has already been used in a professional football game with linear eye motion and nearest-neighbor (i.e., no) interpolation. In such applications brute-force techniques are likely to dominate, because there is little processing time available between image capture and image display. Perhaps the most promising application for IBR is to create rich material appearance by making textures encode occlusion effects (Dana, Ginneken, Nayar, & Koenderink, 1999). This gives the simplicity of traditional graphics but also makes more convincing detail.

- Is the light field defined at surfaces?

For a point on an opaque surface there is a well defined radiance for each direction. The incoming directions will be incident radiance, and the outgoing directions will be outgoing radiance. These are sometimes called *field radiance* and *surface radiance*, respectively. . Thus, the light field is defined for such surfaces, although it is not continuous (it is zero inside the surface, and we define the light field on the surface as the limit function as taken from outside the surface). For dielectric surfaces, the light field is different for a full sphere of directions for both surface and field radiance, so the light field needs the sets of directions to be well defined in that case.

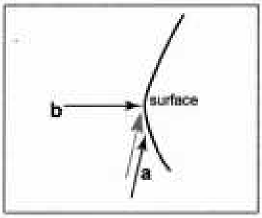


**Figure 25.6.** While the grey arrow is closer to **a**, it may be better to use the value for **b**, because the rays hit at a nearby point.

- What is depth correction?

For matte scenes, we sometimes want to take advantage of approximate geometric information if it exists. Such a case is shown in Figure 25.6. Applying depth correction can give much crisper results, but it does complicate the interpolation scheme.

## Exercises

1. Given a $5\ \mathrm{m} \times 5\ \mathrm{m} \times 3\ \mathrm{m}$ room, how many texels are needed to have texture maps on the walls, floor, and ceiling at 1 square cm resolution? If we wanted to store a light field near the center of the room and use it to reconstruct images without depth correction, how many data points would be needed to reconstruct novel images with the same accuracy as the traditional texture maps?

2. How many operations are needed for nearest-neighbor interpolation in 4D line-space versus full 4D linear interpolation?

# 26

# Visualization

One of the main application areas of computer graphics is *visualization*, where images are used to aid a user in understanding data (Hansen & Johnson, 2005). Sometimes this data has a natural geometric component, such as the elevation data for a region of the Earth. Other data has no obvious geometric meaning, such as trends in the stock market. This non-geometric data might nonetheless benefit from a visual representation, because the human visual system is so good at extracting information from images. The crucial part of visualizing non-geometric data is how the data is mapped to a spatial form. The general area of visualizing non-geometric data is called *information visualization* . This chapter will restrict itself to the more well-understood problems of visualizing 2D and 3D scalar fields, where a scalar data value is defined over a continuous region of space.

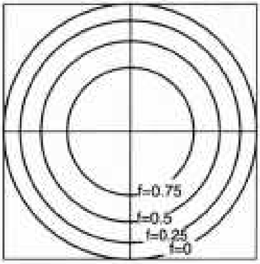


**Figure 26.1.** A contour plot for four levels of the function 1 - $x^2$ - $y^2$.

## 26.1 2D Scalar Fields

For simplicity, assume that our 2D scalar data is defined as

$$f(x,y) = \begin{cases} 1 - x^2 - y^2, & \text{if } x^2 + y^2 < 1, \\ 0 & \text{otherwise,} \end{cases} \qquad (26.1)$$

over the square $(x, y) \in [-1, 1]^2$. In practice, we often have a sampled representation on a rectilinear grid that we interpolate to get a continuous field. We will ignore that issue in 2D for simplicity.

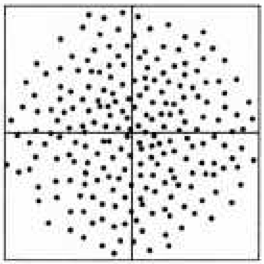


Figure 26.2. A random density plot for four levels of the function 1 - $x^2$ - $y^2$.

One way to visualize a 2D field is to draw lines at a finite set of values $f(x, y) = f_i$ (shown for the function in Equation 26.1 in Figure 26.1). This is done on many topographic maps to indicate elevation. Isocontours are excellent at communicating slope, but are hard to read "globally" to understand large trends and extrema in the data.

Another common way to visualize 2D data is to use small pseudorandom dots whose density is proportional to the value of the function. This is shown for our test function in Figure 26.2. Such random density plots are useful for display on black-and-white media, but are otherwise usually not a good choice for visualization. Random density plots look smoother and smoother as more and smaller dots are used maintaining overall density. As the dot size shrinks below human visual acuity, the image looks smooth. This results in a greyscale continuous tone plot of the function. It is hard for humans to read such plots, because our ability to detect absolute intensity levels is poor. For this reason, color or thresholding is often used. This is shown in greyscale in Figure 26.3. Formally, we can specify such a mapping with just a function $g$ that maps scalar values to colors:

$$g : \mathbb{R} \mapsto [0, 1]^3 .$$

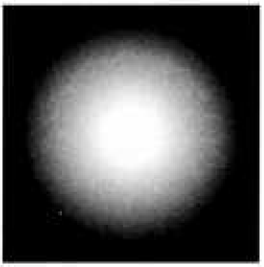


Figure 26.3. A greyscale density plot of the function 1 - $x^2$ - $y^2$.

Here $[0, 1]^3$ refers to the RGB cube. A common strategy is to specify a set of colors to which specific values map and linearly interpolate colors between them. A set of colors that increases in intensity and cycles in hue is often used. Such a set of colors for the domain $[0, 1]$ is

$$\begin{aligned} g(0.00) &= (0.0, 0.0, 0.0) \\ g(0.25) &= (0.0, 0.0, 1.0) \\ g(0.50) &= (1.0, 0.0, 0.0) \\ g(0.75) &= (1.0, 1.0, 0.0) \\ g(1.00) &= (1.0, 1.0, 1.0) \end{aligned}$$

These plots are often called *pseudocolor* displays. We can also display the function as a height plot as shown in Figure 26.4. This type of plot is good for showing the shape of a function. Note that this plot makes it more obvious that the function is spherical.

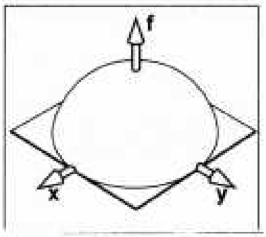


Figure 26.4. A height plot of the function.

Often, more than one of these methods are used together in a single image, such as a colored or contoured height plot. Another hybrid technique that is often used is to shade the height plot and view it orthogonally from above. This is a *shaded relief map*, often used for geographical applications.

