COMPUTER GRAPHICS

PRINCIPLES AND PRACTICE

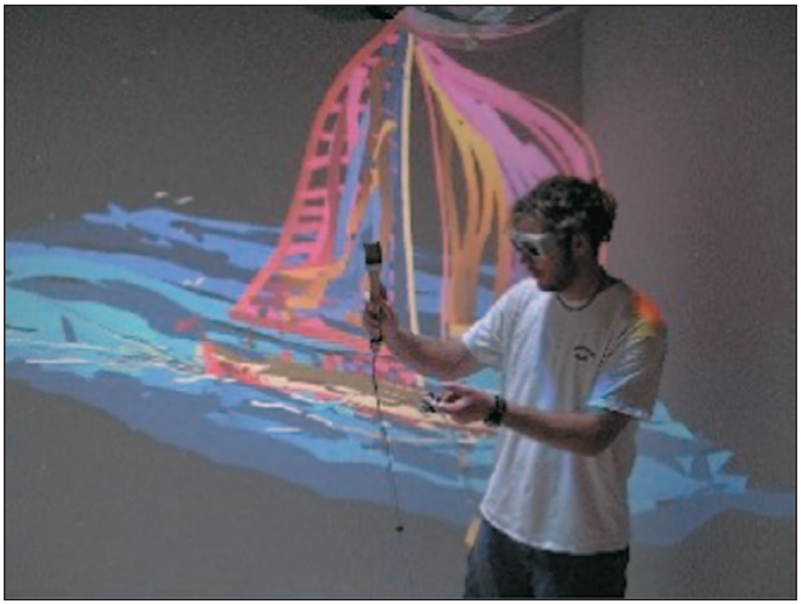
THIRD EDITION

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

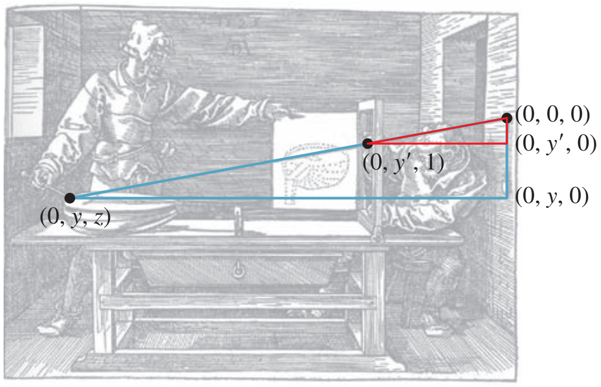
FREE SAMPLE CHAPTER

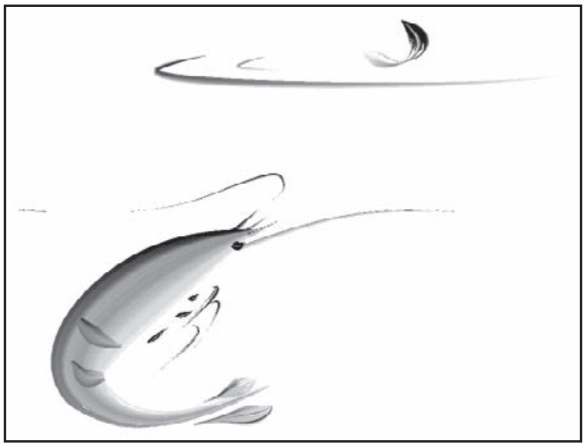
SHARE WITH OTHERS

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.

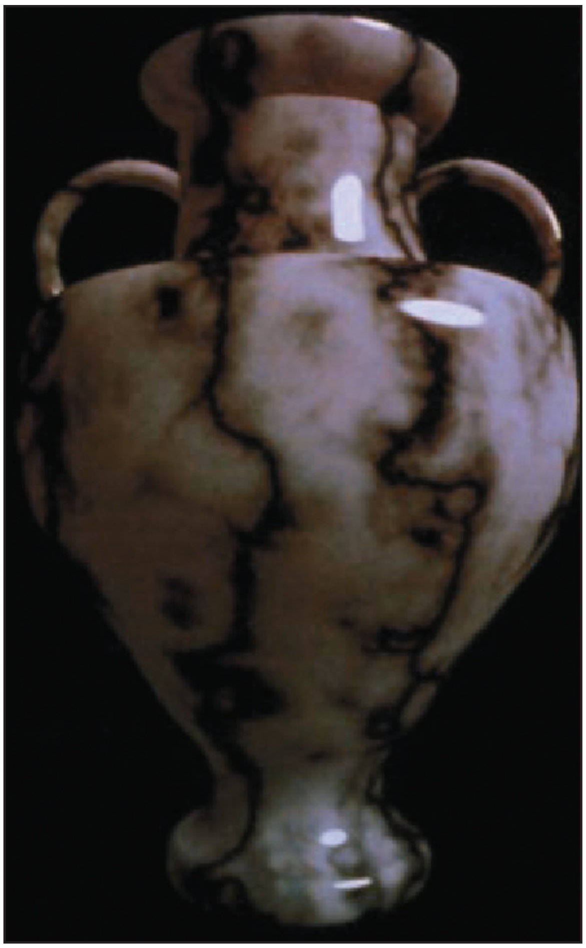


Courtesy of Daniel Keefe, University of Minnesota.

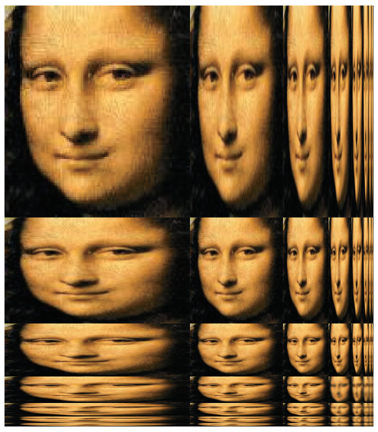




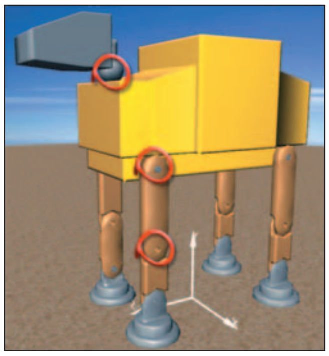
Courtesy of Steve Strassmann. © 1986 ACM, Inc. Reprinted by permission.



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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

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

Copyright © 2014 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

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

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

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

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

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

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

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

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

This page intentionally left blank

Contents at a Glance

Contents

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

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

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

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.

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. Somasegur, helped to both initiate and complete this project.

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

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

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

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

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

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

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

For the Student

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

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

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

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

For the Teacher

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

We recommend to you (aside from the suggestions in the remainder of this preface) two things. The first is that you encourage, or even require, that your students answer the inline exercises in the book. To the student who says, "I've got too much to do! I can't waste time stopping to do some exercise," just say, "We don’t have time to stop for gas . . . we’re already late.” The second is that you assign your students projects or homeworks that have both a fixed goal and an open-ended component. The steady students will complete the fixed-goal parts and learn the material you want to cover. The others, given the chance to do something fun, may do things with the open-ended exercises that will amaze you. And in doing so, they’ll find that they need to learn things that might seem *just* out of reach, until they suddenly master them, and become empowered. Graphics is a terrific medium for this: Successes are instantly visible and rewarding, and this sets up a feedback loop for advancement. The combination of visible feedback with the ideas of scalability that they’ve encountered elsewhere in computer science can be revelatory.

Discussion and Further Reading

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

About the Authors

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

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

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

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

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

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

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

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

This page intentionally left blank

Chapter 10

Transformations in Two Dimensions

10.1 Introduction

As you saw in Chapters 2 and 6, when we think about taking an object for which we have a geometric model and putting it in a scene, we typically need to do three things: *Move* the object to some location, *scale* it up or down so that it fits well with the other objects in the scene, and *rotate* it until it has the right orientation. These operations—translation, scaling, and rotation—are part of every graphics system. Both scaling and rotation are **linear transformations** on the coordinates of the object's points. Recall that a linear transformation,

$$T : \mathbf{R}^2 \to \mathbf{R}^2, \tag{10.1}$$

is one for which $T(\mathbf{v} + \alpha\mathbf{w}) = T(\mathbf{v}) + \alpha T(\mathbf{w})$ for any two vectors $\mathbf{v}$ and $\mathbf{w}$ in $\mathbf{R}^2$, and any real number α. Intuitively, it's a transformation that preserves lines and leaves the origin unmoved.

Inline Exercise 10.1: Suppose T is linear. Insert $\alpha = 1$ in the definition of linearity. What does it say? Insert $\mathbf{v} = \mathbf{0}$ in the definition. What does it say?

Inline Exercise 10.2: When we say that a linear transformation "preserves lines," we mean that if $\boldsymbol{\ell}$ is a line, then the set of points $T(\boldsymbol{\ell})$ must also *lie in* some line. You might expect that we'd require that $T(\boldsymbol{\ell})$ actually *be* a line, but that would mean that transformations like "project everything perpendicularly onto the x-axis" would not be counted as "linear." For this particular projection transformation, describe a line $\boldsymbol{\ell}$ such that $T(\boldsymbol{\ell})$ is contained in a line, but is not itself a line.

The definition of linearity guarantees that for any linear transformation T, we have $T(\mathbf{0}) = \mathbf{0}$: If we choose $\mathbf{v} = \mathbf{w} = \mathbf{0}$ and $\alpha = 1$, the definition tells us that

$$T(\mathbf{0}) = T(\mathbf{0} + 1\mathbf{0}) = T(\mathbf{0}) + 1T(\mathbf{0}) = T(\mathbf{0}) + T(\mathbf{0}). \tag{10.2}$$

Subtracting $T(\mathbf{0})$ from the first and last parts of this chain gives us $\mathbf{0} = T(\mathbf{0})$. This means that **translation**—moving every point of the plane by the same amount—is, in general, *not* a linear transformation except in the special case of translation by zero, in which all points are left where they are. Shortly we'll describe a trick for putting the Euclidean plane into $\mathbf{R^3}$ (but *not* as the $z = 0$ plane as is usually done); once we do this, we'll see that certain linear transformations on $\mathbf{R^3}$ end up performing translations on this embedded plane.

For now, let's look at only the plane. We assume that you have *some* familiarity with linear transformations already; indeed, the serious student of computer graphics should, at some point, study linear algebra carefully. But one can learn a great deal about graphics with only a modest amount of knowledge of the subject, which we summarize here briefly.

In the first few sections, we use the convention of most linear-algebra texts: The vectors are arrows at the origin, and we think of the vector $\begin{bmatrix} u \\ v \end{bmatrix}$ as being identified with the point (u, v). Later we'll return to the point-vector distinction.

For any 2×2 matrix $\mathbf{M}$, the function $\mathbf{v} \mapsto \mathbf{Mv}$ is a linear transformation from $\mathbf{R^2}$ to $\mathbf{R^2}$. We refer to this as a **matrix transformation.** In this chapter, we look at five such transformations in detail, study matrix transformations in general, and introduce a method for incorporating translation into the matrix-transformation formulation. We then apply these ideas to transforming objects *and* changing coordinate systems, returning to the clock example of Chapter 2 to see the ideas in practice.

10.2 Five Examples

We begin with five examples of linear transformations in the plane; we'll refer to these by the names $T_1, \ldots, T_5$ throughout the chapter.

Example 1: Rotation. Let $\mathbf{M}_1 = \begin{bmatrix} \cos 30^\circ & -\sin 30^\circ \\ \sin 30^\circ & \cos 30^\circ \end{bmatrix}$ and

$$T_1 : \mathbf{R^2} \to \mathbf{R^2} : \begin{bmatrix} x \\ y \end{bmatrix} \mapsto \mathbf{M}_1 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos 30^\circ & -\sin 30^\circ \\ \sin 30^\circ & \cos 30^\circ \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}. \tag{10.3}$$

Recall that $\mathbf{e}_1$ denotes the vector $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\mathbf{e}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$; this transformation sends $\mathbf{e}_1$ to the vector $\begin{bmatrix} \cos 30^\circ \\ \sin 30^\circ \end{bmatrix}$ and $\mathbf{e}_2$ to $\begin{bmatrix} -\sin 30^\circ \\ \cos 30^\circ \end{bmatrix}$, which are vectors that are 30° counterclockwise from the x- and y-axes, respectively (see Figure 10.1).

There's nothing special about the number 30 in this example; by replacing 30° with any angle, you can build a transformation that rotates things counterclockwise by that angle.

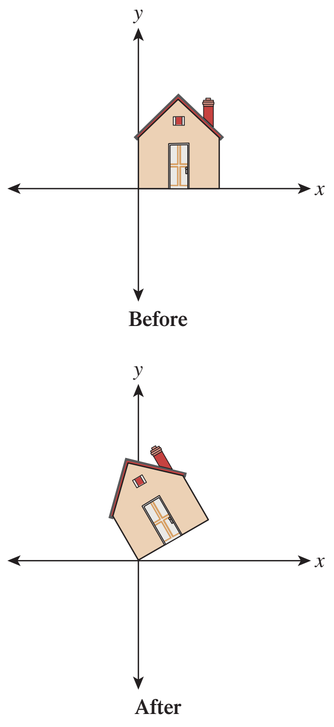


Figure 10.1: Rotation by 30°.

Inline Exercise 10.3: Write down the matrix transformation that rotates everything in the plane by 180° counterclockwise. Actually compute the sines and cosines so that you end up with a matrix filled with numbers in your answer. Apply this transformation to the corners of the unit square, $(0,0)$, $(1,0)$, $(0,1)$, and $(1,1)$.

Example 2: Nonuniform scaling. Let $\mathbf{M}_2 = \begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix}$ and

$$T_2 : \mathbf{R^2} \to \mathbf{R^2} : \begin{bmatrix} x \\ y \end{bmatrix} \mapsto \mathbf{M}_2 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3x \\ 2y \end{bmatrix}. \tag{10.4}$$

This transformation stretches everything by a factor of three in the x-direction and a factor of two in the y-direction, as shown in Figure 10.2. If both stretch factors were three, we'd say that the transformation "scaled things up by three" and is a **uniform scaling transformation.** T_2 represents a generalization of this idea: Rather than scaling uniformly in each direction, it's called a **nonuniform scaling transformation** or, less formally, a **nonuniform scale.**

Once again the example generalizes: By placing numbers other than 2 and 3 along the diagonal of the matrix, we can scale each axis by any amount we please. These scaling amounts can include zero and negative numbers.

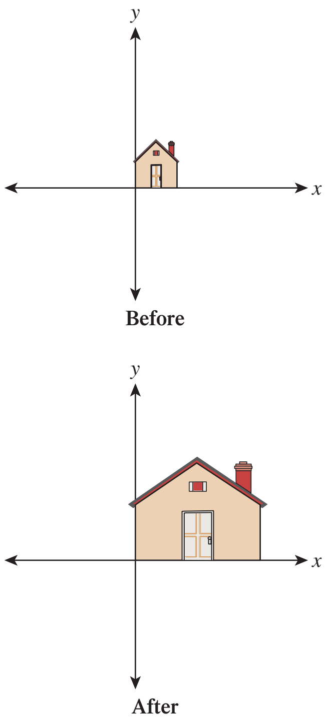


Figure 10.2: T_2 stretches the x-axis by three and the y-axis by two.

Inline Exercise 10.4: Write down the matrix for a uniform scale by -1. How does your answer relate to your answer to inline Exercise 10.3? Can you explain?

Inline Exercise 10.5: Write down a transformation matrix that scales in x by zero and in y by 1. Informally describe what the associated transformation does to the house.

