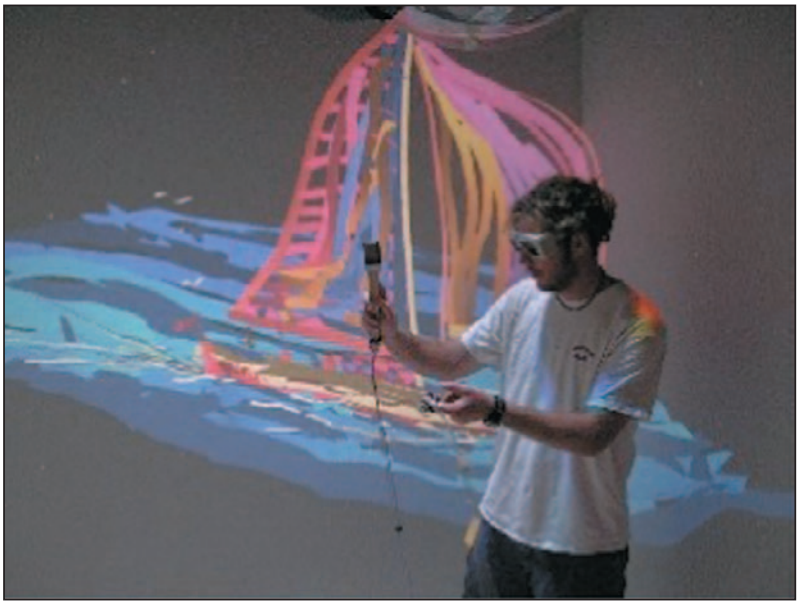
COMPUTER GRAPHICS

PRINCIPLES AND PRACTICE

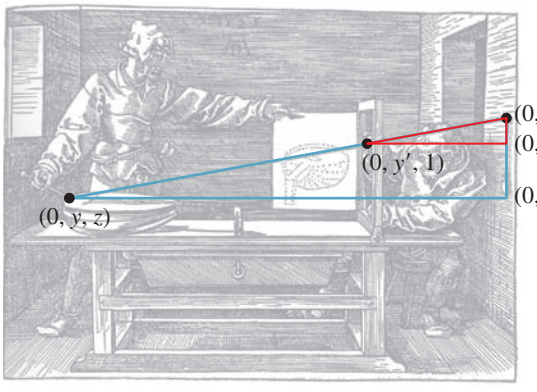
THIRD EDITION

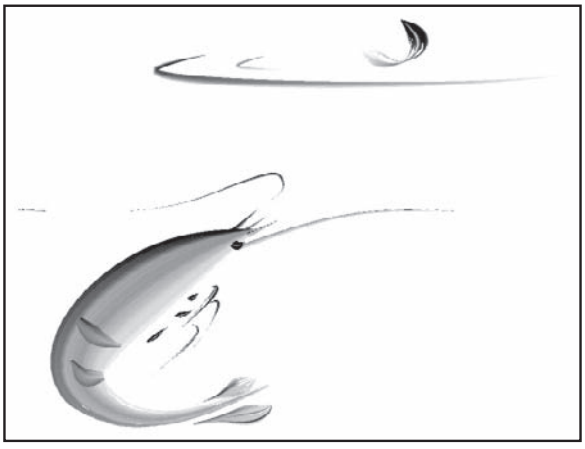
JOHN F. HUGHES · ANDRIES VAN DAM · MORGAN MCGUIRE
DAVID F. SKLAR · JAMES D. FOLEY · STEVEN K. FEINER · KURT AKELEY

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

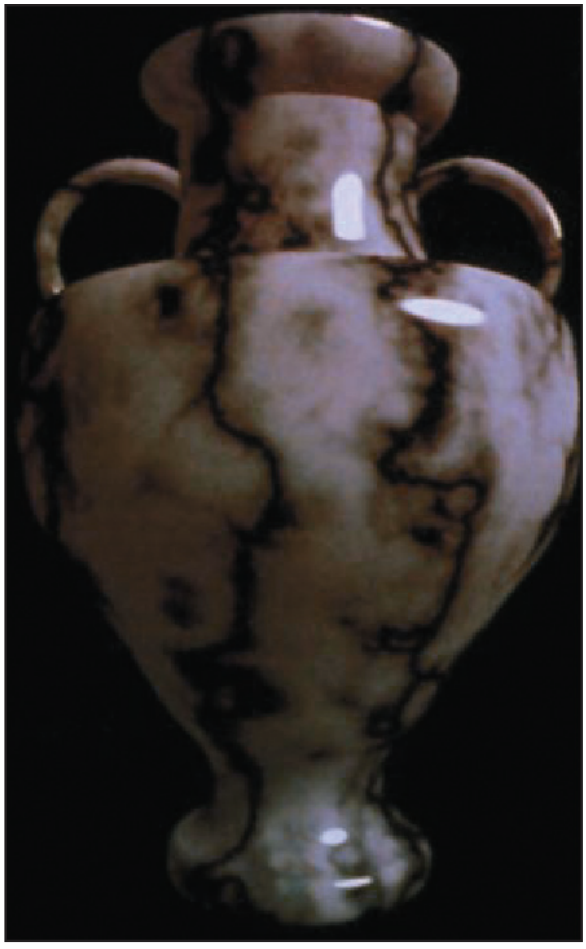


Courtesy of Daniel Keefe, University of Minnesota.

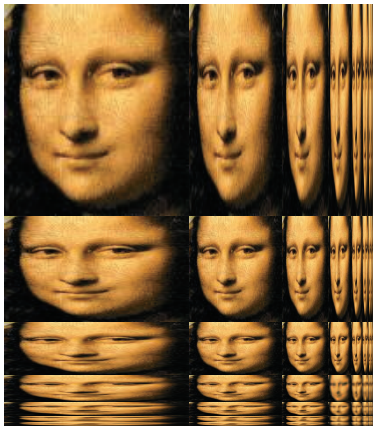




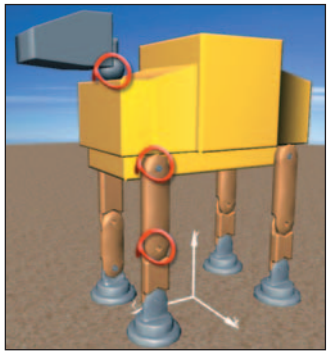
Courtesy of Steve Strassmann. © 1986 ACM, Inc. Reprinted by permission.