## 26.2 3D Scalar Fields

In 3D we can use some of the same techniques as in 2D. We can make a contour plot, where each contour is a 3D surface called an *isosurface*. We can also generalize a random density plot to 3D by scattering particles in 3D. If we take the limit, as we did in 2D to get a pesudocolor display, then we get *direct volume rendering*. These two methods are covered here. It is not clear how to generalize height plots, because we have run out of dimensions.

### 26.2.1 Isosurfaces

Given a 3D scalar field $f(x, y, z)$ we can create an isosurface for $f(x, y, z) = f_0$. In practice, we will have $f$ defined in a 3D rectilinear table that we interpolate for intermediate values. An example image is shown in Figure 26.5

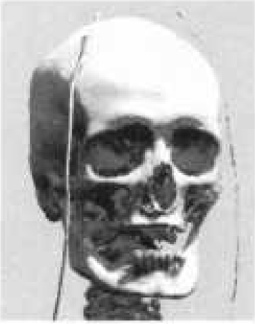


**Figure 26.5.** An isosurface from the NIH/NIM Visible Female data set.

There are two basic approaches to creating images of isosurfaces. The first is to explicitly create a polygonal representation of the isosurface and then render that representation using standard rendering techniques. The second is to use ray tracing to create an image by direct intersection calculation. In ray tracing, no explicit surface is computed. The explicit approach is better when we have small datasets, or we need the isosurface itself rather than just an image of it. The ray tracing approach is better for large datasets where we just need the image of the isosurface.

#### Creating Polygonal Isosurfaces

The basic idea of creating polygonal isosurfaces treats every rectilinear cell as a separate problem (Wyvill, McPheeters, & Wyvill, 1986; Lorensen & Cline, 1987). Given an isovalue $f_0$, there is a surface in the cell if the minimum and maximum of the eight vertex values surround $f_0$. What surfaces occur depend on the arrangement of values above and below $f_0$. This is shown for three cases in Figure 26.6.

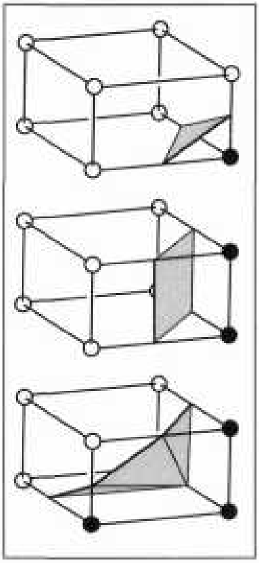


**Figure 26.6.** Three cases for polygonal isosurfacing. The black vertices are on one side of the isovalue, and the white on the other.

There are a total of $2^8 = 256$ cases for vertices above and below the isovalue. We can just enumerate all the cases in a table, and do a look-up. We can also take advantage of some symmetries to reduce the table size. For example, if we reverse above/below vertices, we can halve the table size. If we are willing to do flips and rotations, we can reduce the table to size 16, where only 15 of the cases have polygons.

### Ray Tracing

Although the above algorithm, usually called *marching cubes* is elegant and simple, some care must be taken to ensure accurate results (Nielson, 2003).

The algorithm for intersecting a ray with an isosurface has three phases: traversing a ray through cells which do not contain an isosurface, analytically computing the isosurface when intersecting a voxel containing the isosurface, shading the resulting intersection point (Lin & Ching, 1996; Parker, Parker, et al., 1999). This process is repeated for each pixel on the screen.

To find an intersection, the ray $\mathbf{a} + t\mathbf{b}$ traverses cells in the volume checking each cell to see if its data range bounds an isovalue. If it does, an analytic computation is performed to solve for the ray parameter $t$ at the intersection with the isosurface:

$$\rho(x_a + tx_b, y_a + ty_b, z_a + tz_b) - \rho_{\text{iso}} = 0.$$

When approximating $\rho$ with a trilinear interpolation between discrete grid points, this equation will expand to a cubic polynomial in $t$. This cubic can then be solved in closed form to find the intersections of the ray with the isosurface in that cell. Only the roots of the polynomial which are contained in the cell are examined. There may be multiple roots corresponding to multiple intersection points. In this case, the smallest $t$ (closest to the eye) is used. There may also be no roots of the polynomial, in which case the ray misses the isosurface in the cell.

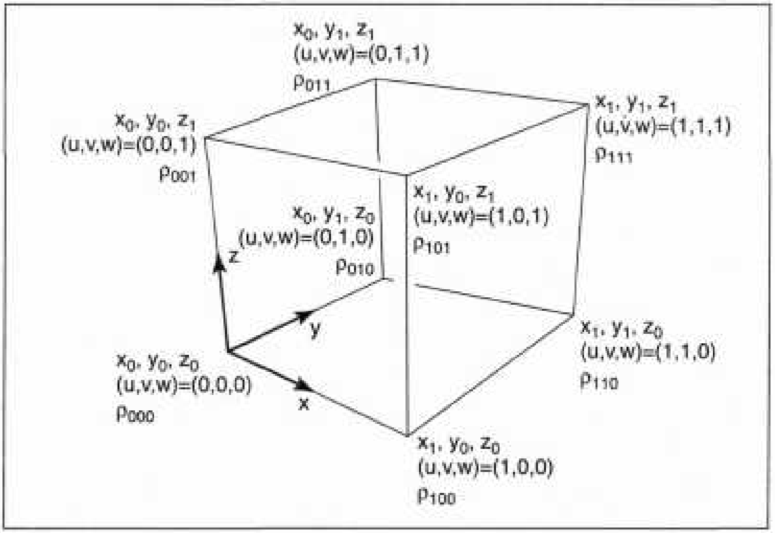


**Figure 26.7.** The geometry for a cell. A "nice" uvw coordinate system is used to make interpolation math cleaner.

A rectilinear volume is composed of a three-dimensional array of point samples that are aligned to the Cartesian axes and are equally spaced in a given dimension. A single cell from such a volume is shown in Figure 26.7. Other cells can be generated by exchanging indices $(i, j, k)$ for the zeros and ones in the figure. The density at a point within the cell is found using *trilinear* interpolation:

$$\begin{aligned} \rho(u,v,w) = {} & (1-u)(1-v)(1-w)\rho_{000} \\ & + (1-u)(1-v)(w)\rho_{001} \\ & + (1-u)(v)(1-w)\rho_{010} \\ & + (u)(1-v)(1-w)\rho_{100} \\ & + (u)(1-v)(w)\rho_{101} \\ & + (1-u)(v)(w)\rho_{011} \\ & + (u)(v)(1-w)\rho_{110} \\ & + (u)(v)(w)\rho_{111}, \end{aligned} \tag{26.2}$$

where

$$\begin{aligned} u &= \frac{x - x_0}{x_1 - x_0}, \\ v &= \frac{y - y_0}{y_1 - y_0}, \\ w &= \frac{z - z_0}{z_1 - z_0}. \end{aligned} \tag{26.3}$$