Example 3: Shearing. Let $\mathbf{M}_3 = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$ and

$$T_3 : \mathbf{R^2} \to \mathbf{R^2} : \begin{bmatrix} x \\ y \end{bmatrix} \mapsto \mathbf{M}_3 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + 2y \\ y \end{bmatrix}. \tag{10.5}$$

As Figure 10.3 shows, T_3 preserves height along the y-axis but moves points parallel to the x-axis, with the amount of movement determined by the y-value. The x-axis itself remains fixed. Such a transformation is called a **shearing transformation.**

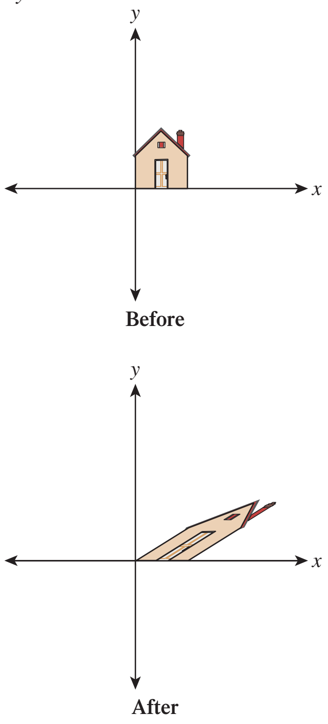


Figure 10.3: A shearing transformation, T_3.

Inline Exercise 10.6: Generalize to build a transformation that keeps the y-axis fixed but shears vertically instead of horizontally.

Example 4: A general transformation. Let $\mathbf{M}_4 = \begin{bmatrix} 1 & -1 \\ 2 & 2 \end{bmatrix}$ and

$$T_4 : \mathbf{R^2} \to \mathbf{R^2} : \begin{bmatrix} x \\ y \end{bmatrix} \mapsto \mathbf{M}_4 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 2 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}. \tag{10.6}$$

Figure 10.4 shows the effects of T_4. It distorts the house figure, but not by just a rotation or scaling or shearing along the coordinate axes.

Example 5: A degenerate (or singular) transformation Let

$$T_5 : \mathbf{R}^2 \to \mathbf{R}^2 : \begin{bmatrix} x \\ y \end{bmatrix} \mapsto \begin{bmatrix} 1 & -1 \\ 2 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x - y \\ 2x - 2y \end{bmatrix}. \tag{10.7}$$

Figure 10.5 shows why we call this transformation **degenerate:** Unlike the others, it collapses the whole two-dimensional plane down to a one-dimensional subspace, a line. There's no longer a nice correspondence between points in the domain and points in the codomain: Certain points in the codomain no longer correspond to *any* point in the domain; others correspond to *many* points in the domain. Such a transformation is also called **singular,** as is the matrix defining it. Those familiar with linear algebra will note that this is equivalent to saying that the determinant of $\mathbf{M}_5 = \begin{bmatrix} 1 & -1 \\ 2 & -2 \end{bmatrix}$ is zero, or saying that its columns are linearly dependent.

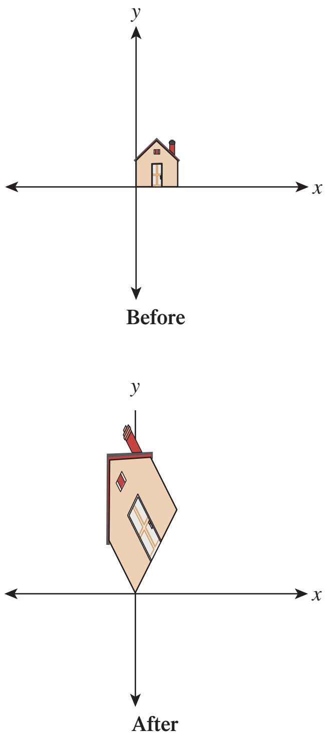


Figure 10.4: A general transformation. The house has been quite distorted, in a way that's hard to describe simply, as we've done for the earlier examples.

10.3 Important Facts about Transformations

Here we'll describe several properties of linear transformations from $\mathbf{R}^2$ to $\mathbf{R}^2$. These properties are important in part because they all generalize: They apply (in some form) to transformations from $\mathbf{R}^n$ to $\mathbf{R}^k$ for any n and k. We'll mostly be concerned with values of n and k between 1 and 4; in this section, we'll concentrate on $n = k = 2$.

10.3.1 Multiplication by a Matrix Is a Linear Transformation

If $\mathbf{M}$ is a 2×2 matrix, then the function $T_{\mathbf{M}}$ defined by

$$T_{\mathbf{M}} : \mathbf{R}^2 \to \mathbf{R}^2 : \mathbf{x} \mapsto \mathbf{Mx} \tag{10.8}$$

is linear. All five examples above demonstrate this.

For nondegenerate transformations, lines are sent to lines, as T_1 through T_4 show. For degenerate ones, a line may be sent to a single point. For instance, T_5 sends the line consisting of all vectors of the form $\begin{bmatrix} b \\ b \end{bmatrix}$ to the zero vector.

Because multiplication by a matrix $\mathbf{M}$ is always a linear transformation, we'll call $T_{\mathbf{M}}$ the **transformation associated to the matrix M.**

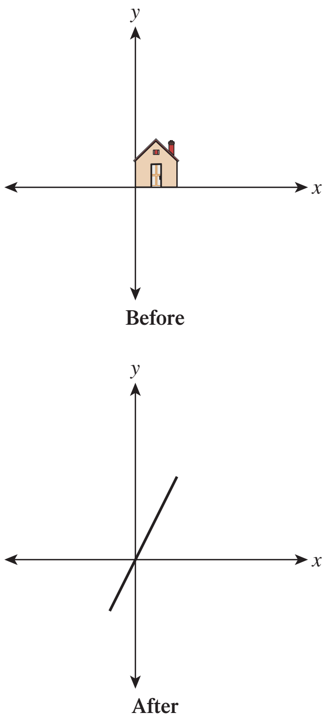


Figure 10.5: A degenerate transformation, T_5.

10.3.2 Multiplication by a Matrix Is the *Only* Linear Transformation

In $\mathbf{R}^n$, it turns out that for *every* linear transform T, there's a matrix $\mathbf{M}$ with $T(\mathbf{x}) = \mathbf{Mx}$, which means that every linear transformation is a matrix transformation. We'll see in Section 10.3.5 how to find $\mathbf{M}$, given T, even if T is expressed in some other way. This will show that the matrix $\mathbf{M}$ is completely determined by the transformation T, and we can thus call it the **matrix associated to the transformation.**

As a special example, the matrix $\mathbf{I}$, with ones on the diagonal and zeroes off the diagonal, is called the **identity matrix;** the associated transformation

$$T(\mathbf{x}) = \mathbf{I}\mathbf{x} \tag{10.9}$$

is special: It's the identity transformation that leaves every vector $\mathbf{x}$ unchanged.

Inline Exercise 10.7: There is an identity matrix of every size: a 1×1 identity, a 2×2 identity, etc. Write out the first three.

10.3.3 Function Composition and Matrix Multiplication Are Related

If $\mathbf{M}$ and $\mathbf{K}$ are 2×2 matrices, then they define transformations $T_{\mathbf{M}}$ and $T_{\mathbf{K}}$. When we compose these, we get the transformation

$$\begin{aligned} T_{\mathbf{M}} \circ T_{\mathbf{K}} : \mathbf{R^2} \to \mathbf{R^2} : \mathbf{x} \mapsto T_{\mathbf{M}}(T_{\mathbf{K}}(\mathbf{x})) &= T_{\mathbf{M}}(\mathbf{Kx}) && (10.10)\\ &= \mathbf{M}(\mathbf{Kx}) && (10.11)\\ &= (\mathbf{MK})\mathbf{x} && (10.12)\\ &= T_{\mathbf{MK}}(\mathbf{x}). && (10.13) \end{aligned}$$

In other words, the composed transformation is also a matrix transformation, with matrix $\mathbf{MK}$. Note that when we write $T_{\mathbf{M}}(T_{\mathbf{K}}(\mathbf{x}))$, the transformation $T_{\mathbf{K}}$ is applied *first.* So, for example, if we look at the transformation $T_2 \circ T_3$, it first shears the house and *then* scales the result nonuniformly.

Inline Exercise 10.8: Describe the appearance of the house after transforming it by $T_1 \circ T_2$ and after transforming it by $T_2 \circ T_1$.

10.3.4 Matrix Inverse and Inverse Functions Are Related

A matrix $\mathbf{M}$ is **invertible** if there's a matrix $\mathbf{B}$ with the property that $\mathbf{BM} = \mathbf{MB} = \mathbf{I}$. If such a matrix exists, it's denoted $\mathbf{M}^{-1}$.

If $\mathbf{M}$ is invertible and $S(\mathbf{x}) = \mathbf{M}^{-1}\mathbf{x}$, then S is the inverse function of $T_{\mathbf{M}}$, that is,

$$\begin{aligned} S(T_{\mathbf{M}}(\mathbf{x})) &= \mathbf{x} \quad \text{and} && (10.14)\\ T_{\mathbf{M}}(S(\mathbf{x})) &= \mathbf{x}. && (10.15) \end{aligned}$$

Inline Exercise 10.9: Using Equation 10.13, explain why Equation 10.15 holds.

If $\mathbf{M}$ is not invertible, then $T_{\mathbf{M}}$ has no inverse.

Let's look at our examples. The matrix for T_1 has an inverse: Simply replace 30 by -30 in all the entries. The resultant transformation rotates clockwise by 30°; performing one rotation and then the other effectively does nothing (i.e., it is the identity transformation). The inverse for the matrix for T_2 is diagonal, with entries

$\frac{1}{3}$ and $\frac{1}{2}$. The inverse of the matrix for T_3 is $\begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}$ (note the negative sign). The associated transformation also shears parallel to the x-axis, but vectors in the upper half-plane are moved to the *left,* which undoes the moving to the right done by T_3.

For these first three it was fairly easy to guess the inverse matrices, because we could understand how to invert the transformation. The inverse of the matrix for T_4 is

$$\frac{1}{4}\begin{bmatrix} 2 & 1 \\ -2 & 1 \end{bmatrix}, \tag{10.16}$$

which we computed using a general rule for inverses of 2×2 matrices (the only such rule worth memorizing):

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad-bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}. \tag{10.17}$$

Finally, for T_5, the matrix has no inverse; if it did, the function T_5 would be invertible: It would be possible to identify, for each point in the codomain, a single point in the domain that's sent there. But we've already seen this isn't possible.

Inline Exercise 10.10: Apply the formula from Equation 10.17 to the matrix for T_5 to attempt to compute its inverse. What goes wrong?

10.3.5 Finding the Matrix for a Transformation

We've said that every linear transformation really is just multiplication by some matrix, but how do we *find* that matrix? Suppose, for instance, that we'd like to find a linear transformation to flip our house across the y-axis so that the house ends up on the left side of the y-axis. (Perhaps you can guess the transformation that does this, and the associated matrix, but we'll work through the problem directly.)

The key idea is this: If we know where the transformation sends $\mathbf{e}_1$ and $\mathbf{e}_2$, we know the matrix. Why? We know that the transformation must have the form

$$T\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix}; \tag{10.18}$$

we just don't know the values of a, b, c, and d. Well, $T(\mathbf{e}_1)$ is then

$$T\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} a \\ c \end{bmatrix}. \tag{10.19}$$

Similarly, $T(\mathbf{e}_2)$ is the vector $\begin{bmatrix} b \\ d \end{bmatrix}$. So knowing $T(\mathbf{e}_1)$ and $T(\mathbf{e}_2)$ tells us all the matrix entries. Applying this to the problem of flipping the house, we know that $T(\mathbf{e}_1) = -\mathbf{e}_1$, because we want a point on the positive x-axis to be sent to the corresponding point on the negative x-axis, so $a = -1$ and $c = 0$. On the other hand, $T(\mathbf{e}_2) = \mathbf{e}_2$, because every vector on the y-axis should be left untouched, so $b = 0$ and $d = 1$. Thus, the matrix for the house-flip transformation is just

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{10.20}$$

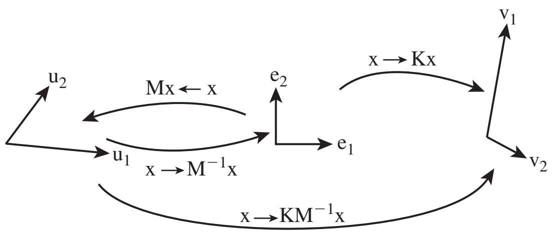


Figure 10.6: Multiplication by the matrix $\mathbf{M}$ *takes* $\mathbf{e}_1$ *and* $\mathbf{e}_2$ *to* $\mathbf{u}_1$ *and* $\mathbf{u}_2$*, respectively, so multiplying* $\mathbf{M}^{-1}$ *does the opposite. Multiplying by* $\mathbf{K}$ *takes* $\mathbf{e}_1$ *and* $\mathbf{e}_2$ *to* $\mathbf{v}_1$ *and* $\mathbf{v}_2$*, so multiplying first by* $\mathbf{M}^{-1}$ *and then by* $\mathbf{K}$*, that is, multiplying by* $\mathbf{KM}^{-1}$*, takes* $\mathbf{u}_1$ *to* $\mathbf{e}_1$ *to* $\mathbf{v}_1$*, and similarly for* $\mathbf{u}_2$*.*

Inline Exercise 10.11: (a) Find a matrix transformation sending $\mathbf{e}_1$ to $\begin{bmatrix} 0 \\ 4 \end{bmatrix}$ and $\mathbf{e}_2$ to $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$.
(b) Use the relationship of matrix inverse to the inverse of a transform, and the formula for the inverse of a 2×2 matrix, to find a transformation sending $\begin{bmatrix} 0 \\ 4 \end{bmatrix}$ to $\mathbf{e}_1$ and $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ to $\mathbf{e}_2$ as well.

As Inline Exercise 10.11 shows, we now have the tools to send the **standard basis vectors** $\mathbf{e}_1$ and $\mathbf{e}_2$ to any two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, and vice versa (provided that $\mathbf{v}_1$ and $\mathbf{v}_2$ are independent, that is, neither is a multiple of the other). We can combine this with the idea that composition of linear transformations (performing one after the other) corresponds to multiplication of matrices and thus create a solution to a rather general problem.

Problem: Given independent vectors $\mathbf{u}_1$ and $\mathbf{u}_2$ and any two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, find a linear transformation, in matrix form, that sends $\mathbf{u}_1$ to $\mathbf{v}_1$ and $\mathbf{u}_2$ to $\mathbf{v}_2$.

Solution: Let $\mathbf{M}$ be the matrix whose columns are $\mathbf{u}_1$ and $\mathbf{u}_2$. Then

$$T : \mathbf{R^2} \to \mathbf{R^2} : \mathbf{x} \mapsto \mathbf{Mx} \tag{10.21}$$

sends $\mathbf{e}_1$ to $\mathbf{u}_1$ and $\mathbf{e}_2$ to $\mathbf{u}_2$ (see Figure 10.6). Therefore,

$$S : \mathbf{R^2} \to \mathbf{R^2} : \mathbf{x} \mapsto \mathbf{M}^{-1}\mathbf{x} \tag{10.22}$$

sends $\mathbf{u}_1$ to $\mathbf{e}_1$ and $\mathbf{u}_2$ to $\mathbf{e}_2$.

Now let $\mathbf{K}$ be the matrix with columns $\mathbf{v}_1$ and $\mathbf{v}_2$. The transformation

$$R : \mathbf{R^2} \to \mathbf{R^2} : \mathbf{x} \mapsto \mathbf{Kx} \tag{10.23}$$

sends $\mathbf{e}_1$ to $\mathbf{v}_1$ and $\mathbf{e}_2$ to $\mathbf{v}_2$.