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



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



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

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

Computer Graphics

Third Edition

This page intentionally left blank

Computer Graphics

Principles and Practice

Third Edition

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

Addison-Wesley

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

Library of Congress Cataloging-in-Publication Data

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

2012045569

ISBN-13: 978-0-321-39952-6
ISBN-10: 0-321-39952-8
2 17

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

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

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

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

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

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

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

This page intentionally left blank

Contents at a Glance

Contents

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

This page intentionally left blank

Preface

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

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

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

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

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

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

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

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

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

Historical Approaches

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

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

Pedagogy

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

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

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

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

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

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

Current Practice

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

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

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

Principles

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

Prerequisites

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

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

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

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

Paths through This Book

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

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

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

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

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

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

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

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

Differences from the Previous Edition

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

Website

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

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

Acknowledgments

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

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

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

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

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

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

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

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

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

For the Student

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

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

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

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

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

For the Teacher

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

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

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

Discussion and Further Reading

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

Note on the Second Printing

This second printing corrects several errors. We are particularly grateful to Martin Magnusson, Alessandro Gentilini, and Daniel Shelpov, who each reported multiple errors and suggested corrections, and especially Davide Cavignino, who single-handedly identified and corrected more mistakes than all other readers combined.

About the Authors

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

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

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

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

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

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

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

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

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

This page intentionally left blank

Chapter 1

Introduction

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

1.1 An Introduction to Computer Graphics

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.1.1 The World of Computer Graphics

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

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

1.1.2 Current and Future Application Areas

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

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

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

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

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

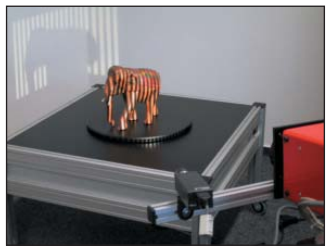


Figure 1.2: A scanner that projects stripes on a model that is slowly rotated on a turntable. The camera records the pattern of stripes in many positions to determine the object's shape. (Courtesy of Polygon Technology, GMBH).

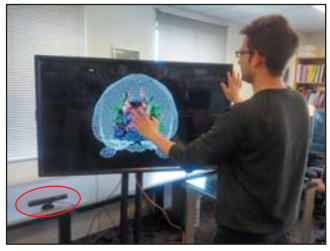


Figure 1.3: Microsoft's Kinect interface can sense the user's position and gestures, allowing a scientist to adjust the view of his data by "body language", without a mouse or keyboard. (Data view courtesy of David Laidlaw; image courtesy of Emanuel Zgraggen.)

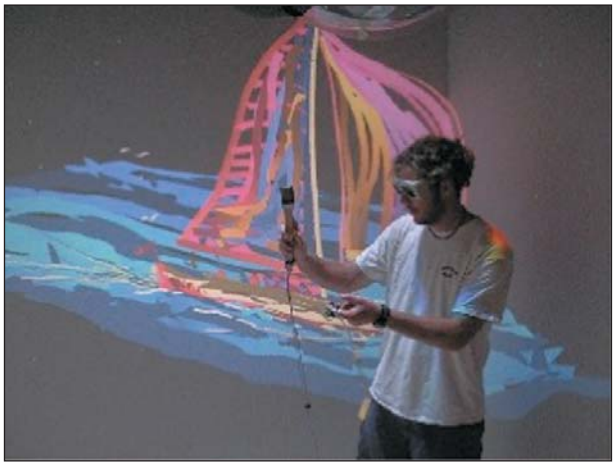


Figure 1.4: An artist stands in a Cave (a room whose walls are displays) and places paint strokes in 3D. The displays are synchronized with stereo glasses to display imagery so that it appears to float in midair in the room. Head-tracking technology allows the software to produce imagery that is correct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

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

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

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

1.1.3 User-Interface Considerations

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

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

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

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

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

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

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

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

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

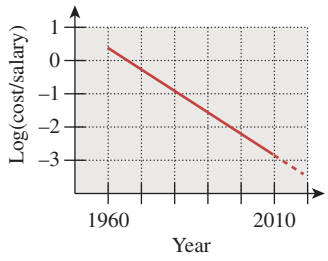


Figure 1.6: The log of the ratio between the cost of a computer and the salary of a person using the computer (roughly amortized for multiuser systems), plotted against the year.

1.2 A Brief History

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

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

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

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

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

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

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

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

1.3 An Illuminating Example

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

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

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

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

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

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

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

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

1.4 Goals, Resources, and Appropriate Abstractions

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

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

We summarize the preceding discussion in a principle:

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

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

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

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

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

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

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

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

1.4.1 Deep Understanding versus Common Practice

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

1.5 Some Numbers and Orders of Magnitude in Graphics

Because we will start our study of graphics with a discussion of light, it's useful to have a few rough figures characterizing the light encountered in ordinary scenes. Visible light, for instance, has a wavelength between approximately 400 and 700 nanometers (a nanometer is 1.0×10^{-9} m). A human hair has a diameter of about 1.0×10^{-4} m, so it's about 100 to 200 wavelengths thick, which helps give a human scale to the phenomena we're discussing.

1.5.1 Light Energy and Photon Arrival Rates

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

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

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

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

Using 650 nm as a typical photon wavelength, we get

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

as the energy of a typical photon.

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

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

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

1.5.2 Display Characteristics and Resolution of the Eye

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

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

1.5.3 Digital Camera Characteristics

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

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

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

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

1.5.4 Processing Demands of Complex Applications

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

1.6 The Graphics Pipeline

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

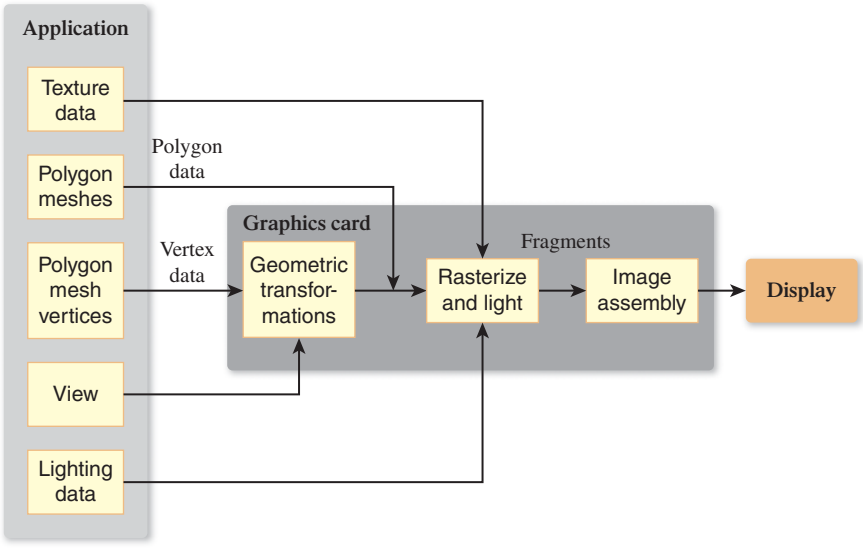


Figure 1.8: The graphics pipeline, version 1.