Note that

$$\begin{aligned} 1 - u &= \frac{x_1 - x}{x_1 - x_0}, \\ 1 - v &= \frac{y_1 - y}{y_1 - y_0}, \\ 1 - w &= \frac{z_1 - z}{z_1 - z_0}. \end{aligned} \tag{26.4}$$

If we redefine $u_0 = 1 - u$ and $u_1 = u$, and use similar definitions for $v_0, v_1, w_0, w_1$, then we get (Figure 26.8)

$$\rho = \sum_{i,j,k=0,1} u_i v_j w_k \rho_{ijk}.$$

It is interesting that the true trilinear isosurface can be fairly complex. The case where two opposite corners of the cube are on opposite sides of the isovalue from

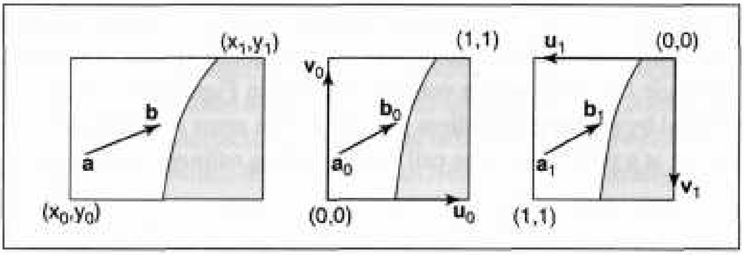


**Figure 26.8.** Various coordinate systems used for interpolation and intersection.

the other six vertices is shown in Figure 26.9. This is quite different from the two triangles given by polygonal isosurfacing for that case. One advantage of direct intersection with the trilinear surface is that ambiguous cases do not arise.

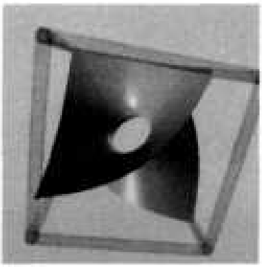


**Figure 26.9.** A true trilinear isosurface generated using direct ray tracing.

For a given point $(x, y, z)$ in the cell, the surface normal is given by the gradient with respect to $(x, y, z)$:

$$\mathbf{N} = \vec{\nabla}\rho = \left(\frac{\partial\rho}{\partial x}, \frac{\partial\rho}{\partial y}, \frac{\partial\rho}{\partial z}\right).$$

Thus, the normal vector of $(N_x, N_Y, N_z) = \vec{\nabla}\rho$ is

$$N_x = \sum_{i,j,k=0,1} \frac{(-1)^{i+1} v_j w_k}{x_1 - x_0} \rho_{ijk},$$

$$N_y = \sum_{i,j,k=0,1} \frac{(-1)^{j+1} u_i w_k}{y_1 - y_0} \rho_{ijk},$$

$$N_z = \sum_{i,j,k=0,1} \frac{(-1)^{k+1} u_i v_j}{z_1 - z_0} \rho_{ijk}.$$

Given a ray $\mathbf{p} = \mathbf{a} + t\mathbf{b}$, the intersection with the isosurface occurs when $\rho(\mathbf{p}) = \rho_{\text{iso}}$. We can convert this ray into coordinates defined by $(u_0, v_0, w_0)$: $\mathbf{p}_0 = \mathbf{a}_0 + t\mathbf{b}_0$ and a second ray defined by $\mathbf{p}_1 = \mathbf{a}_1 + t\mathbf{b}_1$. Here the rays are in the two coordinate systems (Figure 26.8):

$$\mathbf{a}_0 = (u_0^a, v_0^a, w_0^a) = \left(\frac{x_1 - x_a}{x_1 - x_0}, \frac{y_1 - y_a}{y_1 - y_0}, \frac{z_1 - z_a}{z_1 - z_0}\right),$$

and

$$\mathbf{b}_0 = (u_0^b, v_0^b, w_0^b) = \left(\frac{x_b}{x_1 - x_0}, \frac{y_b}{y_1 - y_0}, \frac{z_b}{z_1 - z_0}\right).$$

These equations are different because $\mathbf{a}_0$ is a location and $\mathbf{b}_0$ is a direction. The equations are similar for $\mathbf{a}_1$ and $\mathbf{b}_1$:

$$\mathbf{a}_1 = (u_1^a, v_1^a, w_1^a) = \left( \frac{x_a - x_0}{x_1 - x_0}, \frac{y_a - y_0}{y_1 - y_0}, \frac{z_a - z_0}{z_1 - z_0} \right),$$

and

$$\mathbf{b}_1 = (u_1^b, v_1^b, w_1^b) = \left( \frac{-x_b}{x_1 - x_0}, \frac{-y_b}{y_1 - y_0}, \frac{-z_b}{z_1 - z_0} \right).$$

Note that $t$ is the same for all three rays; it can be found by traversing the cells and doing a brute-force algebraic solution for $t$. The intersection with the isosurface $\rho(\mathbf{p}) = \rho_{\text{iso}}$ occurs when

$$\rho_{\text{iso}} = \sum_{i,j,k=0,1} \left(u_i^a + tu_i^b\right)\left(v_j^a + tv_j^b\right)\left(w_k^a + tw_k^b\right)\rho_{ijk}.$$

This can be simplified to a cubic polynomial in $t$:

$$At^3 + Bt^2 + Ct + D = 0,$$

where

$$\begin{aligned}
A &= \sum_{i,j,k=0,1} u_i^b v_j^b w_k^b \rho_{ijk}, \\
B &= \sum_{i,j,k=0,1} \left(u_i^a v_j^b w_k^b + u_i^b v_j^a w_k^b + u_i^b v_j^b w_k^a\right)\rho_{ijk}, \\
C &= \sum_{i,j,k=0,1} \left(u_i^b v_j^a w_k^a + u_i^a v_j^b w_k^a + u_i^a v_j^a w_k^b\right)\rho_{ijk}, \\
D &= -\rho_{\text{iso}} + \sum_{i,j,k=0,1} u_i^a v_j^a w_k^a \rho_{ijk}.
\end{aligned}$$

The solution to a cubic polynomial is discussed in *Cubic and Quartic Roots* (Schwarze, 1990). His code is available on the web in several *Graphics Gems* archive sites. Two modifications are needed to use it: linear solutions (his code assumes $A$ is non-zero), and the EQN_EPS parameter is set to 1.0e-30, which provided for maximum stability for large coefficients.