If we apply first S and then R to $\mathbf{u}_1$, it will be sent to $\mathbf{e}_1$ (by S), and thence to $\mathbf{v}_1$ by R; a similar argument applies to $\mathbf{u}_2$. Writing this in equations,

$$R(S(\mathbf{x})) = R(\mathbf{M}^{-1}\mathbf{x}) \tag{10.24}$$
$$= \mathbf{K}(\mathbf{M}^{-1}\mathbf{x}) \tag{10.25}$$
$$= (\mathbf{KM}^{-1})\mathbf{x}. \tag{10.26}$$

Thus, the matrix for the transformation sending the **u**'s to the **v**'s is just $\mathbf{KM}^{-1}$.

Let's make this concrete with an example. We'll find a matrix sending

$$\mathbf{u}_1 = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \quad \text{and} \quad \mathbf{u}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \tag{10.27}$$

to

$$\mathbf{v}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad \text{and} \quad \mathbf{v}_2 = \begin{bmatrix} 2 \\ -1 \end{bmatrix}, \tag{10.28}$$

respectively. Following the pattern above, the matrices **M** and **K** are

$$\mathbf{M} = \begin{bmatrix} 2 & 1 \\ 3 & -1 \end{bmatrix} \tag{10.29}$$

$$\mathbf{K} = \begin{bmatrix} 1 & 2 \\ 1 & -1 \end{bmatrix}. \tag{10.30}$$

Using the matrix inversion formula (Equation 10.17), we find

$$\mathbf{M}^{-1} = \frac{-1}{5} \begin{bmatrix} -1 & -1 \\ -3 & 2 \end{bmatrix} \tag{10.31}$$

so that the matrix for the overall transformation is

$$\mathbf{J} = \mathbf{KM}^{-1} = \begin{bmatrix} 1 & 2 \\ 1 & -1 \end{bmatrix} \cdot \frac{-1}{5} \begin{bmatrix} -1 & -1 \\ -3 & 2 \end{bmatrix} \tag{10.32}$$

$$= \begin{bmatrix} 7/5 & -3/5 \\ -2/5 & 3/5 \end{bmatrix}. \tag{10.33}$$

As you may have guessed, the kinds of transformations we used in WPF in Chapter 2 are internally represented as matrix transformations, and transformation groups are represented by sets of matrices that are multiplied together to generate the effect of the group.

Inline Exercise 10.12: Verify that the transformation associated to the matrix **J** in Equation 10.32 really does send $\mathbf{u}_1$ to $\mathbf{v}_1$ and $\mathbf{u}_2$ to $\mathbf{v}_2$.

Inline Exercise 10.13: Let $\mathbf{u}_1 = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$ and $\mathbf{u}_2 = \begin{bmatrix} 1 \\ 4 \end{bmatrix}$; pick any two nonzero vectors you like as $\mathbf{v}_1$ and $\mathbf{v}_2$, and find the matrix transformation that sends each $\mathbf{u}_i$ to the corresponding $\mathbf{v}_i$.

The recipe above for building matrix transformations shows the following: Every linear transformation from $\mathbf{R^2}$ to $\mathbf{R^2}$ is determined by its values on two independent vectors. In fact, this is a far more general property: Any linear transformation from $\mathbf{R^2}$ to $\mathbf{R}^k$ is determined by its values on two independent vectors, and indeed, any linear transformation from $\mathbf{R}^n$ to $\mathbf{R}^k$ is determined by its values on n independent vectors (where to make sense of these, we need to extend our definition of "independence" to more than two vectors, which we'll do presently).

10.3.6 Transformations and Coordinate Systems

We tend to think about linear transformations as moving points around, but leaving the origin fixed; we'll often use them that way. Equally important, however, is their use in changing coordinate systems. If we have two coordinate systems on $\mathbf{R}^2$ with the same origin, as in Figure 10.7, then every arrow has coordinates in both the red and the blue systems. The two red coordinates can be written as a vector, as can the two blue coordinates. The vector $\mathbf{u}$, for instance, has coordinates $\begin{bmatrix} 3 \\ 2 \end{bmatrix}$ in the red system and approximately $\begin{bmatrix} -0.2 \\ 3.6 \end{bmatrix}$ in the blue system.

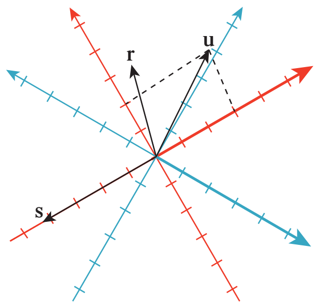


Figure 10.7: Two different coordinate systems for $\mathbf{R}^2$; the vector $\mathbf{u}$, expressed in the red coordinate system, has coordinates 3 and 2, indicated by the dotted lines, while the coordinates in the blue coordinate system are approximately -0.2 and 3.6, where we've drawn, in each case, the positive side of the first coordinate axis in bold.

Inline Exercise 10.14: Use a ruler to find the coordinates of **r** and **s** in each of the two coordinate systems.

We could tabulate every imaginable arrow's coordinates in the red and blue systems to convert from red to blue coordinates. But there is a far simpler way to achieve the same result. The conversion from red coordinates to blue coordinates is *linear* and can be expressed by a matrix transformation. In this example, the matrix is

$$\mathbf{M} = \frac{1}{2} \begin{bmatrix} 1 & -\sqrt{3} \\ \sqrt{3} & 1 \end{bmatrix}. \tag{10.34}$$

Multiplying $\mathbf{M}$ by the coordinates of $\mathbf{u}$ in the red system gets us

$$\begin{aligned} \mathbf{v} &= \mathbf{M}\mathbf{u} && (10.35) \\ &= \frac{1}{2} \begin{bmatrix} 1 & -\sqrt{3} \\ \sqrt{3} & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \end{bmatrix} && (10.36) \\ &= \frac{1}{2} \begin{bmatrix} 3 - 2\sqrt{3} \\ 3\sqrt{3} + 2 \end{bmatrix} && (10.37) \\ &\approx \begin{bmatrix} -0.2 \\ 3.6 \end{bmatrix}, && (10.38) \end{aligned}$$

which is the coordinate vector for $\mathbf{u}$ in the blue system.

Inline Exercise 10.15: Confirm, for each of the other arrows in Figure 10.7, that the same transformation converts red to blue coordinates.

By the way, when creating this example we computed $\mathbf{M}$ just as we did at the start of the preceding section: We found the blue coordinates of each of the two basis vectors for the red coordinate system, and used these as the columns of $\mathbf{M}$.

In the special case where we want to go from the usual coordinates on a vector to its coordinates in some coordinate system with basis vectors $\mathbf{u}_1, \mathbf{u}_2$, which are *unit vectors* and *mutually perpendicular,* the transformation matrix is one whose *rows* are the transposes of $\mathbf{u}_1$ and $\mathbf{u}_2$.

For example, if $\mathbf{u}_1 = \begin{bmatrix} 3/5 \\ 4/5 \end{bmatrix}$ and $\mathbf{u}_2 = \begin{bmatrix} -4/5 \\ 3/5 \end{bmatrix}$ (check for yourself that these are unit length and perpendicular), then the vector $\mathbf{v} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}$, expressed in $\mathbf{u}$-coordinates, is

$$\begin{bmatrix} 3/5 & 4/5 \\ -4/5 & 3/5 \end{bmatrix} \begin{bmatrix} 4 \\ 2 \end{bmatrix} = \begin{bmatrix} 4 \\ -2 \end{bmatrix}. \tag{10.39}$$

Verify for yourself that these really *are* the **u**-coordinates of **v**, that is, that the vector **v** really is the same as $4\mathbf{u}_1 + (-2)\mathbf{u}_2$.

10.3.7 Matrix Properties and the Singular Value Decomposition

Because matrices are so closely tied to linear transformations, and because linear transformations are so important in graphics, we'll now briefly discuss some important properties of matrices.

First, **diagonal** matrices—ones with zeroes everywhere except on the diagonal, like the matrix $\mathbf{M}_2$ for the transformation T_2—correspond to remarkably simple transformations: They just scale up or down each axis by some amount (although if the amount is a negative number, the corresponding axis is also flipped). Because of this simplicity, we'll try to understand other transformations in terms of these diagonal matrices.

Second, if the columns of the matrix **M** are $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k \in R^n$, and they are pairwise orthogonal unit vectors, then $\mathbf{M}^\mathbf{T}\mathbf{M} = \mathbf{I}_k$, the $k \times k$ identity matrix.

In the special case where $k = n$, such a matrix is called **orthogonal.** If the determinant of the matrix is 1, then the matrix is said to be a **special orthogonal** matrix. In $\mathbf{R}^\mathbf{2}$, such a matrix must be a rotation matrix like the one in T_1; in $\mathbf{R}^\mathbf{3}$, the transformation associated to such a matrix corresponds to rotation around some vector by some amount.[1]

Less familiar to most students, but of enormous importance in much graphics research, is the **singular value decomposition (SVD)** of a matrix. Its existence says, informally, that if we have a transformation T represented by a matrix **M**, and if we're willing to use new coordinate systems on both the domain and codomain, then the transformation simply looks like a nonuniform (or possibly uniform) scaling transformation. We'll briefly discuss this idea here, along with the application of the SVD to solving equations; the web materials for this chapter show the SVD for our example transformations and some further applications of the SVD.

The singular value decomposition theorem says this:

Every $n \times k$ matrix **M** can be factored in the form

$$\mathbf{M} = \mathbf{U}\mathbf{D}\mathbf{V}^\mathbf{T}, \tag{10.40}$$

where **U** is $n \times r$ (where $r = \min(n, k)$) with orthonormal columns, **D** is $r \times r$ diagonal (i.e., only entries of the form d_{ii} can be nonzero), and **V** is $r \times k$ with orthonormal columns (see Figure 10.8).

By convention, the entries of **D** are required to be in nonincreasing order (i.e., $|d_{1,1}| \geq |d_{2,2}| \geq |d_{3,3}| \ldots$) and are indicated by single subscripts (i.e., we write d_1 instead of $d_{1,1}$). They are called the **singular values** of **M**. It turns out that M is degenerate (i.e., singular) exactly if any singular value is 0. As a general

1. As we mentioned in Chapter 3, rotation about a vector in $\mathbf{R}^\mathbf{3}$ is better expressed as rotation *in a plane,* so instead of speaking about rotation about z, we speak of rotation in the xy-plane. We can then say that any special orthogonal matrix in $\mathbf{R}^4$ corresponds to a sequence of two rotations in two planes in 4-space.

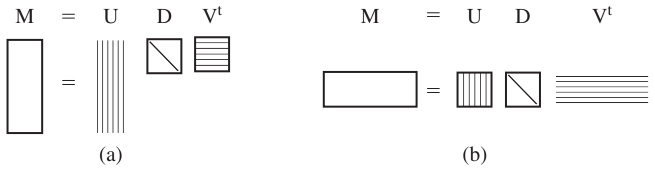


Figure 10.8: (a) An $n \times k$ matrix, with $n > k$, factors as a product of an $n \times n$ matrix with orthonormal columns (indicated by the vertical stripes on the first rectangle), a diagonal $k \times k$ matrix, and a $k \times k$ matrix with orthonormal rows (indicated by the horizontal stripes), which we write as $\mathbf{UDV^T}$, where **U** *and* **V** *have orthonormal* columns. *(b) An $n \times k$ matrix with $n < k$ is written as a similar product; note that the diagonal matrix in both cases is square, and its size is the smaller of n and k.*

guideline, if the ratio of the largest to the smallest singular values is very large (say, 10^6), then numerical computations with the matrix are likely to be unstable.

Inline Exercise 10.16: The singular value decomposition is not unique. If we negate the first row of $\mathbf{V^T}$ and the first column of **U** in the SVD of a matrix **M**, show that the result is still an SVD for **M**.

In the special case where $n = k$ (the one we most often encounter), the matrices **U** and **V** are both square and represent change-of-coordinate transformations in the domain and codomain. Thus, we can see the transformation

$$T(\mathbf{x}) = \mathbf{Mx} \tag{10.41}$$

as a sequence of three steps: (1) Multiplication by $\mathbf{V^T}$ converts **x** to **v**-coordinates; (2) multiplication by **D** amounts to a possibly nonuniform scaling along each axis; and (3) multiplication by **U** treats the resultant entries as coordinates in the **u**-coordinate system, which then are transformed back to standard coordinates.

10.3.8 Computing the SVD

How do we find **U**, **D**, and **V**? In general it's relatively difficult, and we rely on numerical linear algebra packages to do it for us. Furthermore, the results are by no means unique: A single matrix may have multiple singular value decompositions. For instance, if **S** is *any* $n \times n$ matrix with orthonormal columns, then

$$\mathbf{I} = \mathbf{SIS^T} \tag{10.42}$$

is one possible singular value decomposition of the identity matrix. Even though there are many possible SVDs, the singular values are the same for all decompositions.

The **rank** of the matrix **M**, which is defined as the number of linearly independent columns, turns out to be exactly the number of nonzero singular values.

10.3.9 The SVD and Pseudoinverses

Again, in the special case where $n = k$ so that **U** and **V** are square, it's easy to compute $\mathbf{M}^{-1}$ if you know the SVD:

$$\mathbf{M}^{-1} = \mathbf{V}D^{-1}\mathbf{U^T}, \tag{10.43}$$

where D^{-1} is easy to compute—you simply invert all the elements of the diagonal. If one of these elements is zero, the matrix is singular and no such inverse exists; in this case, the **pseudoinverse** is also often useful. It's defined as

$$\mathbf{M}^{\dagger} = \mathbf{V}D^{\dagger}\mathbf{U}^{\mathbf{T}}, \tag{10.44}$$

where $D^{\dagger}$ is just D with every nonzero entry inverted (i.e., you try to invert the diagonal matrix D by inverting diagonal elements, and every time you encounter a zero on the diagonal, you ignore it and simply write down 0 in the answer). The definition of the pseudoinverse makes sense even when $n \neq k$; the pseudoinverse can be used to solve "least squares" problems, which frequently arise in graphics.

The Pseudoinverse Theorem:

(a) If $\mathbf{M}$ is an $n \times k$ matrix with $n > k$, the equation $\mathbf{Mx} = \mathbf{b}$ generally represents an overdetermined system of equations[2] which may have no solution. The vector

$$\mathbf{x}_0 = \mathbf{M}^{\dagger}\mathbf{b} \tag{10.45}$$

represents an optimal "solution" to this system, in the sense that $\mathbf{Mx}_0$ is as close to $\mathbf{b}$ as possible.

(b) If $\mathbf{M}$ is an $n \times k$ matrix with $n < k$, and rank n, the equation $\mathbf{Mx} = \mathbf{b}$ represents an underdetermined system of equations.[3] The vector

$$\mathbf{x}_0 = \mathbf{M}^{\dagger}\mathbf{b} \tag{10.46}$$

represents an optimal solution to this system, in the sense that $\mathbf{x}_0$ is the *shortest* vector satisfying $\mathbf{Mx} = \mathbf{b}$.

Here are examples of each of these cases.