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

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

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

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

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

1.6.1 Texture Mapping and Approximation

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

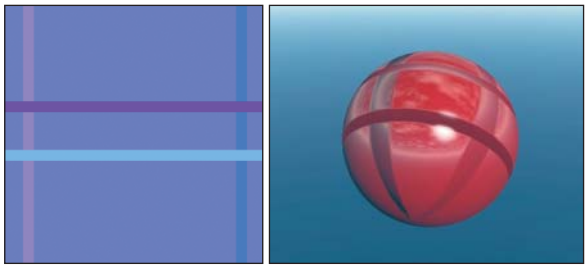


Figure 1.9: (a) The image at left depicts a normal map. Each image point has x- and y-coordinates that correspond to the latitude and longitude of a point on the sphere. The RGB color triple stored at each point determines how much to tilt the normal vector at the corresponding point of the sphere. The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

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

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

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

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

1.6.2 The More Detailed Graphics Pipeline

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

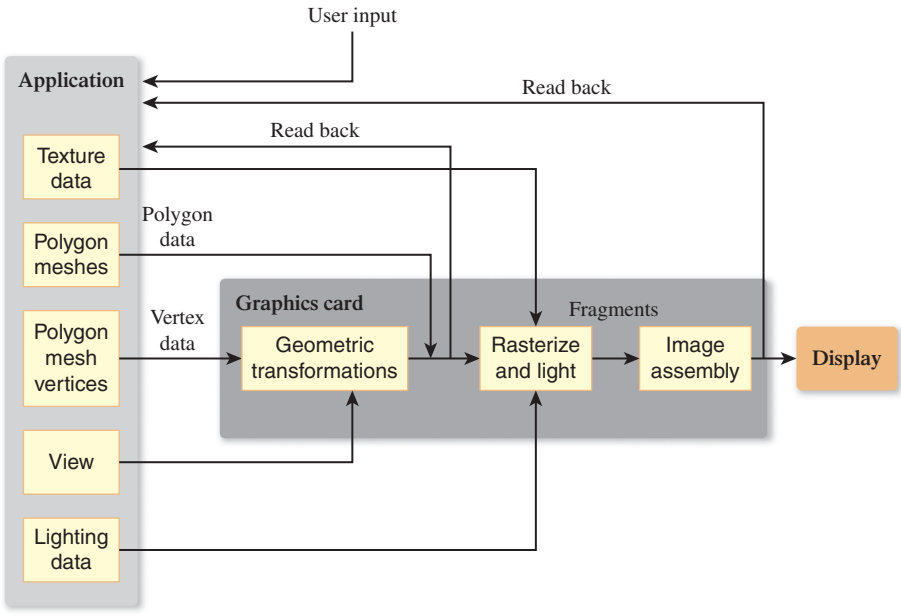


Figure 1.10: In this depiction of a larger graphics pipeline, the application program performs some work (e.g., animation) to determine the geometry to be displayed; this geometric description is handed to the graphics pipelines; the resultant image is displayed. At the same time, user input, in response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

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

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

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

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

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

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

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

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

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

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

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

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

1.7 Relationship of Graphics to Art, Design, and Perception

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

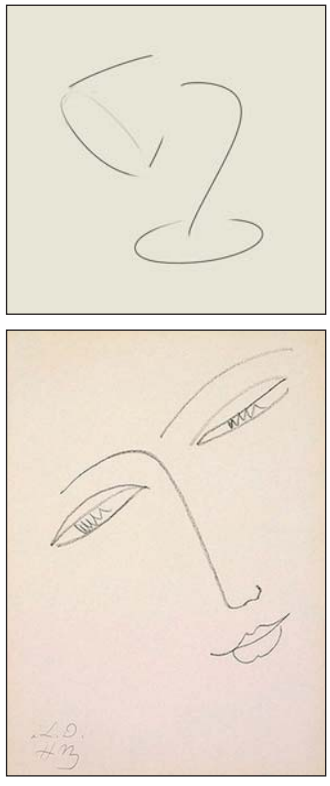


Figure 1.11: The lamp, courtesy of Jack Hughes, has just five strokes. Matisse's "Face of a Woman" depicts both shape and mood in just 13 strokes.

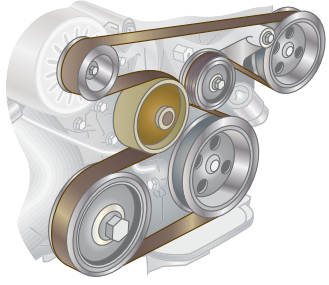


Figure 1.12: A repair manual shows details where needed, but omits unnecessary material.

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

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

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

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

1.8 Basic Graphics Systems

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

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

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

1.8.1 Graphics Data

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

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

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

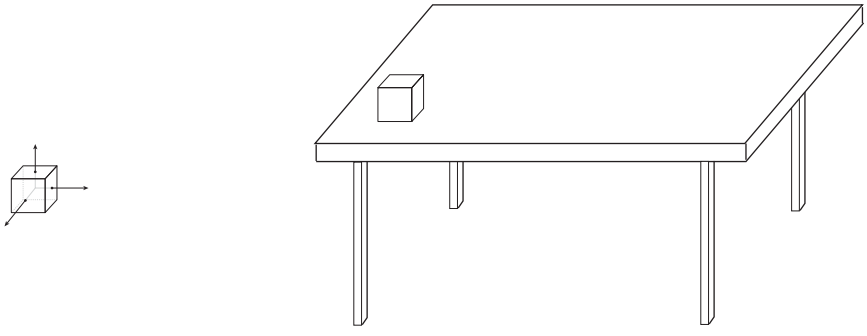


Figure 1.14: On the left, a die is centered on its own axes in modeling coordinates. The same die is placed in the world (on the right) by adding 6.5 to each y-coordinate (the y-direction points "up") to get world coordinates for the die.

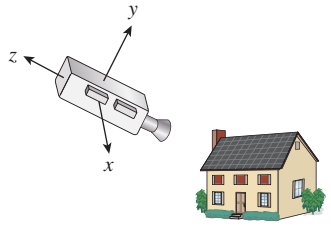


Figure 1.15: The virtual camera looks at a scene from a specified location, and with some orientation or attitude. We can create a coordinate system whose origin is at the center of the camera, whose z-axis points opposite the view direction, and whose x- and y-axes point to the right and to the top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

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

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

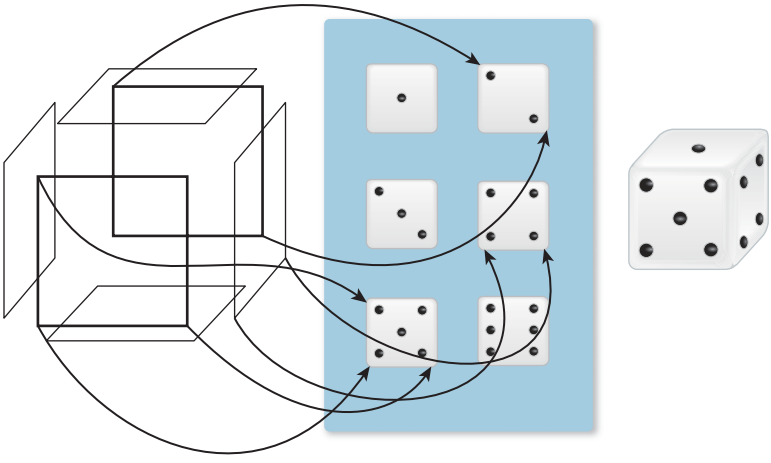


Figure 1.16: The vertices of each of the six faces of the die (shown in an exploded view) are assigned texture coordinates (a few are indicated by the arrows in the diagram); the texture image is then used to determine the appearance of each face of the die, as if the texture were a rubber sheet stretched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

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

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

1.9 Polygon Drawing As a Black Box

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

1.10 Interaction in Graphics Systems

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

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

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

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

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

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

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

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

1.11 Different Kinds of Graphics Applications

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

The following are some of the relevant criteria.