### 26.2.2 Direct Volume Rendering

Another way to create a picture of a 3D scalar field is to do a 3D random density plot using small opaque spheres. To avoid complications, the spheres can be made a constant color and, in effect, they are light emitters with no reflectance. Such a random density plot can be implemented directly using ray tracing and small spheres, or with 3D points using a traditional graphics API. As in 2D, we can take the limit as the sphere size goes to zero. This yields a 3D analog of the pseudocolor display and is usually called *direct volume rendering* (Levoy, 1988; Drebin, Carpenter, & Hanrahan, 1988; Sabella, 1988; Upson & Keeler, 1988).

There are two parameters that affect the appearance of a volume rendering: sphere color, and sphere density. These are controlled by a user-specified *transfer function*:

$$\begin{aligned} \text{color} &= c(\rho), \\ \text{number density} &= d(\rho). \end{aligned}$$

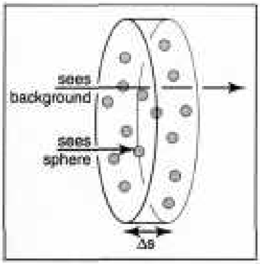


**Figure 26.10.** A thin slab filled with opaque spheres.

Here the *number density* is the number of spheres per unit volume. If we assume that the spheres have a small cross-sectional area $a$, and we consider a region along the line of sight that is of a small thickness $\Delta s$ such that no spheres appear to overlap (Figure 26.10), then the color is

$$L(s + \Delta s) = (1 - F)L(s) + Fc,$$

where $F$ is the fraction of the disk that is covered by spheres as seen from the viewing direction. Because the disk is very thin, we can ignore spheres visually overlapping, so this fraction is just the total cross-sectional area of the spheres divided by the area $A$ of the disk:

$$F = \frac{daA\,\Delta s}{A} = da\Delta s,$$

which yields

$$L(s + \Delta s) = (1 - da\,\Delta s)L(s) + da\Delta sc.$$

We can rearrange terms to give something like a definition of the derivative:

$$\frac{L(s + \Delta s) - L(s)}{\Delta s} = -daL(s) + dac.$$

If we take the limit $\Delta s \to 0$, we get a differential equation:

$$\frac{dL}{ds} = -daL(s) + dac.$$

For constant $d$ and $c$ this equation has the solution

$$L(s) = L(0)e^{-das} + c\left(1 - e^{-das}\right).$$

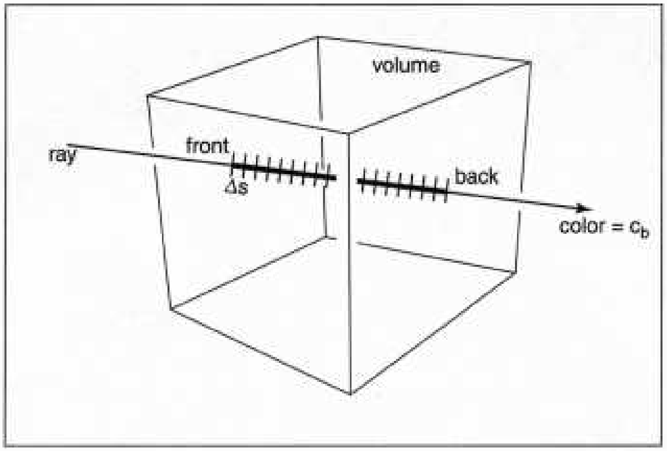


**Figure 26.11.** For direct volume rendering, we can take constant size steps along the ray and numerically integrate.

This would allow us to analytically compute color for constant density/color regions. However, in practice both $d$ and $c$ vary along the ray, and there is no analytic solution to the differential equation. So, in practice, we use a numerical technique. A simple way to proceed is to start at the back of the ray and incrementally step along the ray as shown in Figure 26.11.

We can apply the original equation for each $\Delta s$ slice:

$$L(s + \Delta s) = (1 - d(x, y, z)a\,\Delta s)L(s) + d(x, y, z)a\,\Delta s c(x, y, z).$$

In pseudocode, we initialize the color to the background color $c_b$ and then traverse the volume from back to front:

```
find volume entry and exit points $\mathbf{a}$ and $\mathbf{b}$
$L = c_b$
$\Delta s = $ distance($\mathbf{a}$, $\mathbf{b}$)
$\mathbf{p} = \mathbf{b}$
for $i = 1$ to $N$ do
    $\mathbf{p} = \mathbf{p} - \Delta s(\mathbf{b} - \mathbf{a})$
    $L = L + (1 - d(\mathbf{p})a\Delta s L + d(\mathbf{p})a\,\Delta s c(\mathbf{p})$
```

The step size $\Delta s$ will determine the quality of the integration. To reduce the number of variables, we can use a new density function $g(\mathbf{p}) = d(\mathbf{p})a$.

In some applications direct volume rendering is used to render something similar to surfaces. In these cases the transfer function on density is "on" or "off" and the gradient of the number density is used to get a surface normal for shading.