Example 1: An overdetermined system

The system

$$\begin{bmatrix} 2 \\ 1 \end{bmatrix} \begin{bmatrix} t \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \end{bmatrix} \tag{10.47}$$

has *no* solution: There's simply no number t with $2t = 4$ and $1t = 3$ (see Figure 10.9). But among all the multiples of $\mathbf{M} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$, there *is* one that's closest to the vector $\mathbf{b} = \begin{bmatrix} 4 \\ 3 \end{bmatrix}$, namely $2.2 \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 4.4 \\ 2.2 \end{bmatrix}$, as you can discover with elementary geometry. The theorem tells us we can compute this directly, however, using the pseudoinverse. The SVD and pseudoinverse of $\mathbf{M}$ are

$$\mathbf{M} = \mathbf{UDV}^{\mathbf{T}} = \left(\frac{1}{\sqrt{5}} \begin{bmatrix} 2 \\ 1 \end{bmatrix}\right) \begin{bmatrix} \sqrt{5} \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix} \tag{10.48}$$

$$\mathbf{M}^{\dagger} = \mathbf{VD}^{\dagger}\mathbf{U} = \begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} 1/\sqrt{5} \end{bmatrix} \left(\frac{1}{\sqrt{5}} \begin{bmatrix} 2 & 1 \end{bmatrix}\right) \tag{10.49}$$

$$= \begin{bmatrix} 0.4 & 0.2 \end{bmatrix}. \tag{10.50}$$

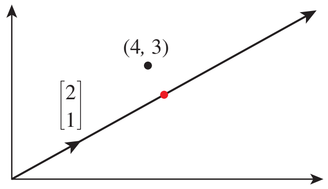


Figure 10.9: The equations $t \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \end{bmatrix}$ *have no common solution. But the multiples of the vector* $\begin{bmatrix} 2 & 1 \end{bmatrix}^{\mathbf{T}}$ *form a line in the plane that passes by the point* $(4, 3)$*, and there's a point of this line (shown in a red circle on the topmost arrow) that's as close to* $(4, 3)$ *as possible.*

2. In other words, a situation like "five equations in three unknowns."
3. That is, a situation like "three equations in five unknowns."

And the solution guaranteed by the theorem is

$$t = \mathbf{M}^\dagger \mathbf{b} = \begin{bmatrix} 0.4 & 0.2 \end{bmatrix} \begin{bmatrix} 4 \\ 3 \end{bmatrix} = 2.2. \tag{10.51}$$

Example 2: An underdetermined system
The system

$$\begin{bmatrix} 1 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = 4 \tag{10.52}$$

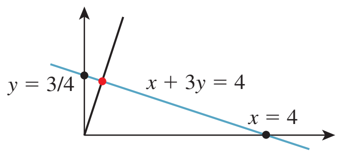


Figure 10.10: Any point of the blue line is a solution; the red point is closest to the origin.

has a great many solutions; any point (x, y) on the line $x + 3y = 4$ is a solution (see Figure 10.10). The solution that's *closest to the origin* is the point on the line $x + 3y = 4$ that's as near to $(0, 0)$ as possible, which turns out to be $x = 0.4; y = 1.2$. In this case, the matrix $\mathbf{M}$ is $\begin{bmatrix} 1 & 3 \end{bmatrix}$; its SVD and pseudoinverse are simply

$$\mathbf{M} = \mathbf{UDV^T} = \begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} \sqrt{10} \end{bmatrix} \begin{bmatrix} 1/\sqrt{10} & 3/\sqrt{10} \end{bmatrix} \text{ and} \tag{10.53}$$

$$\mathbf{M}^\dagger = \mathbf{VD}^\dagger \mathbf{U} = \begin{bmatrix} 1/\sqrt{10} \\ 3/\sqrt{10} \end{bmatrix} \begin{bmatrix} 1/\sqrt{10} \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix} = \begin{bmatrix} 1/10 \\ 3/10 \end{bmatrix}. \tag{10.54}$$

And the solution guaranteed by the theorem is

$$\mathbf{M}^\dagger \mathbf{b} = \begin{bmatrix} 1/10 \\ 3/10 \end{bmatrix} \begin{bmatrix} 4 \end{bmatrix} = \begin{bmatrix} 0.4 \\ 1.2 \end{bmatrix}. \tag{10.55}$$

Of course, this kind of computation is much more interesting in the case where the matrices are much larger, but all the essential characteristics are present even in these simple examples.

A particularly interesting example arises when we have, for instance, two polyhedral models (consisting of perhaps hundreds of vertices joined by triangular faces) that might be "essentially identical": One might be just a translated, rotated, and scaled version of the other. In Section 10.4, we'll see how to represent translation along with rotation and scaling in terms of matrix multiplication. We can determine whether the two models are in fact essentially identical by listing the coordinates of the first in the columns of a matrix $\mathbf{V}$ and the coordinates of the second in a matrix $\mathbf{W}$, and then seeking a matrix $\mathbf{A}$ with

$$\mathbf{AV} = \mathbf{W}. \tag{10.56}$$

This amounts to solving the "overconstrained system" problem; we find that $\mathbf{A} = \mathbf{V}^\dagger \mathbf{W}$ is the best possible solution. If, having computed $\mathbf{A}$, we find that

$$\mathbf{AV} = \mathbf{W}, \tag{10.57}$$

then the models are essentially identical; if the left and right sides differ, then the models are not essentially identical. (This entire approach depends, of course, on corresponding vertices of the two models being listed in the corresponding order; the more general problem is a lot more difficult.)

10.4 Translation

We now describe a way to apply linear transformations to generate *translations,* and at the same time give a nice model for the points-versus-vectors ideas we've espoused so far.

The idea is this: As our Euclidean plane (our set of *points*), we'll take the plane $w = 1$ in xyw-space (see Figure 10.11). The use of w here is in preparation for what we'll do in 3-space, which is to consider the three-dimensional set defined by $w = 1$ in $xyzw$-space.

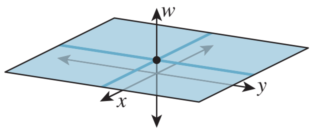


Figure 10.11: The $w = 1$ plane in xyw-space.

Having done this, we can consider transformations that multiply such vectors by a 3×3 matrix $\mathbf{M}$. The only problem is that the result of such a multiplication may not have a 1 as its last entry. We can restrict our attention to those that do:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ p & q & r \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}. \tag{10.58}$$

For this equation to hold for every x and y, we must have $px + qy + r = 1$ for all x, y. This forces $p = q = 0$ and $r = 1$.

Thus, we'll consider transformations of the form

$$\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}. \tag{10.59}$$

If we examine the special case where the upper-left corner is a 2×2 identity matrix, we get

$$\begin{bmatrix} 1 & 0 & c \\ 0 & 1 & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x+c \\ y+f \\ 1 \end{bmatrix}. \tag{10.60}$$

As long as we pay attention only to the x- and y-coordinates, this looks like a translation! We've added c to each x-coordinate and f to each y-coordinate (see Figure 10.12). Transformations like this, restricted to the plane $w = 1$, are called **affine transformations** of the plane. Affine transformations are the ones most often used in graphics.

On the other hand, if we make $c = f = 0$, then the third coordinate becomes irrelevant, and the upper-left 2×2 matrix can perform any of the operations we've seen up until now. Thus, with the simple trick of adding a third coordinate and requiring that it always be 1, we've managed to unify rotation, scaling, and all the other linear transformations with the new class of transformations, *translations,* to get the class of affine transformations.

10.5 Points and Vectors Again

Back in Chapter 7, we said that points and vectors could be combined in certain ways: The difference of points is a vector, a vector could be added to a point

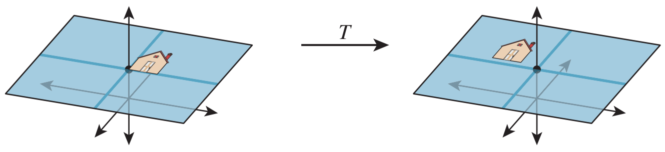


Figure 10.12: The house figure, before and after a translation generated by shearing *parallel to the $w = 1$ plane.*

to get a new point, and more generally, affine combinations of points, that is, combinations of the form

$$\alpha_1 P_1 + \alpha_2 P_2 + \ldots + \alpha_k P_k, \tag{10.61}$$

were allowed if and only if $\alpha_1 + \alpha_2 + \ldots + \alpha_k = 1$.

We now have a situation in which these distinctions make sense in terms of familiar mathematics: We can regard *points* of the plane as being elements of $\mathbf{R}^3$ whose third coordinate is 1, and *vectors* as being elements of $\mathbf{R}^3$ whose third coordinate is 0.

With this convention, it's clear that the difference of points is a vector, the sum of a vector and a point is a point, and combinations like the one in Equation 10.61 yield a point if and only if the sum of the coefficients is 1 (because the third coordinate of the result will be exactly the sum of the coefficients; for the sum to be a *point,* this third coordinate is required to be 1).

You may ask, "Why, when we're already familiar with vectors in 3-space, should we bother calling some of them 'points in the Euclidean plane' and others 'two-dimensional vectors'?" The answer is that the distinctions have geometric significance when we're using this subset of 3-space as a model for 2D transformations. Adding vectors in 3-space is defined in linear algebra, but adding together two of our "points" gives a location in 3-space that's not on the $w = 1$ plane or the $w = 0$ plane, so we don't have a name for it at all.

Henceforth we'll use E^2 (for "Euclidean two-dimensional space") to denote this $w = 1$ plane in *xyw*-space, and we'll write (x, y) to mean the point of E^2 corresponding to the 3-space vector $\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$. It's conventional to speak of an affine transformation as acting on E^2, even though it's defined by a 3×3 matrix.

10.6 Why Use 3 × 3 Matrices Instead of a Matrix and a Vector?

Students sometimes wonder why they can't just represent a linear transformation plus translation in the form

$$T(\mathbf{x}) = \mathbf{Mx} + \mathbf{b}, \tag{10.62}$$

where the matrix **M** represents the linear part (rotating, scaling, and shearing) and **b** represents the translation.

First, you *can* do that, and it works just fine. You might save a tiny bit of storage (four numbers for the matrix and two for the vector, so six numbers instead of nine), but since our matrices always have two 0s and a 1 in the third column, we don't really need to store that column anyhow, so it's the same. Otherwise, there's no important difference.

Second, the reason to unify the transformations into a single matrix is that it's then very easy to take multiple transformations (each represented by a matrix) and **compose** them (perform one after another): We just multiply their matrices together in the right order to get the matrix for the composed transformation. You can do this in the matrix-and-vector formulation as well, but the programming is slightly messier and more error-prone.

There's a third reason, however: It'll soon become apparent that we can also work with triples whose third entry is neither 1 nor 0, and use the operation of **homogenization** (dividing by w) to convert these to points (i.e., triples with $w = 1$), except when $w = 0$. This allows us to study even more transformations, one of which is central to the study of perspective, as we'll see later.

The singular value decomposition provides the tool necessary to decompose not just linear transformations, but affine ones as well (i.e., combinations of linear transformations and translations).

10.7 Windowing Transformations

As an application of our new, richer set of transformations, let's examine **windowing transformations,** which send one axis-aligned rectangle to another, as shown in Figure 10.13. (We already discussed this briefly in Chapter 3.)

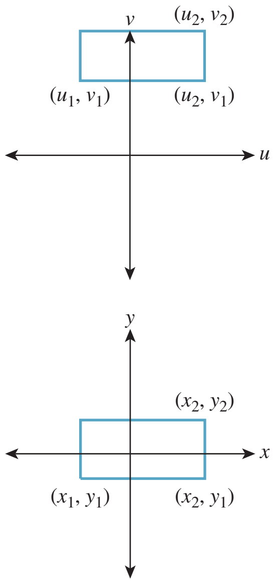


Figure 10.13: Window transformation setup. We need to move the uv-rectangle to the xy-rectangle.

We'll first take a direct approach involving a little algebra. We'll then examine a more automated approach.

We'll need to do essentially the same thing to the first and second coordinates, so let's look at how to transform the first coordinate only. We need to send u_1 to x_1 and u_2 to x_2. That means we need to scale up any coordinate *difference* by the factor $\frac{x_2-x_1}{u_2-u_1}$. So our transformation for the first coordinate has the form

$$t \mapsto \frac{x_2 - x_1}{u_2 - u_1} t + \text{something}. \tag{10.63}$$

If we apply this to $t = u_1$, we know that we want to get x_1; this leads to the equation

$$\frac{x_2 - x_1}{u_2 - u_1} u_1 + \text{something} = x_1. \tag{10.64}$$

Solving for the missing offset gives

$$\begin{aligned} x_1 - \frac{x_2 - x_1}{u_2 - u_1} u_1 &= x_1 \frac{u_2 - u_1}{u_2 - u_1} - \frac{x_2 - x_1}{u_2 - u_1} u_1 && (10.65) \\ &= \frac{x_1 u_2 - x_1 u_1 - x_2 u_1 + x_1 u_1}{u_2 - u_1} && (10.66) \\ &= \frac{x_1 u_2 - x_2 u_1}{u_2 - u_1}, && (10.67) \end{aligned}$$

so that the transformation is

$$t \mapsto \frac{x_2 - x_1}{u_2 - u_1} t + \frac{x_1 u_2 - x_2 u_1}{u_2 - u_1}. \tag{10.68}$$

Doing essentially the same thing for the v and y terms (i.e., the second coordinate) we get the transformation, which we can write in matrix form:

$$T(\mathbf{x}) = \mathbf{M}\mathbf{x}, \tag{10.69}$$

where

$$\mathbf{M} = \begin{bmatrix} \frac{x_2-x_1}{u_2-u_1} & 0 & \frac{x_1u_2-x_2u_1}{u_2-u_1} \\ 0 & \frac{y_2-y_1}{v_2-v_1} & \frac{y_1v_2-y_2v_1}{v_2-v_1} \\ 0 & 0 & 1 \end{bmatrix}. \tag{10.70}$$

Inline Exercise 10.17: Multiply the matrix $\mathbf{M}$ of Equation 10.70 by the vector $\begin{bmatrix} u_1 & v_1 & 1 \end{bmatrix}^{\mathbf{T}}$ to confirm that you do get $\begin{bmatrix} x_1 & y_1 & 1 \end{bmatrix}^{\mathbf{T}}$. Do the same for the opposite corner of the rectangle.

We'll now show you a second way to build this transformation (and many others as well).

10.8 Building 3D Transformations

Recall that in 2D we could send the vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ to the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ by building a matrix $\mathbf{M}$ whose columns were $\mathbf{v}_1$ and $\mathbf{v}_2$, and then use two such matrices (inverting one along the way) to send any two independent vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ to any two vectors $\mathbf{w}_1$ and $\mathbf{w}_2$. We can do the same thing in 3-space: We can send the standard basis vectors $\mathbf{e}_1, \mathbf{e}_2$, and $\mathbf{e}_3$ to any three other vectors, just by using those vectors as the columns of a matrix. Let's start by sending $\mathbf{e}_1$, $\mathbf{e}_2$, and $\mathbf{e}_3$ to three corners of our first rectangle—the two we've already specified and the lower-right one, at location (u_2, v_1). The three vectors corresponding to these points are

$$\begin{bmatrix} u_1 \\ v_1 \\ 1 \end{bmatrix}, \quad \begin{bmatrix} u_2 \\ v_2 \\ 1 \end{bmatrix}, \text{ and } \begin{bmatrix} u_2 \\ v_1 \\ 1 \end{bmatrix}. \tag{10.71}$$

Because the three corners of the rectangle are not collinear, the three vectors are independent. Indeed, this is our definition of independence for vectors in n-space: Vectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$ are independent if there's no $(k-1)$-dimensional subspace containing them. In 3-space, for instance, three vectors are independent if there's no plane through the origin containing all of them.

So the matrix

$$\mathbf{M}_1 = \begin{bmatrix} u_1 & u_2 & u_2 \\ v_1 & v_2 & v_1 \\ 1 & 1 & 1 \end{bmatrix}, \tag{10.72}$$

which performs the desired transformation, will be invertible.