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

Less critical, but still important, are the following.

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

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

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

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

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

1.12 Different Kinds of Graphics Packages

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

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

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

1.13 Building Blocks for Realistic Rendering: A Brief Overview

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

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

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

1.13.1 Light

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

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

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

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

1.13.2 Objects and Materials

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

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

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

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

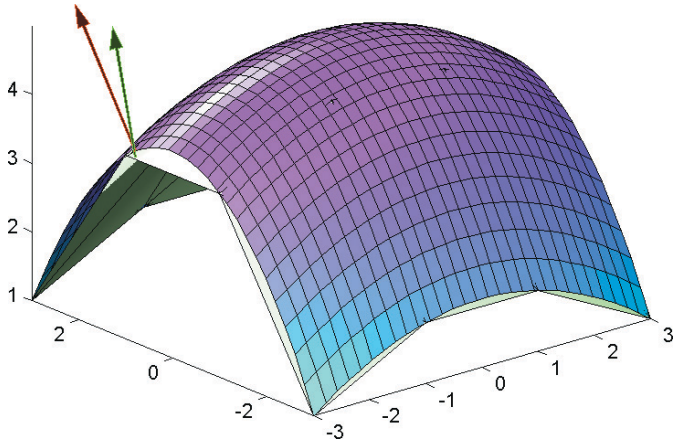


Figure 1.17: A semitransparent smooth surface and a normal vector to it (red), with a polygonal approximation (white) within, and a normal vector to the polygonal mesh at a corresponding point (green).

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

For other surfaces, incoming light scatters in many directions.

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

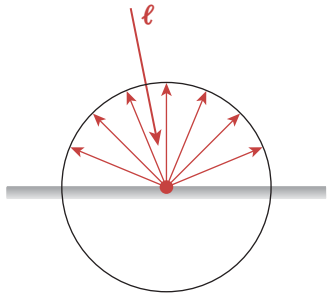


Figure 1.18: Light arrives traveling in direction $\boldsymbol{\ell}$; it's reflected in all directions. For a source of fixed brightness, the intensity of the reflected light is greatest when $\boldsymbol{\ell}$ is perpendicular to the surface.

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

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

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

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

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

1.13.3 Light Capture

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

1.13.4 Image Display

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

1.13.5 The Human Visual System

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

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

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

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

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

1.13.6 Mathematics

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

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

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

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

1.13.7 Integration and Sampling

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


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

1.14 Learning Computer Graphics

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

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

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

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

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

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

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

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

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

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

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

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

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

This page intentionally left blank

Chapter 2

Introduction to 2D Graphics Using WPF

2.1 Introduction

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

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

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

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

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

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

2.2 Overview of the 2D Graphics Pipeline

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

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

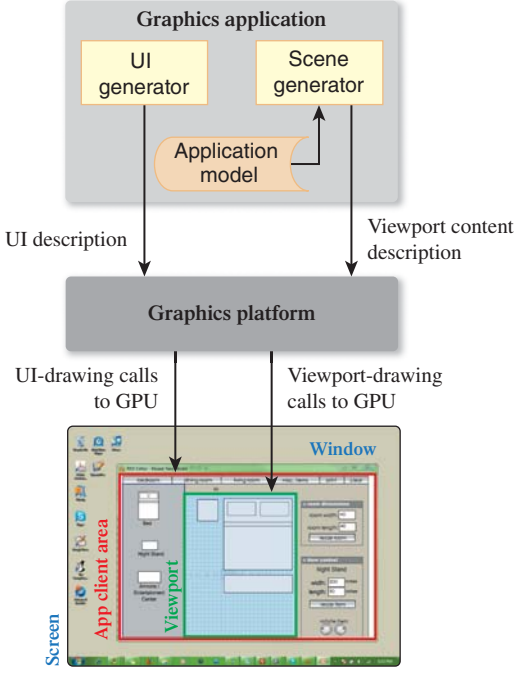


Figure 2.1: Graphics platform as an intermediary between a 2D application and display-device screen-space resources allocated by the window manager.

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

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

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

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

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

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

2.3 The Evolution of 2D Graphics Platforms

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

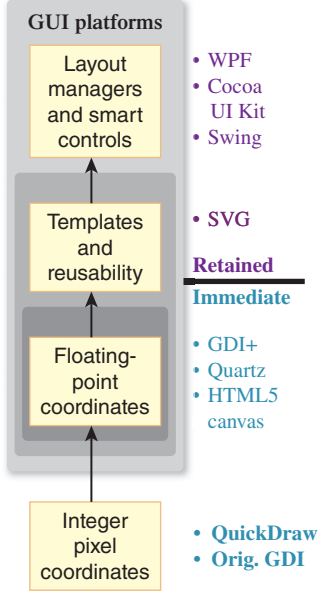


Figure 2.2: Evolution in level of abstraction in commercial 2D graphics platforms—from immediate mode to retained mode.

2.3.1 From Integer to Floating-Point Coordinates

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

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

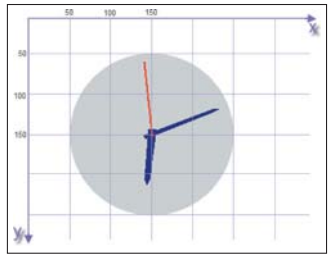


Figure 2.3: Clock scene with GDI coordinate-system overlay.

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

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

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

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

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

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

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

2.3.2 Immediate-Mode versus Retained-Mode Platforms

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

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

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

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

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

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

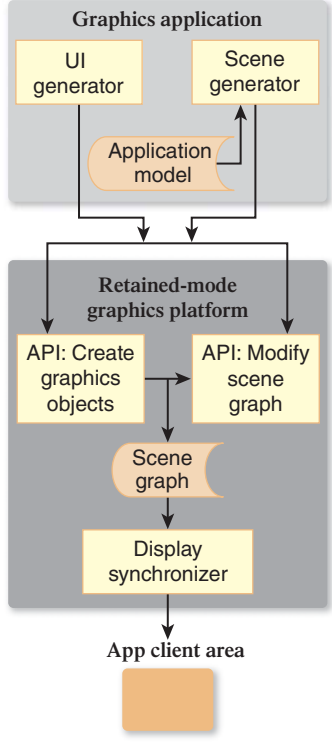


Figure 2.4: Schematic of a graphics application atop a retained-mode platform storing a scene graph.