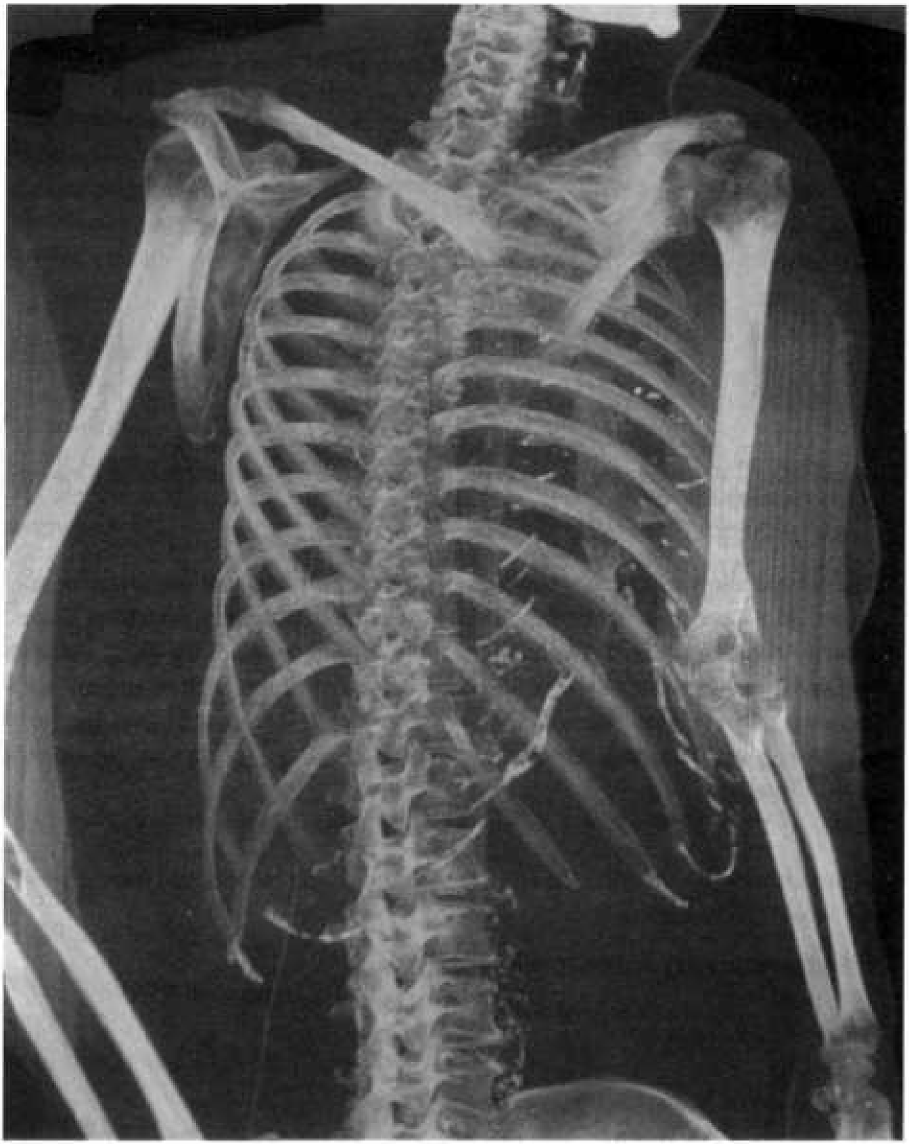


**Figure 26.12.** A maximum-intensity projection of the NIH/NIM Visible Female dataset. Each pixel contains a greyscale value that corresponds to the maximum density encountered along that ray. Image courtesy Steve Parker.

This can produce images of pseudosurfaces that are less sensitive to noise than traditional isosurfacing.

Another way to do volume rendering is *maximum-intensity projection*. Here, we set each pixel to the maximum density value encountered along a ray. This turns the ray integration into a search along the ray which is more efficient. Figure 26.12 shows an image generated using maximum-intensity projection.

## Frequently Asked Questions

### • What is the best transfer function for direct volume rendering?

The answer depends highly on the application and the characteristics of the data. Some empirical tests have been run and can be found in (Pfister et al., 2001). Various optical models used in direct volume rendering are described in (Max, 1995).

### • What do I do to visualize vector or tensor data?

Vector data is often visualized using streamlines, arrows, and *line-integral convolution* (LIC). Such techniques are surveyed in (Interrante & Grosch, 1997). Tensor data is more problematic. Even simple diffusion tensor data is hard to visualize effectively because you just run out of display dimensions for mapping of data dimensions. See (Kindlmann, Weinstein, & Hart, 2000).

### • How do I interactively view a volume by changing isovalues?

One way is to use ray tracing on a parallel machine. The other is to use polygonal isosurfacing with a preprocess that helps search for cells containing an isosurface. That search can be implemented using the data structure in (Livnat, Shen, & Johnson, 1996).

### • My volume data is unstructured tetrahedra. How do I do isosurfacing or direct volume rendering?

Isosurfacing can still be done in a polygonal fashion, but there are fewer cases to preprocess. Ray tracing can also be used for isosurfacing or direct volume rendering, but the traversal algorithm must progress through the unstructured data either using neighbor pointers (Garrity, 1990) or by adding cells to an efficiency structure (Parker, Parker, et al., 1999).

- What is "splatting" for direct volume rendering?

Splatting refers to projecting semitransparent voxels onto the screen using some sort of painters' algorithm (Laur & Hanrahan, 1991).

## Exercises

1. If we have a tetrahedral data element with densities at each of the four vertices, how many "cases" are there for polygonal isosurfaces?
2. Suppose we have $n^3$ data elements in a volume. If the densities in the volume are "well behaved," approximately how many cells will contain an isosurface for a particular isovalue?
3. Should we add shadowing to direct volume rendering? Why or why not?

# Index

A/D converter, 73
adjoint matrix, 127
aerial perspective, 509
aliasing, 67, 72, 74, 75, 98, 99, 112–114, 117
ambient shading, 193
amodal completion, 508
analog-to-digital converter, 73
angle, 20
  cosine, 21
  sine, 21
animation, 2
anti-umbra, 231
antialiasing, 67, 98, 99
aperture problem, 493
API, 1, 3
apparent motion, 493
application program interface, 3
arc length, 305
arc-length parameterized curve, 37
array
  padded, 274
  tiling, 274
artistic shading, 197
assert(), 11
associativity, 80, 83
attribute variables, 391
average, 281

B-spline
  control points, 335
  curve, 334, 335
  filter, 90, 99
  function
    Fourier transform, 110
  interpolation, 342
  non-uniform, 340
  NURBS, 344
  repeated knots, 342
  uniform cubic, 339
  uniform quadratic, 337
Bézier curves, 327
backface elimination, 265
barycentric coordinates, 43, 44, 63, 300
basic execution model, 390
basis, 24
  function, 310
  matrix, 313
  vectors, 24
Bernstein basis polynomials, 329

- bidirectional reflectance distribution function, 458, 561
  - Lambertian 459
- bijection, 17
- binary space partitioning (BSP) tree, 177, 236
- blending function, 310, 316
- blurring, 96
- bounding box, 219
- bounding volume, 265
  - hierarchy, 222
- box filter, 67, 78, 83, 89, 99, 100
- box function, 78, 83
  - Fourier transform, 109
- BRDF, 458, 561
  - Lambertian, 459
- bricking, 274
- BSP tree, 177, 236
  - ray tracing, 227
- built-in variables, 391
- bump mapping, 392