We can similarly build the matrix $\mathbf{M}_2$, with the corresponding xs and ys in it. Finally, we can compute

$$\mathbf{M}_2\mathbf{M}_1^{-1}, \tag{10.73}$$

which will perform the desired transformation. For instance, the lower-left corner of the starting rectangle will be sent, by $\mathbf{M}_1^{-1}$, to $\mathbf{e}_1$ (because $\mathbf{M}_1$ sent $\mathbf{e}_1$ to the lower-left corner); multiplying $\mathbf{e}_1$ by $\mathbf{M}_2$ will send it to the lower-left corner of the target rectangle. A similar argument applies to all three corners. Indeed, if we compute the inverse algebraically and multiply out everything, we'll once again arrive at the matrix given in Equation 10.7. But we don't need to do so: We know that this must be the right matrix. Assuming we're willing to use a matrix-inversion routine, there's no need to think through anything more than "I want these three points to be sent to these three other points."

Summary: Given any three noncollinear points P_1, P_2, P_3 in E^2, we can find a matrix transformation and send them to any three points Q_1, Q_2, Q_3 with the procedure above.

10.9 Another Example of Building a 2D Transformation

Suppose we want to find a 3×3 matrix transformation that rotates the entire plane 30° counterclockwise around the point $P = (2, 4)$, as shown in Figure 10.14. As you'll recall, WPF expresses this transformation via code like this:

```
<RotateTransform Angle="-30" CenterX="2" CenterY="4"/>
```

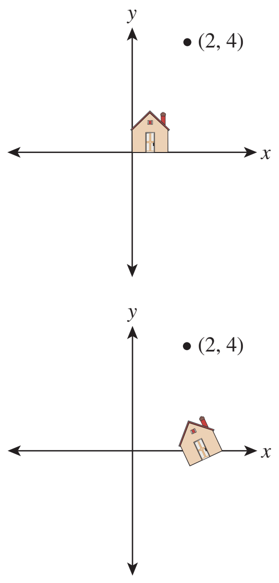


Figure 10.14: We'd like to rotate the entire plane by 30° counterclockwise about the point $P = (2, 4)$.

An implementer of WPF then must create a matrix like the one we're about to build.

Here are two approaches.

First, we know how to rotate about the origin by 30°; we can use the transformation T_1 from the start of the chapter. So we can do our desired transformation in three steps (see Figure 10.15).

1. Move the point $(2, 4)$ to the origin.
2. Rotate by 30°.
3. Move the origin back to $(2, 4)$.

The matrix that moves the point $(2, 4)$ to the origin is

$$\begin{bmatrix} 1 & 0 & -2 \\ 0 & 1 & -4 \\ 0 & 0 & 1 \end{bmatrix}. \tag{10.74}$$

The one that moves it back is similar, except that the 2 and 4 are not negated. And the rotation matrix (expressed in our new 3×3 format) is

$$\begin{bmatrix} \cos 30^\circ & -\sin 30^\circ & 0 \\ \sin 30^\circ & \cos 30^\circ & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{10.75}$$

The matrix representing the entire sequence of transformations is therefore

$$\begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos 30^\circ & -\sin 30^\circ & 0 \\ \sin 30^\circ & \cos 30^\circ & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -2 \\ 0 & 1 & -4 \\ 0 & 0 & 1 \end{bmatrix}. \tag{10.76}$$

Inline Exercise 10.18: (a) Explain why this is the correct order in which to multiply the transformations to get the desired result.
(b) Verify that the point $(2, 4)$ is indeed left unmoved by multiplying $\begin{bmatrix} 2 & 4 & 1 \end{bmatrix}^{\mathbf{T}}$ by the sequence of matrices above.

The second approach is again more automatic: We find three points whose target locations we know, just as we did with the windowing transformation above. We'll use $P = (2, 4)$, $Q = (3, 4)$ (the point one unit to the right of P), and $R = (2, 5)$ (the point one unit above P). We know that we want P sent to P, Q sent to $(2 + \cos 30^\circ, 4 + \sin 30^\circ)$, and R sent to $(2 - \sin 30^\circ, 4 + \cos 30^\circ)$. (Draw a picture to convince yourself that these are correct). The matrix that achieves this is just

$$\begin{bmatrix} 2 & 2+\cos 30^\circ & 4-\sin 30^\circ \\ 4 & 4+\sin 30^\circ & 4+\cos 30^\circ \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 3 & 2 \\ 4 & 4 & 5 \\ 1 & 1 & 1 \end{bmatrix}^{-1}. \tag{10.77}$$

Both approaches are reasonably easy to work with.

There's a third approach—a variation of the second—in which we specify where we want to send a point and two vectors, rather than three points. In this case, we might say that we want the point P to remain fixed, and the vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ to go to

$$\begin{bmatrix} \cos 30^\circ \\ \sin 30^\circ \\ 0 \end{bmatrix} \text{ and } \begin{bmatrix} -\sin 30^\circ \\ \cos 30^\circ \\ 0 \end{bmatrix}, \tag{10.78}$$

respectively. In this case, instead of finding matrices that send the vectors $\mathbf{e}_1, \mathbf{e}_2$, and $\mathbf{e}_3$ to the desired three points, before and after, we find matrices that send those vectors to the desired point and two vectors, before and after. These matrices are

$$\begin{bmatrix} 2 & 1 & 0 \\ 4 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \text{ and } \begin{bmatrix} 2 & \cos 30^\circ & -\sin 30^\circ \\ 4 & \sin 30^\circ & \cos 30^\circ \\ 1 & 0 & 0 \end{bmatrix}, \tag{10.79}$$

so the overall matrix is

$$\begin{bmatrix} 2 & \cos 30^\circ & -\sin 30^\circ \\ 4 & \sin 30^\circ & \cos 30^\circ \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 4 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}^{-1}. \tag{10.80}$$

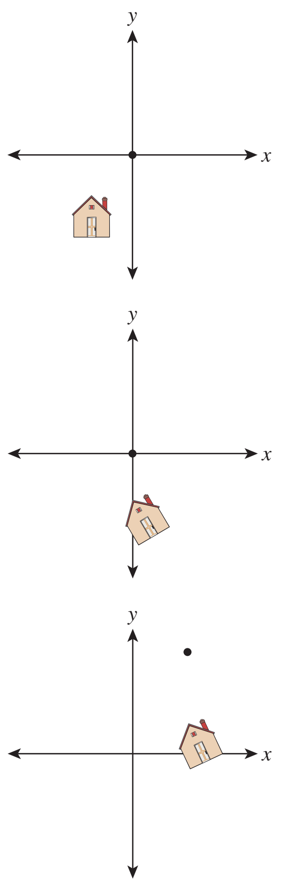


Figure 10.15: The house after translating $(2,4)$ to the origin, after rotating by 30°, and after translating the origin back to $(2,4)$.

These general techniques can be applied to create any linear-plus-translation transformation of the $w = 1$ plane, but there are some specific ones that are good to know. Rotation in the xy-plane, by an amount θ (rotating the positive x-axis toward the positive y-axis) is given by

$$R_{xy}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{10.81}$$

In some books and software packages, this is called **rotation around z;** we prefer the term "rotation in the xy-plane" because it also indicates the direction of rotation (from x, toward y). The other two standard rotations are

$$R_{yz}(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \tag{10.82}$$

and

$$R_{zx}(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}; \tag{10.83}$$

note that the last expression rotates z toward x, and *not* the opposite. Using this naming convention helps keep the pattern of plusses and minuses symmetric.

10.10 Coordinate Frames

In 2D, a linear transformation is completely specified by its values on two independent vectors. An affine transformation (i.e., linear plus translation) is completely specified by its values on any three noncollinear points, or on any point and pair of independent vectors. A projective transformation on the plane (which we'll discuss briefly in Section 10.13) is specified by its values on four points, no three collinear, or on other possible sets of points and vectors. These facts, and the corresponding ones for transformations on 3-space, are so important that we enshrine them in a principle:

✓ **THE TRANSFORMATION UNIQUENESS PRINCIPLE:** For each class of transformations—linear, affine, and projective—and any corresponding coordinate frame, and any set of corresponding target elements, there's a unique transformation mapping the frame elements to the correponding elements in the target frame. If the target elements themselves constitute a frame, then the transformation is invertible.

To make sense of this, we need to define a **coordinate frame.** As a first example, a coordinate frame for linear transformations is just a "basis": In two dimensions, that means "two linearly independent vectors in the plane." The elements of the frame are the two vectors. So the principle says that if $\mathbf{u}$ and $\mathbf{v}$ are linearly independent vectors in the plane, and $\mathbf{u}'$ and $\mathbf{v}'$ are any two vectors, then there's a unique linear transformation sending $\mathbf{u}$ to $\mathbf{u}'$ and $\mathbf{v}$ to $\mathbf{v}'$. It further says that if $\mathbf{u}'$ and $\mathbf{v}'$ are independent, then the transformation is invertible.

More generally, a **coordinate frame** is a set of geometric elements rich enough to uniquely characterize a transformation in some class. For linear transformations of the plane, a coordinate frame consists of two independent vectors in the plane, as we said; for affine transforms of the plane, it consists of three noncollinear points in the plane, *or* of one point and two independent vectors, etc.

In cases where there are multiple kinds of coordinate frames, there's always a way to convert between them. For 2D affine transformations, the three noncollinear points P, Q, and R can be converted to $P, \mathbf{v}_1 = Q - P$, and $\mathbf{v}_2 = R - P$; the conversion in the other direction is obvious. (It may not be obvious that the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ are linearly independent. See Exercise 10.4.)

There's a restricted use of "coordinate frame" for affine maps that has some advantages. Based on the notion that the origin and the unit vectors along the positive directions for each axis form a frame, we'll say that a **rigid coordinate frame** for the plane is a triple $(P, \mathbf{v}_1, \mathbf{v}_2)$, where P is a point and $\mathbf{v}_1$ and $\mathbf{v}_2$ are *perpendicular* unit vectors with the rotation from $\mathbf{v}_1$ toward $\mathbf{v}_2$ being counterclockwise (i.e., with $\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{v}_1 = \mathbf{v}_2$). The corresponding definition for 3-space has one point and three mutually perpendicular unit vectors forming a right-hand coordinate system. Transforming one rigid coordinate frame $(P, \mathbf{v}_1, \mathbf{v}_2)$ to another $(Q, \mathbf{u}_1, \mathbf{u}_2)$ can always be effected by a sequence of transformation,

$$T_Q \circ R \circ T_P^{-1}, \tag{10.84}$$

where $T_P(A) = A + P$ is translation by P, and similarly for T_Q, and R is the rotation given by

$$R = [\mathbf{u}_1; \mathbf{u}_2] \cdot [\mathbf{v}_1; \mathbf{v}_2]^{\mathbf{T}}, \tag{10.85}$$

where the semicolon indicates that $\mathbf{u}_1$ is the first column of the first factor, etc.

The G3D library, which we use in examples in Chapters 12, 15, and 32, uses rigid coordinate frames extensively in modeling, encapsulating them in a class, `CFrame`.

10.11 Application: Rendering from a Scene Graph

We've discussed affine transformations on a two-dimensional affine space, and how, once we have a coordinate system and can represent points as triples, as in $\mathbf{x} = \begin{bmatrix} x & y & 1 \end{bmatrix}^{\mathbf{T}}$, we can represent a transformation by a 3×3 matrix $\mathbf{M}$. We transform the point $\mathbf{x}$ by multiplying it on the left by $\mathbf{M}$ to get $\mathbf{Mx}$. With this in mind, let's return to the clock example of Chapter 2 and ask how we could start from a WPF description and convert it to an image, that is, how we'd do some of the work that WPF does. You'll recall that the clock shown in Figure 10.16 was created in WPF with code like this,

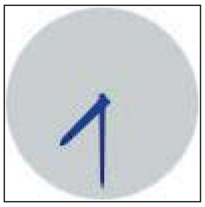


Figure 10.16: Our clock model.

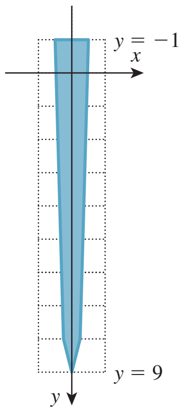


Figure 10.17: The clock-hand template.

```
<Canvas ... >
  <Ellipse
     Canvas.Left="-10.0" Canvas.Top="-10.0"
     Width="20.0" Height="20.0"
     Fill="lightgray" />
  <Control Name="Hour Hand" .../>
  <Control Name="Minute Hand" .../>
  <Canvas.RenderTransform>
     <TransformGroup>
      <ScaleTransform ScaleX="4.8" ScaleY="4.8" />
      <TranslateTransform X="48" Y="48" />
   </TransformGroup>
  </Canvas.RenderTransform>
</Canvas>
```

where the code for the hour hand is

```
<Control Name="HourHand" Template="{StaticResource ClockHandTemplate}">
  <Control.RenderTransform>
    <TransformGroup>
      <ScaleTransform ScaleX="1.7" ScaleY="0.7" />
      <RotateTransform Angle="180"/>
      <RotateTransform x:Name="ActualTimeHour" Angle="0"/>
    </TransformGroup>
  </Control.RenderTransform>
</Control>
```

and the code for the minute hand is similar, the only differences being that `ActualTimeHour` is replaced by `ActualTimeMinute` and the scale by 1.7 in X and 0.7 in Y is omitted.

The `ClockHandTemplate` was a polygon defined by five points in the plane: $(-0.3, -1)$, $(-0.2, 8)$, $(0, 9)$, $(0.2, 8)$, and $(0.3, -1)$ (see Figure 10.17).

We're going to slightly modify this code so that the clock face and clock hands are both described in the same way, as polygons. We *could* create a polygonal version of the circular face by making a regular polygon with, say, 1000 vertices, but to keep the code simple and readable, we'll make an octagonal approximation of a circle instead.

Now the code begins like this:

```
<Canvas ...
    <Canvas.Resources>
      <ControlTemplate x:Key="ClockHandTemplate">
        <Polygon
          Points="-0.3,-1    -0.2,8    0,9    0.2,8    0.3,-1"
           Fill="Navy"/>
      </ControlTemplate>
      <ControlTemplate x:Key="CircleTemplate">
        <Polygon
           Points="1,0    0.707,0.707    0,1     -.707,.707
                  -1,0    -.707,-.707    0,-1    0.707,-.707"
           Fill="LightGray"/>
      </ControlTemplate>
    </Canvas.Resources>
```

This code defines the geometry that we'll use to create the face and hands of the clock. With this change, the circular clock face will be defined by transforming a template "circle," represented by eight evenly spaced points on the unit circle. This form of specification, although not idiomatic in WPF, is quite similar to scene specification in many other scene-graph packages.

The actual creation of the scene now includes building the clock face from the `CircleTemplate`, and building the hands as before.

```
  <!- 1. Background of the clock ->
  <Control Name="Face"
           Template="{StaticResource CircleTemplate}">
     <Control.RenderTransform>
           <ScaleTransform ScaleX="10" ScaleY="10" />
     </Control.RenderTransform>
  </Control>

  <!- 2. The minute hand ->
  <Control Name="MinuteHand"
           Template="{StaticResource ClockHandTemplate}">
     <Control.RenderTransform>
        <TransformGroup>
           <RotateTransform Angle="180" />
           <RotateTransform x:Name="ActualTimeMinute" Angle="0" />
        </TransformGroup>
     </Control.RenderTransform>
  </Control>

  <!- 3. The hour hand ->
  <Control Name="HourHand" Template="{StaticResource ClockHandTemplate}">
     <Control.RenderTransform>
        <TransformGroup>
          <ScaleTransform ScaleX="1.7" ScaleY="0.7" />
          <RotateTransform Angle="180" />
          <RotateTransform x:Name="ActualTimeHour"
                           Angle="0" />
        </TransformGroup>
     </Control.RenderTransform>
  </Control>
```

All that remains is the transformation from `Canvas` to WPF coordinates, and the timers for the animation, which set the `ActualTimeMinute` and `ActualTimeHour` values.

```
  <Canvas.RenderTransform>
    ...same as before...
   </Canvas.RenderTransform>

  <Canvas.Triggers>
    <EventTrigger RoutedEvent="FrameworkElement.Loaded">
       <BeginStoryboard>
          <Storyboard>
            <DoubleAnimation
              Storyboard.TargetName="ActualTimeHour"
              Storyboard.TargetProperty="Angle"
              From="0.0" To="360.0"
              Duration="00:00:01:0" RepeatBehavior="Forever"
              />
            <DoubleAnimation
              Storyboard.TargetName="ActualTimeMinute"
              Storyboard.TargetProperty="Angle"
              From="0.0" To="4320.0"
              Duration="00:00:01:0" RepeatBehavior="Forever"
              />
          </Storyboard>
       </BeginStoryboard>
    </EventTrigger>
  </Canvas.Triggers>

</Canvas>
```

As a starting point in transforming this scene description into an image, we'll assume that we have a basic graphics library that, given an array of points representing a polygon, can draw that polygon. The points will be represented by a $3 \times k$ array of homogeneous coordinate triples, so the first column of the array will be the homogeneous coordinates of the first polygon point, etc.