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

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

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

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

2.3.3 Procedural versus Declarative Specification

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

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

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

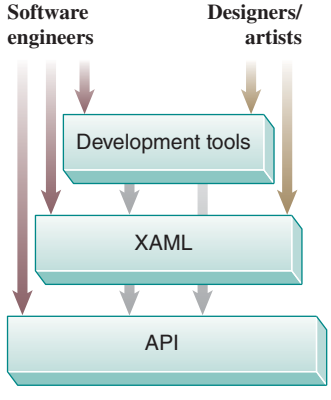


Figure 2.5: WPF application/ developer interface layers.

2.3.3.1 Lowest Layer: Object-Oriented API

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

2.3.3.2 Middle Layer: XAML

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

2.3.3.3 Highest Layer: Tools

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

2.4 Specifying a 2D Scene Using WPF

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

2.4.1 The Structure of an XAML Application

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

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

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

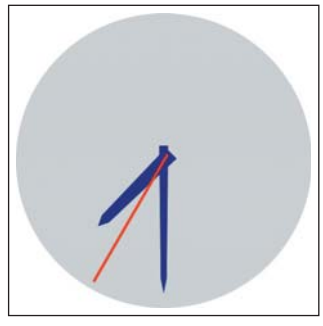


Figure 2.6: WPF-based clock application.

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

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

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

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

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

2.4.2 Specifying the Scene via an Abstract Coordinate System

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

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

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

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

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

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

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

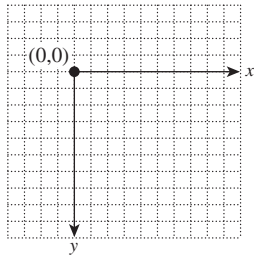


Figure 2.7: Abstract coordinate system.

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

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

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

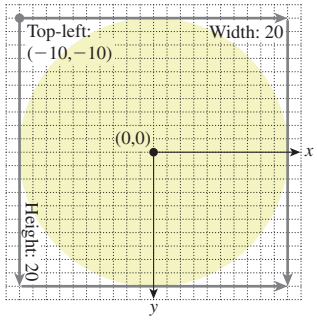


Figure 2.8: Defining the clock face's geometry using our abstract coordinate system.

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

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

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

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

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

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

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

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

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

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

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

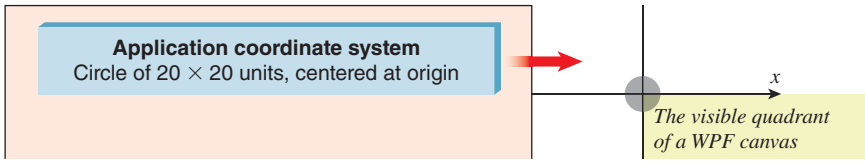


Figure 2.10: Schematic view of our application's initial specification of the clock-face ellipse.

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

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

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

2.4.3 The Spectrum of Coordinate-System Choices

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

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

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

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

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

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

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

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

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

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

2.4.4 The WPF Canvas Coordinate System

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

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

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

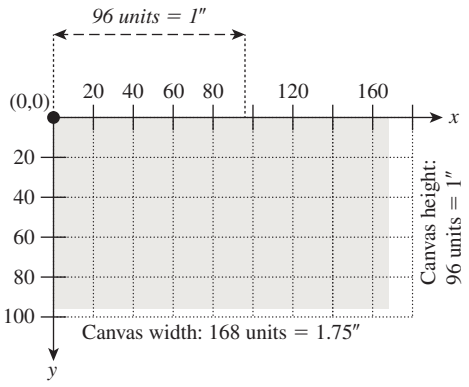


Figure 2.11: A WPF canvas of size 168 × 96 *units, with* `ClipToBounds=True`*. Note the hardwired location of the origin and the hardwired semantics of* 96 *units to the physical inch. On the display device, when rendered by an accurate device driver, this canvas will appear with a size of* 1.75 × 1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

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

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

2.4.5 Using Display Transformations

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

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

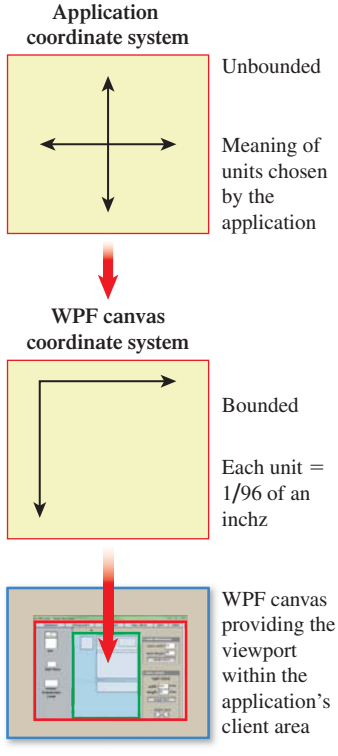


Figure 2.12: Progression from an application's abstract coordinate system, then into the WPF canvas coordinate system, and finally onto the display device as a part of the application window's client area.

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

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

```
<Canvas ... >

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


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

</Canvas>
```

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

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

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

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

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

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

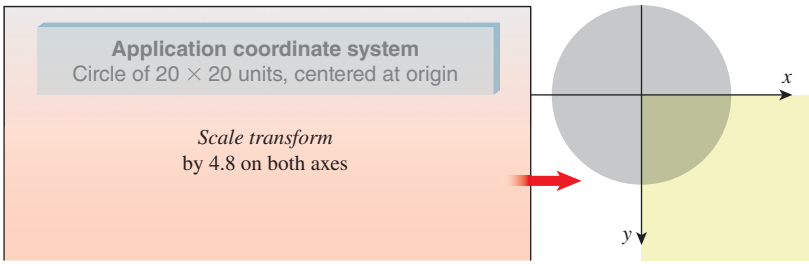


Figure 2.13: Schematic view of our application now enhanced with a scale transform.