- CAD/CAM, 3
- callbacks, 4
- camera
  - digital, 68, 466
- candela, 462, 489
- canonical, 160
- canonical form, 310, 312
- canonical view volume, 160
- Cardinal spline, 325
- cardinality, 48
- Cartesian coordinates, 24
- Cartesian product, 16
- Catmull-Rom filter, 90, 91
- Catmull-Rom splines, 325
- CCD, 68
- cellular automata, 371
- chromatic adaptation, 537
- chromaticity, 469
- CIE scotopic sensitivity function, 472
- circle, 20
- clipping, 175, 259
  - line segment, 264
  - triangle, 264
- closed interval, 17
- CMOS, 68
- coefficients, 310
- cofactor matrix, 127
- cofactors, 125
- color, 465, 486
  - 24 bit, 55
  - additive, 55
  - constancy, 488
  - hue, 475
  - opponent, 474
  - RGB, 51, 54, 465
  - saturation, 475
  - space
    - CIE, 487
    - HSV, 488
    - RGB, 472, 473, 487
  - trichromatic, 466
  - value, 475
  - state, 383
- column vector, 123
- commutativity, 80, 81, 83
- compositing, 56
- compression, 68
  - lossless, 68
  - lossy, 68
  - sigmoidal, 537
- cones, 486
- conformational parameters, 365
- constant
  - semi-saturation, 535, 536
- constructive solid geometry, 229
- continuity, 92, 308
- continuous motion, 494
- control driven programming, 405
- control point, 311, 335
- convex hull, 333
- convolution, 75, 76, 108, 112
  - and Fourier transform, 108
  - by separable filter, 94, 95
  - continuous, 82–84
  - discrete, 76–81
  - discrete-continuous, 85, 86
  - of an image, 95
  - properties, 79, 80, 83
  - smoothing, 76
  - two-dimensional, 87, 88
- coordinate frame, 29

coordinates
- global, 272
- local, 272
- polar, 21

correspondence problem, 500
cosine, 21
Cox-de Boor recurrence, 341
Cramer's rule, 121, 129, 207
cross product, 26
crossed disparity, 501
CRT, 52
CSG (constructive solid geometry), 229
cube map, 254
cubic polynomials, 315
cues
- dynamic occlusion, 503

culling, 265
- backface elimination, 265

curve, 30, 301
- arc-length parameterized, 37
- B-spline, 334, 335
- Bézier, 327
- implicit, 30
- parametric, 36, 40
- tangent, 31

cycle (viewing), 178

D/A converter, 73
data bus, 384
data sink, 391
data source, 391
de Casteljau algorithm, 333
debugging, 12
degree of continuity, 92
delta function, 84, 111
density representation, 529
dependency scheme, 319
depth buffer, 186
- integer, 186

depth of field, 233
derivative, 31
determinant, 119, 125
- Laplace's expansion, 125

diagonal matrix, 124
diagonalization, 130
dielectric, 213
difference equation, 368, 369
- explicit schemes, 369
- implicit schemes, 369

differential geometry, 308
diffuse reflection, 459
digital audio, 72
digital camera, 68
digital-to-analog converter, 73
Dirac delta function, 84, 111
direct lighting, 554
direct volume rendering, 590
Direct3D, 4
directional hemispherical reflectance, 459
directional light, 192
discontinuous motion, 493
discrete impulse, 80, 81
discriminant, 19
disparity, 500
displacement map, 253
display
- CRT, 52
- interlaced, 495
- LCD, 52
- raster, 51
- resolution, 51

display list, 267, 385
distal stimulus, 478
distance
- signed, 35

distribution ray tracing, 229
domain, 16
dot product, 25
double, 9
drop shadow, 97

efficiency, 8
eigenvalues, 129
eigenvectors, 129
environment maps, 254
event-driven programming, 408
expressive parameters, 365
expected value, 285, 288
explicit curve, 302
eye
- cones, 486
- retina, 466
- rods 486

familiar size, 512
Fechner's law, 484
field radiance, 456, 581
field-of-view, 173, 490
filter
- B-spline, 90, 99
- bilateral, 530
- box, 67, 78, 83, 84, 89, 99, 100
- Catmull-Rom, 90, 91
- degree of continuity, 92
- edge-preserving smoothing, 531
- Gaussian, 90, 94, 99
- identity, 80, 84
- impulse response, 91
- interpolating, 90, 91
- Mitchell-Netravali, 91
- overshoot, 92
- resampling, 101, 102
- ringing, 92
- ripple free, 92
- separable, 93, 103
- separable tent, 93
- shifted, 81
- tent, 83, 84, 89, 93, 99
- trilateral, 530

filtering, 78
flexors, 364
flicker fusion, 494
float, 9
focus of expansion, 514
forward kinematics, 362
Fourier series, 105
Fourier transform, 105–110
fovea, 490
fragment program, 388
fragment shader, 388
frame of reference, 29
- allocentric, 497
- egocentric, 497
- exocentric, 497

frame rate, 380, 495
frame update rate, 495
free-form curve, 302
frequency-based operators, 529
Fresnel equations, 214, 562
full multigrid method, 531
function, 15
- bijection, 17
- domain, 16
- inverse, 17
- inversion, 292
- range, 16

gamma, 53, 472
gamma correction, 54
Gaussian
- Fourier transform, 110
- filter, 90, 94, 99

gaze direction, 164
geometric continuity, 309
geometric surface, 381
gif, 68
global coordinates, 28, 272
global illumination, 547
Gouraud color interpolation, 63
GPU, 389
- general purpose computation, 398

gradient, 31
gradient domain
- compression, 531
- operator, 531

graphics hardware, 379
- processor, 389

Graphics Processing Unit, 389
graphics state, 383
GUI
- API, 411
- elements, 403, 419
- events, 412
  - Callback, 419
  - paint, 415
  - redraw, 415
  - RedrawRoutine(), 416
  - service registration, 419, 434
  - service routines, 416, 419, 420
  - timer, 413

haloing, 534
hard shadow, 231
Hermite cubic, 324
- polynomials, 324

Hermite form, 315

hidden surface elimination, 177
    BSP tree, 177
    ray tracing, 203
    z-buffer, 186
high dynamic range images, 521
highlight, 194
histogram adjustment, 542
histogram equalization, 542
homogeneous coordinates, 152, 169
homogenization, 169
horizon ratio, 513
hue, 469, 475, 488
hypotenuse, 20