We'll now explain how we can go from something like the WPF description to a sequence of `drawPolygon` calls. First, let's transform the XAML code into a tree structure, as shown in Figure 10.18, representing the scene graph (see Chapter 6).

We've drawn transformations as diamonds, geometry as blue boxes, and named parts as beige boxes. For the moment, we've omitted the matter of instancing of the `ClockHandTemplate` and pretended that we have two separate identical copies of the geometry for a clock hand. We've also drawn next to each transformation the matrix representation of the transformation. We've assumed that the angle in `ActualTimeHour` is 15° (whose cosine and sine are approximately 0.96 and 0.26, respectively) and the angle in `ActualTimeMinutes` is 180° (i.e., the clock is showing 12:30).

Inline Exercise 10.19: (a) Remembering that rotations in WPF are specified in degrees and that they rotate objects in a *clockwise* direction, check that the matrix given for the rotation of the hour hand by 15° is correct.
(b) If you found that the matrix was wrong, recall that in WPF x increases to the right and y increases *down*. Does this change your answer? By the way, if you ran this program in WPF and debugged it and printed the matrix, you'd find the negative sign on the $(2,1)$ entry instead of the $(1,2)$ entry. That's because WPF internally uses row vectors to represents points, and multiplies them by transformation matrices on the right.

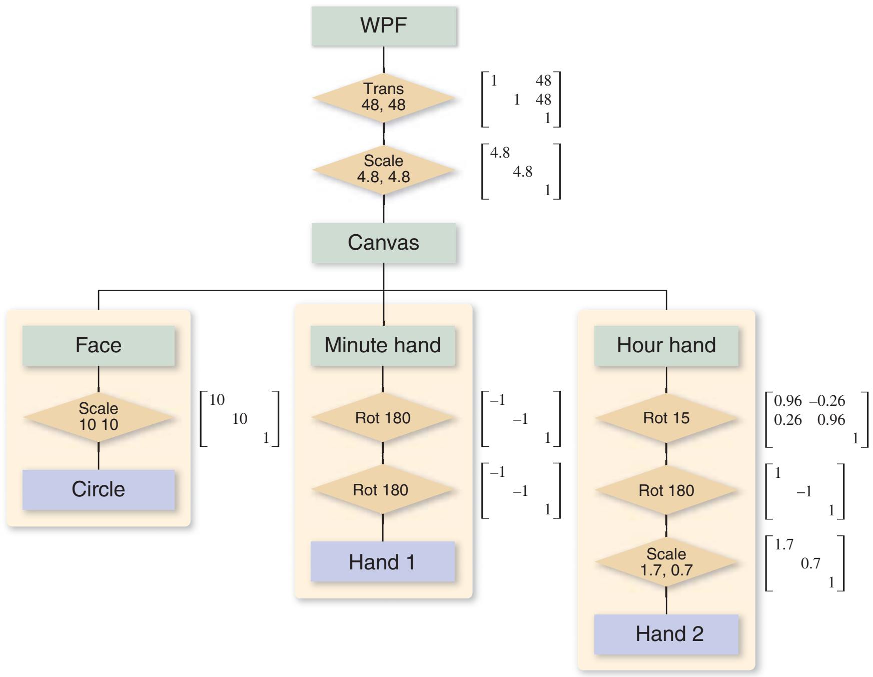


Figure 10.18: A scene-graph representation of the XAML code for the clock.

The order of items in the tree is a little different from the textual order, but there's a natural correspondence between the two. If you consider the hour hand and look at all transformations that occur in its associated render transform or in the render transform of anything containing it (i.e., the whole clock), those are exactly the transforms you encounter as you read from the leaf node corresponding to the hour hand up toward the root node.

Inline Exercise 10.20: Write down all transformations applied to the circle template that's used as the clock face by reading the XAML program. Confirm that they're the same ones you get by reading upward from the "Circle" box in Figure 10.18.

In the scene graph we've drawn, the transformation matrices are the most important elements. We're now going to discuss how these matrices and the coordinates of the points in the geometry nodes interact.

Recall that there are two ways to think about transformations. The first is to say that the minute hand, for instance, has a rotation operation applied to each of its points, creating a new minute hand, which in turn has a translation applied to each point, creating yet another new minute hand, etc. The tip of the minute hand is at location $(0, 9)$, once and for all. The tip of the *rotated* minute hand is somewhere else, and the tip of the translated and rotated minute hand is somewhere else again. It's common to talk about all of these as if they were the same thing ("*Now* the tip of the minute hand is at $(3, 17)\ldots$"), but that doesn't really make sense—the tip of the minute hand cannot be in two different places.

The second view says that there are several different coordinate systems, and that the transformations tell you how to get from the tip's coordinates in one system to its coordinates in another. We can then say things like, "The tip of the minute hand is at $(0,9)$ in **object space** or **object coordinates,** but it's at $(0,-9)$ in canvas coordinates." Of course, the position in canvas coordinates depends on the amount by which the tip of the minute hand is rotated (we've assumed that the `ActualTimeMinute` rotation is 180°, so it has just undergone two 180° rotations). Similarly, the WPF coordinates for the tip of the minute hand are computed by further scaling each canvas coordinate by 4.8, and then adding 48 to each, resulting in WPF coordinates of $(48,4.8)$.

The terms **object space, world space, image space,** and **screen space** are frequently used in graphics. They refer to the idea that a single point of some object (e.g., "Boston" on a texture-mapped globe) starts out as a point on a unit sphere (object space), gets transformed into the "world" that we're going to render, eventually is projected onto an image plane, and finally is displayed on a screen. In some sense, all those points refer to the same thing. But each point has different coordinates. When we talk about a certain point "in world space" or "in image space," we really mean that we're working with the coordinates of the point in a coordinate system associated with that space. In image space, those coordinates may range from -1 to 1 (or from 0 to 1 in some systems), while in screen space, they may range from 0 to 1024, and in object space, the coordinates are a triple of real numbers that are typically in the range $[-1,1]$ for many standard objects like the sphere or cube.

For this example, we have seven coordinate systems, most indicated by pale green boxes. Starting at the top, there are WPF coordinates, the coordinates used by `drawPolygon()`. It's possible that internally, `drawPolygon()` must convert to, say, pixel coordinates, but this conversion is hidden from us, and we won't discuss it further. Beneath the WPF coordinates are canvas coordinates, and within the canvas are the clock-face coordinates, minute-hand coordinates, and hour-hand coordinates. Below this are the hand coordinates, the coordinate system in which the single prototype hand was created, and circle coordinates, in which the prototype octagonal circle approximation was created. Notice that in our model of the clock, the clock-face, minute-hand, and hour-hand coordinates all play similar roles: In the hierarchy of coordinate systems, they're all children of the canvas coordinate system. It might also have been reasonable to make the minute-hand and hour-hand coordinate systems children of the clock-face coordinate system. The advantage of doing so would have been that translating the clock face would have translated the whole clock, making it easier to adjust the clock's position on the canvas. Right now, adjusting the clock's position on the canvas requires that we adjust three different translations, which we'd have to add to the face, the minute hand, and the hour hand.

We're hoping to draw each shape with a `drawPolygon()` call, which takes an array of point coordinates as an argument. For this to make sense, we have to declare the coordinate system in which the point coordinates are valid. We'll assume that `drawPolygon()` expects WPF coordinates. So when we want to tell it about the tip of the minute hand, we'll need the numbers $(48,4.8)$ rather than $(0,9)$.

Here's a strawman algorithm for converting a scene graph into a sequence of `drawPolygon()` calls. We'll work with $3 \times k$ arrays of coordinates, because we'll represent the point $(0, 9)$ as a homogeneous triple $(0, 9, 1)$, which we'll write vertically as a column of the matrix that represents the geometry.

```
for each polygonal geometry element, $g$
   let $v$ be the $3 \times k$ array of vertices of $g$
   let $n$ be the parent node of $g$
   let $\mathbf{M}$ be the $3 \times 3$ identity matrix
   while ($n$ is not the root)
      if $n$ is a transformation with matrix $\mathbf{S}$
        $\mathbf{M} = \mathbf{SM}$
      $n$ = parent of $n$

   $w = \mathbf{M}v$
   drawPolygon($w$)
```

As you can see, we multiply together several matrices, and then multiply the result (the **composite transformation matrix**) by the vertex coordinates to get the WPF coordinates for each polygon, which we then draw.

Inline Exercise 10.21: (a) How many elementary operations are needed, approximately, to multiply a 3×3 matrix by a $3 \times k$ matrix?
(b) If $\mathbf{A}$ and $\mathbf{B}$ are 3×3 and $\mathbf{C}$ is 3×1000, would you rather compute $(\mathbf{AB})\mathbf{C}$ or $\mathbf{A}(\mathbf{BC})$, where the parentheses are meant to indicate the order of calculations that you perform?
(c) In the code above, should we have multiplied the vertex coordinates by each matrix in turn, or was it wiser to accumulate the matrix product and only multiply by the vertex array at the end? Why?

If we hand-simulate the code in the clock example, the circle template coordinates are multiplied by the matrix

$$\begin{bmatrix} 1 & 0 & 48 \\ 0 & 1 & 48 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4.8 & 0 & 0 \\ 0 & 4.8 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 10 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{10.86}$$

The minute-hand template coordinates are multiplied by the matrix

$$\begin{bmatrix} 1 & 0 & 48 \\ 0 & 1 & 48 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4.8 & 0 & 0 \\ 0 & 4.8 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{10.87}$$

And the hour-hand template coordinates are multiplied by the matrix

$$\begin{bmatrix} 1 & 0 & 48 \\ 0 & 1 & 48 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4.8 & 0 & 0 \\ 0 & 4.8 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.96 & -0.26 & 0 \\ 0.26 & 0.96 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1.7 & 0 & 0 \\ 0 & 0.7 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{10.88}$$

Inline Exercise 10.22: Explain where each of the matrices for the minute hand arose.

Notice how much of this matrix multiplication is *shared.* We could have computed the product for the circle and reused it in each of the others, for instance. For a large scene graph, the overlap is often much greater. If there are 70 transformations applied to an object with only five or six vertices, the cost of multiplying matrices together far outweighs the cost of multiplying the composite matrix by the vertex coordinate array.

We can avoid duplicated work by revising our strawman algorithm. We perform a depth-first traversal of the scene graph, maintaining a stack of matrices as we do so. Each time we encounter a new transformation with matrix **M**, we multiply **M** by the current transformation matrix **C** (the one at the top of the stack) and push the result, **MC**, onto the stack. Each time our traversal rises up through a transformation node, we pop a matrix from the stack. The result is that whenever we encounter geometry (like the coordinates of the hand points, or of the ellipse points), we can multiply the coordinate array on the left by the current transformation to get the WPF coordinates of those points. In the pseudocode below, we assume that the scene graph is represented by a `Scene` class with a method that returns the root node of the graph, and that a transformation node has a `matrix` method that returns the matrix for the associated transformation, while a geometry node has a `vertexCoordinateArray` method that returns a $3 \times k$ array containing the homogeneous coordinates of the k points in the polygon.

```
void drawScene(Scene myScene)
   s = empty Stack
   s.push( 3 × 3 identity matrix )
   explore(myScene.rootNode(), s)


void explore(Node n, Stack& s)
   if n is a transformation node
     push n.matrix() * s.top() onto s

   else if n is a geometry node
     drawPolygon(s.top() * n.vertexCoordinateArray())

   foreach child k of n
     explore(k, s)

   if n is a transformation node
     pop top element from s
```

In some complex models, the cost of matrix multiplications can be enormous. If the same model is to be rendered over and over, and none of the transformations change (e.g., a model of a building in a driving-simulation game), it's often worth it to use the algorithm above to create a list of polygons in world coordinates that can be redrawn for each frame, rather than reparsing the scene once per frame. This is sometimes referred to as **prebaking** or **baking** a model.

The algorithm above is the core of the standard one used for scene traversals in scene graphs. There are two important additions, however.

First, geometric transformations are not the only things stored in a scene graph—in some cases, attributes like color may be stored as well. In a simple version, each geometry node has a color, and the `drawPolygon` procedure is passed both the vertex coordinate array and the color. In a more complex version, the color attribute may be set at some node in the graph, and that color is used for all the geometry "beneath" that node. In this latter form, we can keep track of the color with a parallel stack onto which colors are pushed as they're encountered, just as transformations are pushed onto the transformation stack. The difference is that while transformations are multiplied by the previous composite transformation before being pushed on the stack, the colors, representing an absolute rather than a relative attribute, are pushed without being combined in any way with previous color settings. It's easy to imagine a scene graph in which color-alteration nodes are allowed (e.g., "Lighten everything below this node by 20%"); in such a structure, the stack would have to accumulate color transformations. Unless the transformations are quite limited, there's no obvious way to combine them except to treat them as a sequence of transformations; matrix transformations are rather special in this regard.

Second, we've studied an example in which the scene graph is a *tree,* but depth-first traversal actually makes sense in an arbitrary directed acyclic graph (DAG). And in fact, our clock model, in reality, *is* a DAG: The geometry for the two clock hands is shared by the hands (using a WPF `StaticResource`). During the depth-first traversal we arrive at the hand geometry twice, and thus render two different hands. For a more complex model (e.g., a scene full of identical robots) such repeated encounters with the same geometry may be very frequent: Each robot has two identical arms that refer to the same underlying arm model; each arm has three identical fingers that refer to the same underlying finger model, etc. It's clear that in such a situation, there's some lost effort in retraversal of the arm model. Doing some analysis of a scene graph to detect such retraversals and avoid them by prebaking can be a useful optimization, although in many of today's graphics applications, scene traversal is only a tiny fraction of the cost, and lighting and shading computations (for 3D models) dominate. You should avoid optimizing the scene-traversal portions of your code until you've verified that they are the expensive part.

10.11.1 Coordinate Changes in Scene Graphs

Returning to the scene graph and the matrix products, the transformations applied to the minute hand to get WPF coordinates,

$$\begin{bmatrix} 1 & 0 & 48 \\ 0 & 1 & 48 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4.8 & 0 & 0 \\ 0 & 4.8 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad (10.89)$$

represent the transformation from minute-hand coordinates to WPF coordinates. To go from WPF coordinates to minute-hand coordinates, we need only apply the inverse transformation. Remembering that $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$, this inverse transformation is

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1/4.8 & 0 & 0 \\ 0 & 1/4.8 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -48 \\ 0 & 1 & -48 \\ 0 & 0 & 1 \end{bmatrix}. \qquad (10.90)$$

You can similarly find the coordinate transformation matrix to get from any one coordinate system in a scene graph to any other. Reading upward, you accumulate the matrices you encounter, with the first matrix being farthest to the right; reading downward, you accumulate their inverses in the opposite order. When we build scene graphs in 3D, exactly the same rules apply.