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

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

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

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

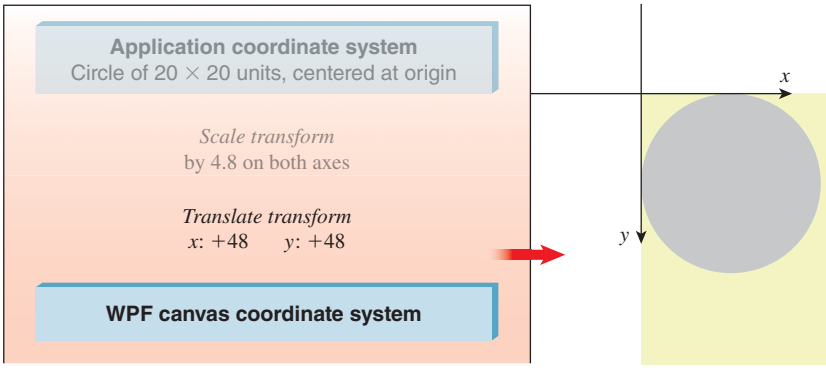


Figure 2.14: Schematic view of our application now enhanced with a two-step display-transform sequence (scale and translate).

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

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

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

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

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

2.4.6 Creating and Using Modular Templates

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

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

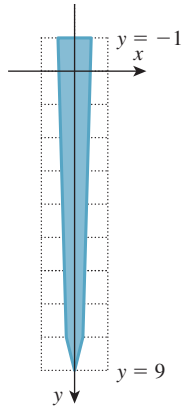


Figure 2.15: Geometry of our clock-hand template.

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

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

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

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

```
<Canvas ... >

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


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

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

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

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

```
  <!- THE SCENE ->

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

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

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

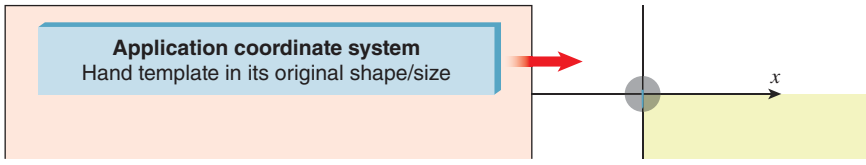


Figure 2.16: Minute hand subjected to the display-transform sequence (1 of 3).

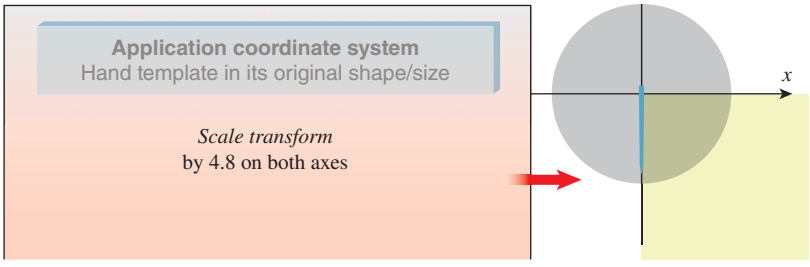


Figure 2.17: Minute hand subjected to the display-transform sequence (2 of 3).

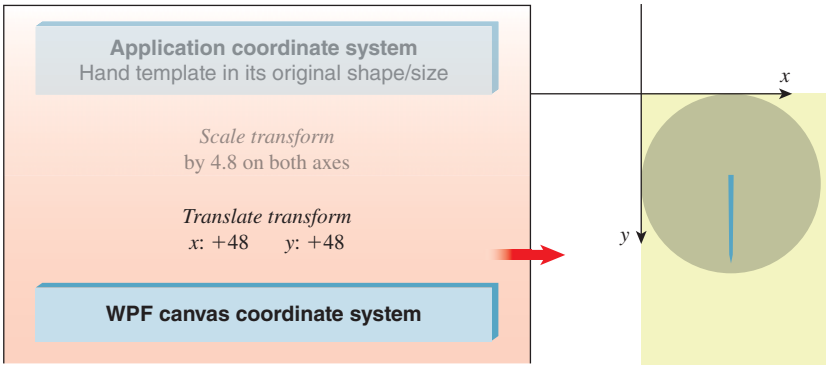


Figure 2.18: Minute hand subjected to the display-transform sequence (3 of 3).

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

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

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

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

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

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

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

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

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

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

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

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

</Canvas>
```

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

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

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

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

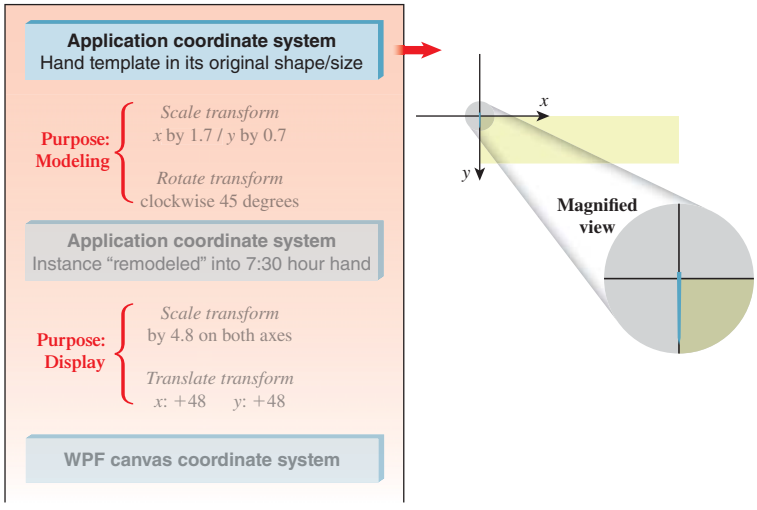


Figure 2.19: Instance of hand template, prior to modeling transform.

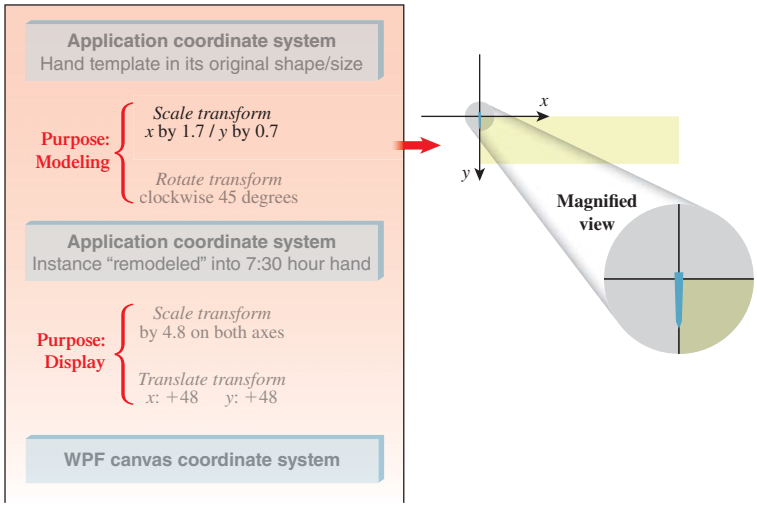


Figure 2.20: Instance of hand template, transformed into hour-hand shape.

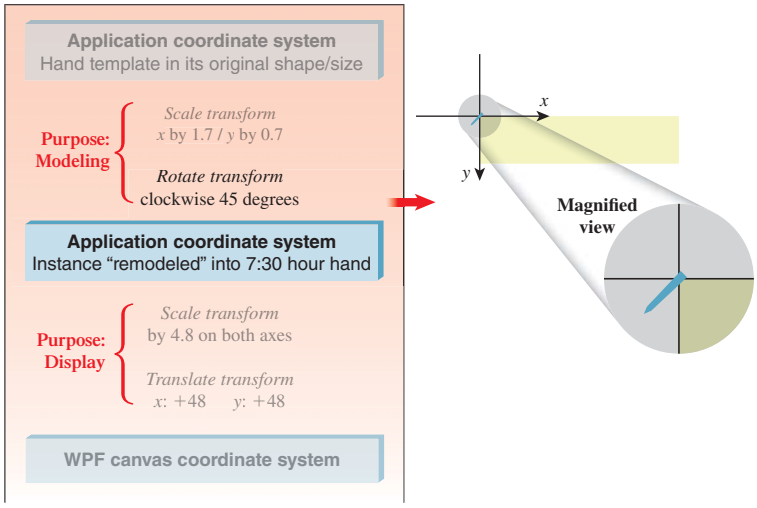


Figure 2.21: Final result of modeling transform that constructs an hour hand at the 7:30 position.

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

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

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

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

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

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

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

2.5 Dynamics in 2D Graphics Using WPF

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

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

2.5.1 Dynamics via Declarative Animation

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

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

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

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

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

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

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

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

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

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