IBR, 577
identity filter
    continuous, 84
    discrete, 80
identity matrix, 123
IEEE floating point, 5
illuminance, 525
illumination
    global, 547
image, 9, 71
    blurring, 96
    compression, 68
    convolution, 95
    drop shadow, 97
    Gaussian-blurred, 539
    high dynamic range, 521
    processing, 2
    reconstruction, 99–102
    resampling, 99, 100, 102
    resmpling, 101
    sharpening, 97
    unsharp mask, 97
image-based modeling, 580
image-based rendering, 577
immediate mode rendering, 384
implicit curve, 30, 302
    gradient, 31
implicit function, 302
importance sampling, 289
impulse
    discrete, 80, 81
    response, 91
    train, 111
include guards, 11
incremental computation, 59
independent random variable, 286
indirect lighting, 547
information visualization, 3, 583
infrared light, 462
inline functions, 10
instancing, 216
integral geometry, 281
integration, 280
    Monte Carlo, 288, 552
    quasi-Monte Carlo, 290
intensity, 463
interpolate, 308
interpolating filter, 90, 91
interpolation, 42
    B-spline, 342
    linear, 42
    normal vector, 197
intersection
    ray–bounding box, 219
    ray-implicit, 205
    ray-parametric, 206
    ray-sphere, 205
    ray-triangle, 206
interval, 17
    closed, 17
    open, 17
    set operations, 17
inverse Fourier transform, 106
inverse function, 17
inverse kinematics, 363
inverse matrix, 124, 128
inverse optics, 496
inversion
    function, 292
irradiance, 454
isosurface, 585

Jacobian, 363
jaggies, 98
Java, 4
jittering, 231
jpeg, 69

kernel, 398
kinetic depth effect, 502

- knot, 318
  - vector, 319

- Lagrange form, 317
- Lambert's cosine law, 192
- Lambertian, 191
  - BRDF, 459
- Law of Large Numbers, 288
- LCD, 52
- level-of-detail (LOD), 5
- light
  - directional, 192
  - infrared, 462
  - ultraviolet, 462
- light field, 578
- lighting, 265
  - direct, 554
  - indirect, 547
  - two-sided, 193
- lightness, 475, 481, 487, 531
  - constancy, 481
- limit curve, 331
- line
  - drawing, 57
    - incremental, 59
  - implicit, 33
  - normal coordinates, 283
  - parametric, 37, 40, 61
  - segment, 41
  - slope, 33
  - slope-intercept form, 33
  - vector form, 40
- linear independence, 24
- linear interpolation, 42
- linear perspective, 503
- linear scaling, 523, 534
- linear system
  - solution, 129
    - Cramer's rule, 121
- local coordinates, 29, 272
- location, 23
- LOD (level-of-detail), 5
- logarithm, 18
  - natural, 18
- luminaire, 547
  - non-diffuse, 558
- luminance, 462, 524
  - scotopic, 472, 543
- luminance distribution, 485
- luminance mapping function, 542
- luminous efficiency function, 462

- mappings, 15
- marching cubes, 586
- material
  - dielectric, 213
- matrix, 121
  - adjoint, 127
  - cofactor, 127
  - cofactors, 125
  - determinant, 119
  - diagonal, 124
  - diagonal form, 130
  - identity, 123
  - inverse, 124, 128
  - inversion using SVD, 154
  - multiplication, 122
  - product with scaler, 122
  - projection, 172
  - square, 122
  - stack, 273
  - symmetric, 123
  - transpose, 123
- maximum-intensity projection, 593
- measure
  - 2D lines, 282
  - 3D lines, 283
  - zero measure sets, 280
- mesh, 248
- metamer, 467, 468, 486, 487
- Metropolis sampling, 295
- microfacets, 562
- midpoint algorithm, 57
- MIP-mapping, 256
- Mitchell-Netravali filter, 91
- modeling, 2, 4
- modelview-controller, 421, 422, 423
  - component, 421
  - framework, 422
  - model, 422
  - view, 422, 423
- moiré pattern, 75, 98, 99, 103, 230

monitor
- gamma, 53
- phosphors, 472
- tristimulus values, 473
- white point, 472

Monte Carlo integration, 288, 552
- importance sampling, 289
- stratified sampling, 289

morph, 365
motion blur, 234
motion capture, 72, 366
- electromagnetic, 366
- optical, 366

motion parallax, 502
moving average, 76
mutual illumination, 547

nanometer, 452
natural cubic, 323
- splines, 323

natural logarithm, 18
non-photorealistic rendering, 197
non-uniform B-Spline, 340
normal coordinates, 283
normal direction, 383
normal interpolation, 197
normal vector, 31, 149
- at vertices, 193

number density, 590
NURBS, 344
Nyquist frequency, 113, 114
Nyquist-Shannon sampling theorem, 114

object recognition, 509
occlusion cues, 503
ocularmotor cue
- accomodation, 498
- convergence, 498
- vergence, 498

offset, 23
open interval, 17
OpenGL, 4
operator, 10
- frequency-based, 529
- gradient-domain, 531
- spatial, 532
- tone reproduction, 542

opponent colors, 474
optic flow, 501
origin, 23
orthogonal, 24
orthographic projection, 164
orthographic view volume, 162
orthonormal, 24
- basis, 27

overshoot, 92

padded array, 274
painter's algorithm, 178
parallelepiped
- volume, 119

parallelogram
- area, 119, 125

parallelogram rule, 23
parameterization, 304
- natural, 304

parametric
- continuity, 309
- curve, 36, 40, 303
- function, 302
- line, 40
- surface, 41

parametrization
- arc-length, 305

particle tracing, 549
path tracing, 551
penumbra, 231
persistent application state, 422
perspective
- projection, 166
- three point, 166