For a 3D scene, there's the description not only of the model, but also of how to transform points of the model into points on the display. This latter description is provided by specifying a camera. But even in 2D, there's something closely analogous: The `Canvas` in which we created our clock model corresponds to the "world" of a 3D scene; the way that we transform this world to make it appear on the display (scale by $(4.8, 4.8)$ and then translate by $(48, 48)$) corresponds to the viewing transformation performed by a 3D camera.

Typically the polygon coordinates (the ones we've placed in templates) are called modeling coordinates. Given the analogy to 3D, we can call the canvas coordinates world coordinates, while the WPF coordinates can be called image coordinates. These terms are all in common use when discussing 3D scene graphs.

As an exercise, let's consider the tip of the hour hand; in modeling coordinates (i.e., in the clock-hand template) the tip is located at $(0, 9)$. In the same way, the tip of the minute hand, in modeling coordinates, is at $(0, 9)$. What are the `Canvas` coordinates of the tip of the hour hand? We must multiply (reading from leaf toward root) by all the transformation matrices from the hour-hand template up to the `Canvas`, resulting in

$$\begin{bmatrix} 0.96 & -0.26 & 0 \\ 0.26 & 0.96 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1.7 & 0 & 0 \\ 0 & 0.7 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 9 \\ 1 \end{bmatrix} \tag{10.91}$$

$$= \begin{bmatrix} -1.64 & -.18 & 0 \\ -0.44 & -0.68 & 0 \\ 001 & & \end{bmatrix} \begin{bmatrix} 0 \\ 9 \\ 1 \end{bmatrix} = \begin{bmatrix} 1.63 \\ -6.09 \\ 1 \end{bmatrix}, \tag{10.92}$$

where all coordinates have been rounded to two decimal places for clarity. The `Canvas` coordinates of the tip of the minute hand are

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 9 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 9 \\ 1 \end{bmatrix}. \tag{10.93}$$

We can thus compute a vector from the hour hand's tip to the minute hand's tip by subtracting these two, getting $\begin{bmatrix} -1.63 & 15.08 & 0 \end{bmatrix}^{\mathbf{T}}$. The result is the homogeneous-coordinate representation of the vector $\begin{bmatrix} -1.63 & 15.08 \end{bmatrix}^{\mathbf{T}}$ in `Canvas` coordinates.

Suppose that we wanted to know the direction from the tip of the minute hand to the tip of the hour hand *in minute-hand coordinates.* If we knew this direction, we could add, within the minute-hand part of the model, a small arrow that pointed toward the hour-hand. To find this direction vector, we need to know the coordinates of the tip of the hour hand in minute-hand coordinates. So we must go from hour-hand coordinates to minute-hand coordinates, which we can do by working up the tree from the hour hand to the `Canvas`, and then back down to the minute hand. The location of the hour-hand tip, in minute-hand coordinates, is given by

$$\begin{bmatrix}-1 & 0 & 0\\ 0 & -1 & 0\\ 0 & 0 & 1\end{bmatrix}^{-1}\begin{bmatrix}-1 & 0 & 0\\ 0 & -1 & 0\\ 0 & 0 & 1\end{bmatrix}^{-1}\begin{bmatrix}0.96 & -0.26 & 0\\ 0.26 & 0.96 & 0\\ 0 & 0 & 1\end{bmatrix}$$
$$\cdot\begin{bmatrix}-1 & 0 & 0\\ 0 & -1 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1.7 & 0 & 0\\ 0 & 0.7 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}0\\ 9\\ 1\end{bmatrix}. \tag{10.94}$$

We subtract from this the coordinates, $(0, 9)$, of the tip of the minute hand (in the minute-hand coordinate system) to get a vector from the tip of the minute hand to the tip of the hour hand.

As a final exercise, suppose we wanted to create an animation of the clock in which someone has grabbed the minute hand and held it so that the rest of the clock spins around the minute hand. How could we do this?

Well, the reason the minute hand moves from its initial 12:00 position on the `Canvas` (i.e., its position *after* it has been rotated 180° the first time) is that a sequence of further transformations have been applied to it. This sequence is rather short: It's just the varying rotation. If we apply the *inverse* of this varying rotation to each of the clock elements, we'll get the desired result. Because we apply both the rotation and its inverse to the minute hand, we could delete both, but the structure is more readable if we retain them. We could also apply the inverse rotation as part of the `Canvas`'s render transform.

Inline Exercise 10.23: If we want to implement the second approach—inserting the inverse rotation in the `Canvas`'s render transform—should it appear (in the WPF code) before or after the scale-and-translate transforms that are already there? Try it!

10.12 Transforming Vectors and Covectors

We've agreed to say that the point $(x, y) \in E^2$ corresponds to the 3-space vector $\begin{bmatrix}x & y & 1\end{bmatrix}^{\mathbf{T}}$, and that the vector $\begin{bmatrix}u\\ v\end{bmatrix}$ corresponds to the 3-space vector $\begin{bmatrix}u & v & 0\end{bmatrix}^{\mathbf{T}}$. If we use a 3×3 matrix $\mathbf{M}$ (with last row $\begin{bmatrix}0 & 0 & 1\end{bmatrix}$) to transform 3-space via

$$T : \mathbf{R^3} \to \mathbf{R^3} : \mathbf{x} \mapsto \mathbf{Mx}, \tag{10.95}$$

then the restriction of T to the $w = 1$ plane has its image in E^2 as well, so we can write

$$(T|E^2) : E^2 \to E^2 : \mathbf{x} \mapsto \mathbf{Mx}. \tag{10.96}$$

But we also noted above that we could regard T as transforming **vectors,** or displacements of two-dimensional Euclidean space, which are typically written with two coordinates but which we represent in the form $\begin{bmatrix}u & v & 0\end{bmatrix}^{\mathbf{T}}$. Because the last entry of such a "vector" is always 0, the last column of $\mathbf{M}$ has no effect on how vectors are transformed. Instead of computing

$$\mathbf{M}\begin{bmatrix} u \\ v \\ 0 \end{bmatrix}, \tag{10.97}$$

we could equally well compute

$$\begin{bmatrix} m_{1,1} & m_{1,2} & 0 \\ m_{2,1} & m_{2,2} & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u \\ v \\ 0 \end{bmatrix}, \tag{10.98}$$

and the result would have a 0 in the third place. In fact, we could transform such vectors directly as two-coordinate vectors, by simply computing

$$\begin{bmatrix} m_{1,1} & m_{1,2} \\ m_{2,1} & m_{2,2} \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}. \tag{10.99}$$

For this reason, it's sometimes said for an affine transformation of the Euclidean plane represented by multiplication by a matrix $\mathbf{M}$ that the associated transformation of vectors is represented by

$$\overline{\mathbf{M}} = \begin{bmatrix} m_{1,1} & m_{1,2} \\ m_{2,1} & m_{2,2} \end{bmatrix}. \tag{10.100}$$

What about covectors? Recall that a typical covector could be written in the form

$$\phi_{\mathbf{w}} : \mathbf{R}^2 \to \mathbf{R}^2 : \mathbf{v} \mapsto \mathbf{w} \cdot \mathbf{v}, \tag{10.101}$$

where $\mathbf{w}$ was some vector in $\mathbf{R}^2$. We'd like to transform $\phi_{\mathbf{w}}$ in a way that's consistent with T. Figure 10.19 shows why: We often build a geometric model of some shape and compute all the normal vectors to the shape. Suppose that $\mathbf{n}$ is one such surface normal. We then place that shape into 3-space by applying some "modeling transformation" $T_{\mathbf{M}}$ to it, and we'd like to know the normal vectors to that transformed shape so that we can do things like compute the angle between a light-ray $\mathbf{v}$ and that surface normal. If we call the transformed surface normal $\mathbf{m}$,

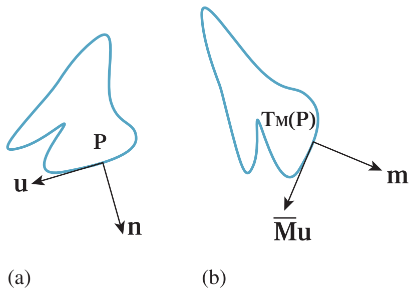


Figure 10.19: (a) A geometric shape that's been modeled using some modeling tool; the normal vector $\mathbf{n}$ *at a particular point* P *has been computed too. The vector* $\mathbf{u}$ *is tangent to the shape at* P*. (b) The shape has been translated, rotated, and scaled as it was placed into a scene. At the transformed location of* P*, we want to find the normal vector* $\mathbf{m}$ *with the property that its inner product with the transformed tangent* $\overline{\mathbf{M}}\mathbf{u}$ *is still 0.*

we want to compute $\mathbf{v} \cdot \mathbf{m}$. How is $\mathbf{m}$ related to the surface normal $\mathbf{n}$ of the original model?

The original surface normal $\mathbf{n}$ was defined by the property that it was orthogonal to every vector $\mathbf{u}$ that was tangent to the surface. The new normal vector $\mathbf{m}$ must be orthogonal to all the transformed tangent vectors, which are tangent to the transformed surface. In other words, we need to have

$$\mathbf{m} \cdot \overline{\mathbf{M}}\mathbf{u} = 0 \tag{10.102}$$

for every tangent vector $\mathbf{u}$ to the surface. In fact, we can go further. For *any* vector $\mathbf{u}$, we'd like to have

$$\mathbf{m} \cdot \overline{\mathbf{M}}\mathbf{u} = \mathbf{n} \cdot \mathbf{u}, \tag{10.103}$$

that is, we'd like to be sure that the angle between an untransformed vector and $\mathbf{n}$ is the same as the angle between a transformed vector and $\mathbf{m}$.

Before working through this, let's look at a couple of examples. In the case of the transformation T_1, the vector perpendicular to the bottom side of the house (we'll use this as our vector $\mathbf{n}$) should be transformed so that it's still perpendicular to the bottom of the transformed house. This is achieved by rotating it by 30° (see Figure 10.20).

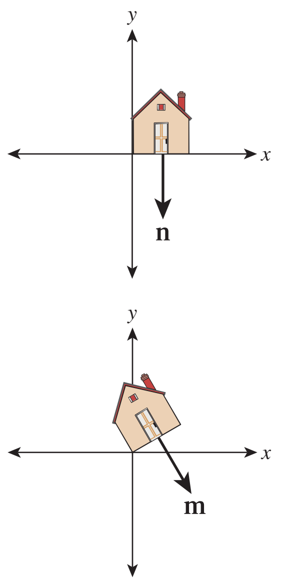


Figure 10.20: For a rotation, the normal vector rotates the same way as all other vectors.

If we just *translate* the house, then $\mathbf{n}$ again should be transformed just the way we transform ordinary vectors, that is, not at all.

But what about when we shear the house, as with example transformation T_3? The associated vector transformation is still a shearing transformation; it takes a vertical vector and tilts it! But the vector $\mathbf{n}$, if it's to remain perpendicular to the bottom of the house, must not be changed at all (see Figure 10.21). So, in this case, we see the necessity of transforming covectors differently from vectors.

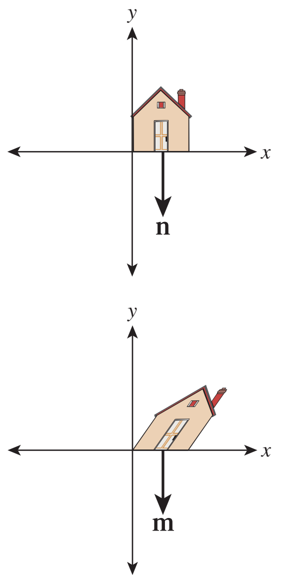


Figure 10.21: While the vertical sides of the house are sheared, the normal vector to the house's bottom remains unchanged.

Let's write down, once again, what we want. We're looking for a vector $\mathbf{m}$ that satisfies

$$\mathbf{m} \cdot (\overline{\mathbf{M}}\mathbf{u}) = \mathbf{n} \cdot \mathbf{u} \tag{10.104}$$

for every possible vector $\mathbf{v}$. To make the algebra more obvious, let's swap the order of the vectors and say that we want

$$(\overline{\mathbf{M}}\mathbf{u}) \cdot \mathbf{m} = \mathbf{u} \cdot \mathbf{n}. \tag{10.105}$$

Recalling that $\mathbf{a} \cdot \mathbf{b}$ can be written $\mathbf{a}^{\mathbf{T}}\mathbf{b}$, we can rewrite this as

$$(\overline{\mathbf{M}}\mathbf{u})^{\mathbf{T}}\mathbf{m} = \mathbf{u}^{\mathbf{T}}\mathbf{n}. \tag{10.106}$$

Remembering that $(\mathbf{AB})^{\mathbf{T}} = \mathbf{A}^{\mathbf{T}}\mathbf{B}^{\mathbf{T}}$, and then simplifying, we get

$$(\overline{\mathbf{M}}\mathbf{u})^{\mathbf{T}}\mathbf{m} = \mathbf{u}^{\mathbf{T}}\mathbf{n} \tag{10.107}$$

$$(\mathbf{u}^{\mathbf{T}}\overline{\mathbf{M}}^{\mathbf{T}})\mathbf{m} = \mathbf{u}^{\mathbf{T}}\mathbf{n} \tag{10.108}$$

$$\mathbf{u}^{\mathbf{T}}(\overline{\mathbf{M}}^{\mathbf{T}}\mathbf{m}) = \mathbf{u}^{\mathbf{T}}\mathbf{n}, \tag{10.109}$$

where the last step follows from the associativity of matrix multiplication. This last equality is of the form $\mathbf{u} \cdot \mathbf{a} = \mathbf{u} \cdot \mathbf{b}$ for all $\mathbf{u}$. Such an equality holds if and only if $\mathbf{a} = \mathbf{b}$, that is, if and only if

$$\overline{\mathbf{M}}^{\mathbf{T}}\mathbf{m} = \mathbf{n}, \tag{10.110}$$

so

$$\mathbf{m} = (\overline{\mathbf{M}}^{\mathbf{T}})^{-1}\mathbf{n}, \tag{10.111}$$

where we are assuming that $\overline{\mathbf{M}}$ is invertible.

We can therefore conclude that the covector $\phi_{\mathbf{n}}$ transforms to the covector $\phi_{(\overline{\mathbf{M}}^{\mathbf{T}})^{-1}\mathbf{n}}$. For this reason, the inverse transpose is sometimes referred to as the **covector transformation** or (because of its frequent application to normal vectors) the **normal transform.** Note that if we choose to write covectors as row vectors, then the transpose is not needed, but we have to multiply the row vector on the *right* by $\overline{\mathbf{M}}^{-1}$.

◈ The normal transform, in its natural mathematical setting, goes in the opposite direction: It takes a normal vector in the codomain of $T_{\mathbf{M}}$ and produces one in the domain; the matrix for this **adjoint transformation** is $\mathbf{M}^{\mathbf{T}}$. Because we need to use it in the other direction, we end up with an inverse as well.

Taking our shearing transformation, T_3, as an example, when written in *xyw*-space the matrix $\mathbf{M}$ for the transformation is

$$\begin{bmatrix} 1 & 2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{10.112}$$

and hence $\overline{\mathbf{M}}$ is

$$\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}, \tag{10.113}$$

while the normal transform is

$$(\overline{\mathbf{M}}^{-1})^{\mathbf{T}} = \begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}^{\mathbf{T}} = \begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix}. \tag{10.114}$$

Hence the covector $\phi_{\mathbf{n}}$, where $\mathbf{n} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$, for example, becomes the covector $\phi_{\mathbf{m}}$, where $\mathbf{m} = \begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix} \mathbf{n} = \begin{bmatrix} 2 \\ -3 \end{bmatrix}$.