```
<TransformGroup>
   <ScaleTransform ScaleX="1.7" ScaleY="0.7" />
   <!- Rotate into 12 o'clock default position ->
   <RotateTransform Angle="180"/>
   <!- Additional rotation for animation to show actual time: ->
   <RotateTransform x:Name="ActualTimeHour" Angle="0"/>
</TransformGroup>
```

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

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

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

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

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

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

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

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

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

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

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

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

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

2.5.2 Dynamics via Procedural Code

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

2.6 Supporting a Variety of Form Factors

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

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

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

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

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

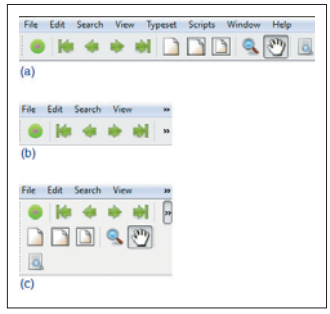


Figure 2.22: Example of automated UI layout adaptation in a Windows application.

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

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

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

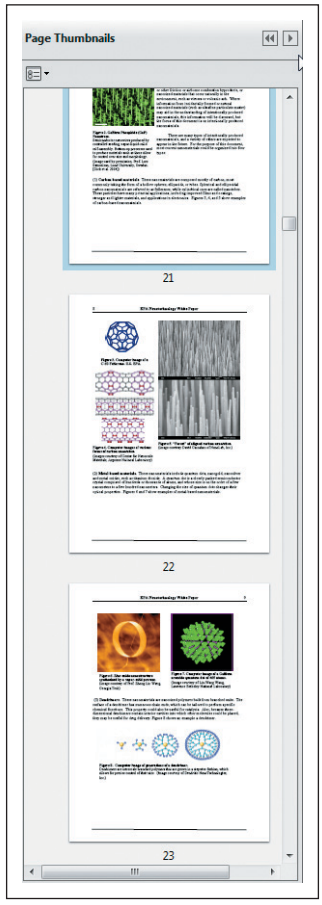


Figure 2.23: Example of automated scene adaptation in a popular PDF viewer.

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

2.7 Discussion and Further Reading

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

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

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

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

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

Chapter 3

An Ancient Renderer Made Modern

3.1 A Dürer Woodcut

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

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

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

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

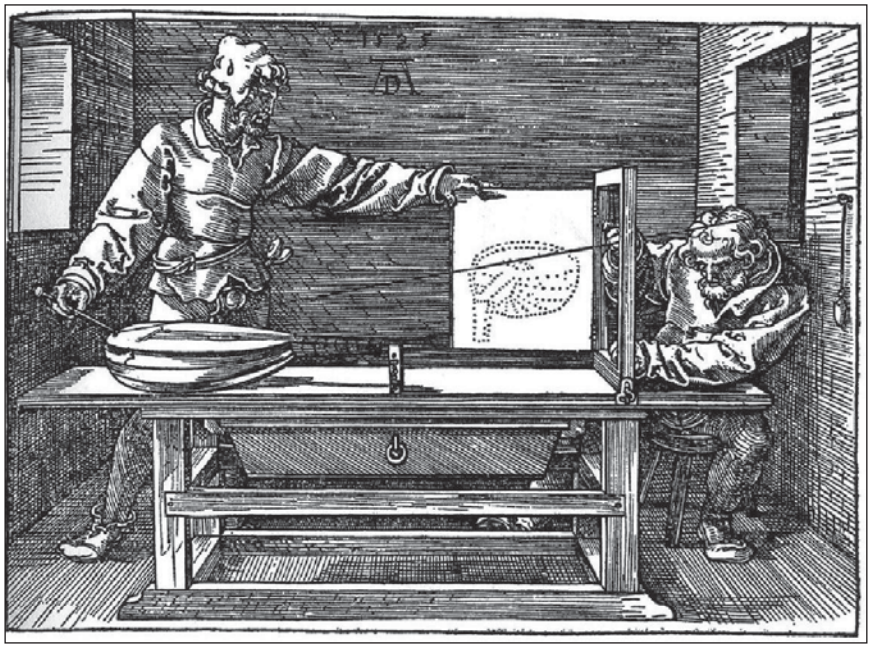


Figure 3.1: Two people using an early "rendering engine" to make a picture of a lute.

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

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

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

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

Listing 3.1: Pseudocode for the Dürer perspective rendering algorithm.

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

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

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

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

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

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

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

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

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

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

Figure 3.2: A different Dürer rendering approach.

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

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

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

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

3.2 Visibility

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

3.3 Implementation

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

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

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

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

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

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

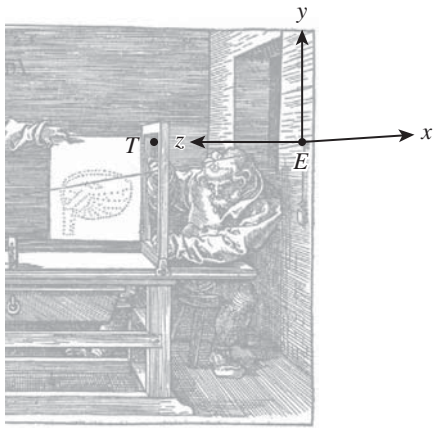


Figure 3.3: The coordinate system for the Dürer woodcut: The origin is at the screw eye, labeled E, and the y- and z-coordinate axes are shown there. The picture frame lies in the plane $z = 1$, parallel to the plane of the wall, $z = 0$. The x-coordinate arrow is horizontal, lying in the plane of the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

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

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

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

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

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

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

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

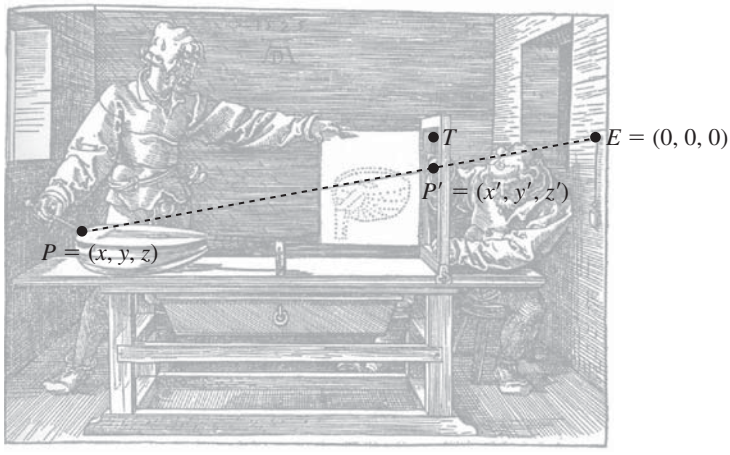


Figure 3.4: The point $P = (x, y, z)$ is on our object. The string from P to the eye, E, will pass through the window frame at some location $P' = (x', y', z')$. Note that $z' = 1$, because we chose our coordinates to make that happen.

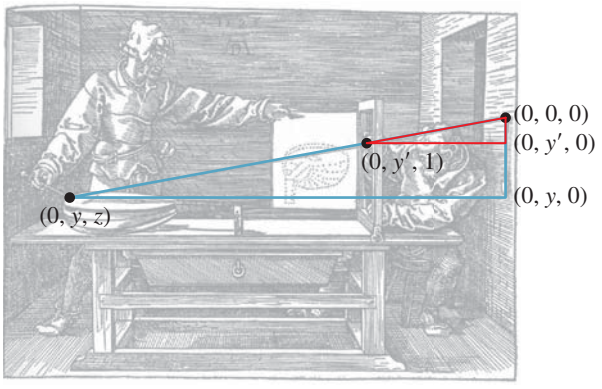


Figure 3.5: Two similar triangles overlaid on the picture in the $x = 0$ plane. The vertical edges of the small red and large blue triangles have lengths y' and y, respectively. What are the lengths of their horizontal edges?

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

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

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

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

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

Listing 3.3: Minor alteration to the Dürer rendering algorithm.

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

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

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

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

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

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

3.3.1 Drawing

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

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

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

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

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

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

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

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

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

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

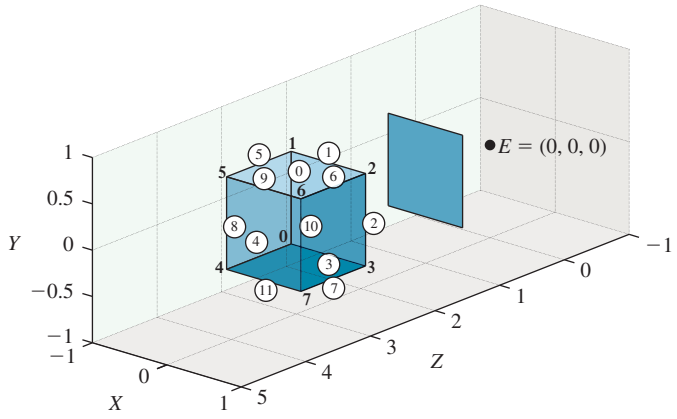


Figure 3.6: The labels for the vertices and edges of our cube model. Edge indices are in circles. The eyepoint and frame from the Dürer woodcut are also included, although we have chosen to adjust their relative positions by placing the frame in this case so that it extends from $-\frac{1}{2}$ to $\frac{1}{2}$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

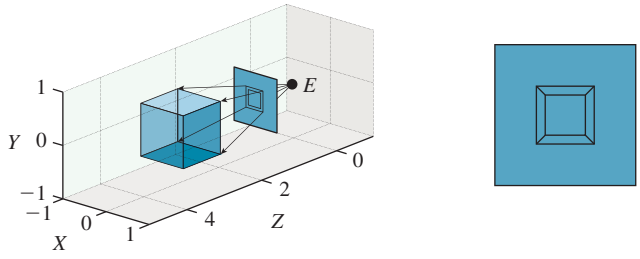


Figure 3.7: The result of the rendering algorithm, (a) shown in place (i.e., drawn in the frame), with rays from the eye to the four near corners of the cube shown, projecting those corners onto the picture plane, and (b) seen directly, with the surrounding square (which ranges from $-\frac{1}{2}$ to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

This enhanced model is shown in Figure 3.6.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.4 The Program

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

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

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

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

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

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

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

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

    Point [] pictureVertices = new Point[nPoints];

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

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

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

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

    ...
}
```