Phong exponent, 196
Phong normal interpolation, 197
Phong shading, 194
photometry, 451, 462
photon, 452
photoreceptors, 486, 524
physics-based animation, 367
picking, 201, 236
pictorial depth cues, 503
pixel, 51
plane, implicit, 38
point, 23
polar coordinates, 21

polygon
ray intersection, 208
polynomial
Hermite cubic, 324
position, 23
power, 453
ppm, 69
primitives, 380, 381
probability, 284
density function, 284
expected value, 285
procedural curve, 302
programming models
main(), 404
control-driven programming, 404
event-driven programming, 404
external control model, 404
internal control model, 404
MainEventLoop(), 408
projection
orthographic, 164
projection matrix, 172
property
global, 308
local, 308
proximal stimulus, 478
pseudocolor plot, 584
psychophysics, 485
Pythagorean theorem, 20

quadratic equation, 19, 205
quadric curve, 36
quasi-Monte Carlo integration, 290

radiance, 455, 578
field, 456, 581
surface, 456, 581
radiant exitance, 455
radiometry, 451
radiosity, 463, 548
random point density plot, 584
random sampling, 230
random variable, 286
independent identically distributed, 288
range, 16
raster display, 51
rasterization
triangle, 63
ray, 41
shadow, 211
ray tracing, 201, 209
distribution, 229
instancing, 216
object-oriented, 210
re-parameterization, 304
real-time graphics, 380
reconstruction, 71, 85, 86, 99–102, 112–114, 117
artifact, 73, 74
reflect transform, 140
reflectance, 525
directional hemispherical, 459
reflection
Lambertian, 191
specular, 212
total internal, 214
refraction, 216
refresh rate, 495
regular sampling, 230
rejection method, 294
relative size, 512
rendering, 2
non-photorealistic, 197
rendering equation, 461
renormalization, 103
resampling, 99–102, 104, 117
separable, 104
resampling filter, 101, 102
resolution, 51
retina, 466
retroreflection, 575
RGB color, 9, 51, 54, 465
right-handed coordinates, 27
rigid body transforms, 153
rigid skinning, 364
ringing, 92
ripple free, 92
rods, 486
rotation transform, 138
row vector, 123

saccades, 490
sampling, 71
  aliasing, 72, 74, 75
  jittering, 231
  random, 230
  regular, 230
  sample rate, 73, 75
  stratified, 231
  undersampling, 73, 74
sampling theory, 104
saturation, 475, 487
scalar product, 25
scale selection mechanism, 540
scale transform, 136
scaling
  linear, 534
scene graph data structure, 383
Schlick approximation, 214
scotopic luminance, 543
semi-saturation constant, 535, 536
separable filter, 93, 103, 104
separable tent filter, 93
set, 16
  Cartesian product, 16
  operations
    on intervals, 17
shading, 191, 265
  ambient, 193
  artistic, 197
  diffuse, 191
  Lambertian, 191
  Phong, 194
shadow
  anti-umbra, 231
  attached, 505
  detached, 505
  hard, 231
  penumbra, 231
  soft, 231
  umbra, 231
shadow map, 255
shadow ray, 211
shape constancy, 514
shared-point scheme, 319
sharpening, 97
shear transform, 137
shifted filter, 81
sigmoid, 534, 535
sigmoidal compression, 537
signal processing, 71
signed distance, 35
silhouettes, 197
sine, 21
single instruction, multiple data (SIMD) processor, 389
singular value decomposition (SVD), 131, 144
  for matrix inversion, 154
singular values, 131
site, 308
size constancy, 514
slant, 497, 505
slope, 31
smooth skinning, 364
smoothing, 76
soft shadow, 231
software engineering, 8
solid texture
  turbulence, 245
spatial operator, 532
spectral distribution, 485
spectral energy, 453
spectral power, 454
spectrum, 485
specular highlights, 525
specular reflection, 212
sphere
  implicit form, 205
  normal vector, 206
  vector form, 205
splatting, 594
spline, 307
  curve, 307
spot light, 255
square matrix, 122
state attributes, 382
steady state, 453
step function, 78, 92
Steven's law, 484
stratified sampling, 231, 289
stream processor, 389
structure-from-motion, 502

- subdivision scheme, 331
- subjective contour, 483
- support, 77
- surface radiance, 456, 581
- surface
  - implicit, 38
  - parametric, 41
- SVD, 131, 144
- symmetric matrix, 123

- tangent, 31
  - vector, 31
- temporal aliasing, 495
- tent filter, 83, 89, 93, 99
- tent function, 83, 84
  - Fourier transform, 110
- texture
  - marble, 245
- texture mapping, 239
  - MIP-mapping, 256
- three-point perspective, 166
- tiff, 69
- tiling, 274
- tilt, 497, 505
- TIN, 248
- tone reproduction, 522
- tone reproduction operator, 542
- tonemapping, 522
- total internal reflection, 214
- transfer function, 590
- transformation
  - matrix, 135
  - reflection, 140
  - rigid body, 153
  - rotate, 138
  - scale, 136
  - shear, 137
- transport equation, 551, 554
- transpose, 123
- triangle, 43
  - barycentric coordinates, 43
  - fans, 266
  - mesh, 248, 381
    - winged-edge data structure, 270
  - random point selection, 300
  - ray intersection, 206
  - rasterization, 63
  - strips, 266
- triangular irregular network (TIN), 248
- trichromatic color theory, 466
- tristimulus values, 469
- turbulence function, 245
- two-sided lighting, 193

- ultraviolet light, 462
- umbra, 231
- uncrossed disparity, 501
- undersampling, 73, 74, 113
- uniform cubic B-spline, 339
- uniform quadratic B-spline, 337
- uniform variables, 391
- unit vector, 23
- units
  - extensive, 453
  - intensive, 453
- unsharp mask, 97
- user-defined variables, 391

- value, 475
- variable
  - attribute, 391
  - built-in, 391
  - uniform, 391
  - user-defined, 391
- variance, 287
- variation diminishing property, 330
- varying data, 391
- vector, 23
  - addition, 23
  - basis, 24
  - cross product, 26
  - dot product, 25
  - equality, 23
  - length, 23, 24
  - linear independence, 24
  - normal, 31, 38, 149
  - orthogonal, 24
  - orthonormal, 24
  - projection, 25
  - subtraction, 24
  - unit, 23
  - zero, 23

vertex
- buffer objects, 385
- function calls, 382
- normal, 193
- program, 388
- shader, 388

video games, 3

view volume
- orthographic, 162

view-up vector, 164

viewing, 164
- field-of-view, 173

virtual reality, 2

vision
- foveal, 491
- high acuity, 489
- phottopic, 489
- scotopic, 489

visual acuity, 480

visual attention, 516

visual cue, 496, 498
- disparity, 496
- motion, 496
- ocularmotor, 496
- pictorial, 496

visualization, 2, 583
- information, 583

volume rendering, 585, 590
- maximum-intensity projection, 593
- transfer function, 590

wagon wheel illusion, 495

Weber's law, 484

winged-edge data structure, 270

z-buffer, 186
- integer, 186

zero measure sets, 280

zero vector, 23