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

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

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

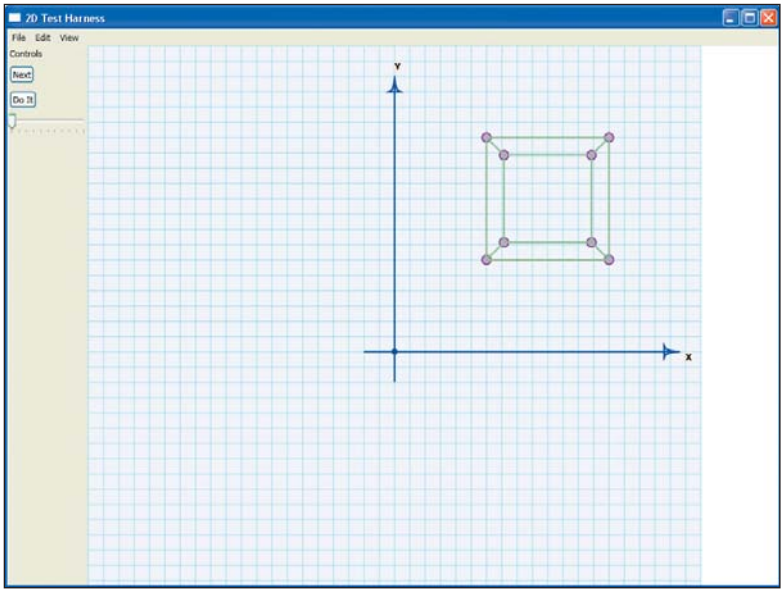


Figure 3.8: The result of the Dürer program: a wire-frame cube, shown in perspective, on a background that looks like graph paper. The axes on the graph paper are part of the `GraphPaper` *itself, set up by the test bed, and are not drawn by the Dürer rendering part of the program.*

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

3.5 Limitations

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

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

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

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

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

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

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

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

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

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

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

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

3.6 Discussion and Further Reading

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

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