Inline Exercise 10.24: (a) Find an equation (in coordinates, not vector form) for a line passing through the point $P = (1, 1)$, with normal vector $\mathbf{n} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$. (b) Find a second point Q on this line. (c) Find $P' = T_3(P)$ and $Q' = T_3(Q)$, and a coordinate equation of the line joining P' and Q'. (d) Verify that the normal to this second line is in fact proportional to $\mathbf{m} = \begin{bmatrix} 2 \\ -3 \end{bmatrix}$, confirming that the normal transform really did properly transform the normal vector to this line.

Inline Exercise 10.25: We assumed that the matrix $\mathbf{M}$ was invertible when we computed the normal transform. Give an intuitive explanation of why, if $\mathbf{M}$ is degenerate (i.e., not invertible), it's impossible to define a normal transform. Hint: Suppose that $\mathbf{u}$, in the discussion above, is sent to $\mathbf{0}$ by $\mathbf{M}$, but that $\mathbf{u} \cdot \mathbf{n}$ is nonzero.

10.12.1 Transforming Parametric Lines

All the transformations of the $w = 1$ plane we've looked at share the property that they send lines into lines. But more than that is true: They send *parametric* lines to *parametric* lines, by which we mean that if ℓ is the parametric line $\ell = \{P + t\mathbf{v} : t \in \mathbf{R}\}$, and $Q = P + \mathbf{v}$ (i.e., ℓ starts at P and reaches Q at $t = 1$), and T is the transformation $T(\mathbf{v}) = \mathbf{Mv}$, then $T(\ell)$ is the line

$$T(\ell) = \{T(P) + t(T(Q) - T(P)) : t \in \mathbf{R}\}, \tag{10.115}$$

and in fact, the point at parameter t in ℓ (namely $P + t(Q - P)$) is sent by T to the point at parameter t in $T(\ell)$ (namely $T(P) + t(T(Q) - T(P))$).

This means that for the transformations we've considered so far, transforming the plane commutes with forming affine or linear combinations, so you can either transform and then average a set of points, or average and then transform, for instance.

10.13 More General Transformations

Let's look at one final transform, T, which is a prototype for transforms we'll use when we study projections and cameras in 3D. All the essential ideas occur in 2D, so we'll look at this transformation carefully. The matrix $\mathbf{M}$ for the transformation T is

$$\mathbf{M} = \begin{bmatrix} 2 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}. \tag{10.116}$$

It's easy to see that $T_{\mathbf{M}}$ doesn't transform the $w = 1$ plane into the $w = 1$ plane.

Inline Exercise 10.26: Compute $T(\begin{bmatrix} 2 & 0 & 1 \end{bmatrix}^{\mathbf{T}})$ and verify that the result is not in the $w = 1$ plane.

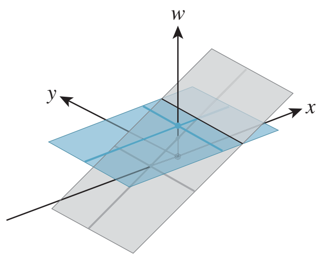


Figure 10.22: The blue $w = 1$ plane transforms into the tilted gray plane under $T_{\mathbf{M}}$.

Figure 10.22 shows the $w = 1$ plane in blue and the transformed $w = 1$ plane in gray. To make the transformation T useful to us in our study of the $w = 1$ plane, we need to take the points of the gray plane and "return" them to the blue plane somehow. To do so, we introduce a new function, H, defined by

$$H : \mathbf{R}^3 - \{ \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} : x, y, \in \mathbf{R}\} \to \mathbf{R}^3 : \begin{bmatrix} x \\ y \\ w \end{bmatrix} \mapsto \begin{bmatrix} x/w, y/w, 1 \end{bmatrix}. \tag{10.117}$$

Figure 10.23 show how the analogous function in two dimensions sends every point except those on the $w = 0$ line to the line $w = 1$: For a typical point P, we connect P to the origin O with a line and see where this line meets the $w = 1$ plane. Notice that even a point in the negative-w half-space on the same line gets sent to the same location on the $w = 1$ line. This connect-and-intersect operation isn't defined, of course, for points on the x-axis, because the line connecting them to the origin is the axis itself, which never meets the $w = 1$ line. H is often called **homogenization** in graphics.

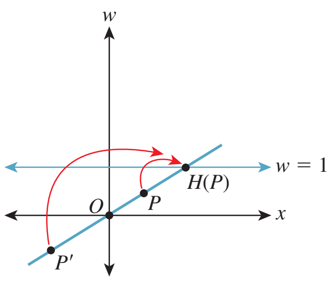


Figure 10.23: Homogenization $\begin{bmatrix} x \\ w \end{bmatrix} \mapsto \begin{bmatrix} x/w \\ 1 \end{bmatrix}$ *in two dimensions.*

With H in hand, we'll define a new transformation on the $w = 1$ plane by

$$S(\mathbf{v}) = H(T_{\mathbf{M}}(\mathbf{v}). \tag{10.118}$$

This definition has a serious problem: As you can see from Figure 10.22, some points in the image of T are in the $w = 0$ plane, on which H is not defined so that S cannot be defined there. For now, we'll ignore this and simply not apply S to any such points.

Inline Exercise 10.27: Find all points $\mathbf{v} = \begin{bmatrix} x & y & 1 \end{bmatrix}^{\mathbf{T}}$ of the $w = 1$ plane such that the w-coordinate of $T_{\mathbf{M}}(\mathbf{v})$ is 0. These are the points on which S is undefined.

The transformation S, defined by multiplication by the matrix $\mathbf{M}$, followed by homogenization, is called a **projective transformation.** Notice that if we followed either a linear or affine transformation with homogenization, the homogenization would have no effect. Thus, we have three nested classes of transformations: linear, affine (which includes linear *and* translation and combinations of them), and projective (which includes affine *and* transformations like S).

Figure 10.24 shows several objects in the $w = 1$ plane, drawn as seen looking down the w-axis, with the y-axis, on which S is undefined, shown in pale green. Figure 10.25 shows these objects after S has been applied to them. Evidently, S takes lines to lines, mostly: A line segment like the blue one in the figure that meets the y-axis in the segment's interior turns into two rays, but the two rays both lie in the same line. We say that the line $y = 0$ has been "sent to infinity." The red vertical line at $x = 1$ in Figure 10.24 transforms into the red vertical line at $x = 0$ in Figure 10.25. And every ray through the origin in Figure 10.24 turns into a horizontal line in Figure 10.25. We can say even more: Suppose that P_1 denotes radial projection onto the $x = 1$ line in Figure 10.24, while P_2 denotes horizontal projection onto the $z = 0$ line in Figure 10.25. Then

$$S(P_1(X)) = P_2(S(X)) \tag{10.119}$$

for any point X that's not on the y-axis. In other words, S converts radial projection into parallel projection. In Chapter 13 we'll see exactly the same trick in 3-space: We'll convert radial projection toward the eye into parallel projection. This is useful because in parallel projection, it's *really* easy to tell when one object obscures another by just comparing "depth" values!

Let's look at how S transforms a *parameterized* line. Consider the line ℓ starting at a point P and passing through a point Q when $t = 1$,

$$\ell(t) = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} + t \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} \tag{10.120}$$

$$= P + t(Q - P), \tag{10.121}$$

where $P = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}^{\mathbf{T}}$ and $Q = \begin{bmatrix} 3 & 1 & 1 \end{bmatrix}^{\mathbf{T}}$ so that in the $w = 1$ plane, the line starts at $(x, y) = (1, 0)$ when $t = 0$ and goes to the right and slightly upward, arriving at $(x, y) = (3, 1)$ when $t = 1$ (see Figure 10.26).

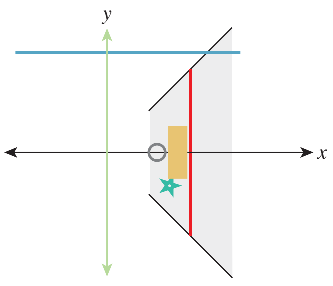


Figure 10.24: Objects in the $w = 1$ plane before transformation.

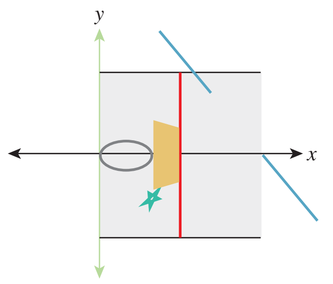


Figure 10.25: The same objects after transformation by S.

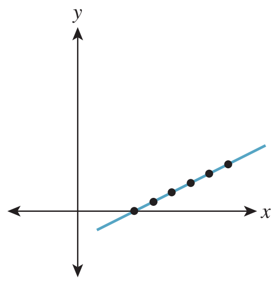


Figure 10.26: The line ℓ passes through P at $t = 0$ and Q at $t = 1$; the black points are equispaced in the interval $0 \leq t \leq 1$.

The function T transforms this to the line ℓ' that starts at $T(P) = \begin{bmatrix}1 & 0 & 1\end{bmatrix}^{\mathbf{T}}$ when $t = 0$ and arrives at $T(Q) = \begin{bmatrix}5 & 1 & 3\end{bmatrix}^{\mathbf{T}}$ when $t = 1$, and whose equation is

$$\ell'(t) = \begin{bmatrix}1 & 0 & 1\end{bmatrix} + t\begin{bmatrix}4 & 1 & 2\end{bmatrix} \tag{10.122}$$

$$= T(P) + t(T(Q) - T(P)). \tag{10.123}$$

Figure 10.27 shows the line in 3-space, after transformation by $T_{\mathbf{M}}$; the point spacing remains constant.

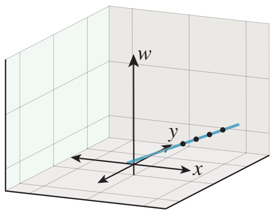


Figure 10.27: After transformation by $T_{\mathbf{M}}$, the points are still equispaced.

We know that this is the parametric equation of the line, because *every* linear transformation transforms parametric lines to parametric lines. But when we apply H, something interesting happens. Because the function H is *not linear,* the parametric line is *not* transformed to a parametric line. The point $\ell'(t) = \begin{bmatrix}1+4t & t & 1+2t\end{bmatrix}^{\mathbf{T}}$ is sent to

$$m(t) = \begin{bmatrix}(1+4t)/(1+2t)\\ t/(1+2t)\\ 1\end{bmatrix} \tag{10.124}$$

$$= \begin{bmatrix}1\\0\\1\end{bmatrix} + \frac{t}{1+2t}\begin{bmatrix}2\\1\\0\end{bmatrix} \tag{10.125}$$

Equation 10.125 has *almost* the form of a parametric line, but the coefficient of the direction vector, which is proportional to $S(Q) - S(P)$, has the form

$$\frac{at+b}{ct+d}, \tag{10.126}$$

which is called a **fractional linear transformation** of t. This nonstandard form is of serious importance in practice: It tells us that if we interpolate a value at the midpoint M of P and Q, for instance, from the values at P and Q, and then transform all three points by S, then $S(M)$ will generally *not* be at the midpoint of $S(P)$ and $S(Q)$, so the interpolated value will not be the correct one to use if we need post-transformation interpolation. Figure 10.28 shows how the equally spaced points in the domain have become unevenly spaced after the projective transformation.

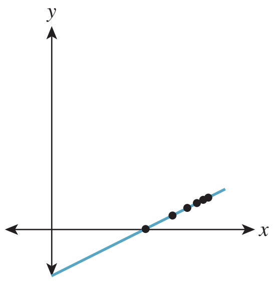


Figure 10.28: After homogenization, the points are no longer equispaced.

In other words, transformation by S and interpolation are not commuting operations. When we apply a transformation that includes the homogenization operation H, we cannot assume that interpolation will give the same results pre- and post-transformation. Fortunately, there's a solution to this problem (see Section 15.6.4).

Inline Exercise 10.28: (a) Show that if n and f are distinct nonzero numbers, the transformation defined by the matrix

$$\mathbf{N} = \begin{bmatrix}\frac{f}{f-n} & 0 & \frac{fn}{n-f}\\ 0 & 1 & 0\\ 1 & 0 & 0\end{bmatrix}, \tag{10.127}$$

when followed by homogenization, sends the line $x = 0$ to infinity, the line $x = n$ to $x = 0$, and the line $x = f$ to $x = 1$.
(b) Figure out how to modify the matrix to send $x = f$ to $x = -1$ instead.

Inline Exercise 10.29: (a) Show that if T is any linear transformation on $\mathbf{R}^3$, then for any nonzero $\alpha \in \mathbf{R}$ and any vector $\mathbf{v} \in \mathbf{R}^3$, $H(T(\alpha\mathbf{v})) = H(T(\mathbf{v}))$.
(b) Show that if $\mathbf{K}$ is any matrix, then $H(T_{\mathbf{K}}(\mathbf{v})) = H(T_{\alpha\mathbf{K}}(\mathbf{v}))$ as well.
(c) Conclude that in a sequence of matrix operations in which there's an H at the end, matrix scale doesn't matter, that is, you can multiply a matrix by any nonzero constant without changing the end result.

Suppose we have a matrix transformation on 3-space given by $T(\mathbf{v}) = \mathbf{Kv}$, and T is nondegenerate (i.e., $T(\mathbf{v}) = \mathbf{0}$ only when $\mathbf{v} = \mathbf{0}$). Then T takes lines through the origin to lines through the origin, because if $\mathbf{v} \neq \mathbf{0}$ is any nonzero vector, then $\{\alpha\mathbf{v} : \alpha \in \mathbf{R}\}$ is the line through the origin containing $\mathbf{v}$, and when we transform this, we get $\{\alpha T(\mathbf{v}) : \alpha \in \mathbf{R}\}$, which is the line through the origin containing $T(\mathbf{v})$. Thus, rather than thinking of the transformation T as moving around points in $\mathbf{R}^3$, we can think of it as acting on the set of lines through the origin. By intersecting each line through the origin with the $w = 1$ plane, we can regard T as acting on the $w = 1$ plane, but with a slight problem: A line through the origin in 3-space that meets the $w = 1$ plane may be transformed to one that does not (i.e., a horizontal line), and vice versa. So using the $w = 1$ plane to "understand" the lines-to-lines version of the transformation T is a little confusing.

The idea of considering linear transformations as transformations on the set of lines through the origin is central to the field of **projective geometry.** An understanding of projective geometry can lead to a deeper understanding of the transformations we use in graphics, but is by no means essential. Hartshorne [Har09] provides an excellent introduction for the student who has studied abstract algebra.

Transformations of the $w = 1$ plane like the ones we've been looking at in this section, consisting of an arbitrary matrix transformation on $\mathbf{R}^3$ followed by H, are called **projective transformations.** The class of projective transformations includes all the more basic operations like translation, rotation, and scaling of the plane (i.e., *affine* transformations of the plane), but include many others as well. Just as with linear and affine transformations, there's a uniqueness theorem: If P, Q, R, and S are four points of the plane, no three collinear, then there's exactly one projective transformation sending these points to $(0, 0)$, $(1, 0)$, $(0, 1)$, and $(1, 1)$, respectively. (Note that this one transformation might be described by two *different* matrices. For example, if $\mathbf{K}$ is the matrix of a projective transformation S, then $2\mathbf{K}$ defines exactly the same transformation.)

For all the affine transformations we discussed in earlier sections, we've determined an associated transformation of vectors and of normal vectors. For projective transformations, this process is messier. Under the projective transformation shown in Figures 10.24 and 10.25, we can consider the top and bottom edges of the tan rectangle as vectors that point in the same direction. After the transformation, you can see that they have been transformed to point in *different* directions. There's no single "vector" transformation to apply. If we have a vector $\mathbf{v}$ starting at the point P, we have to apply "the vector transformation at P" to $\mathbf{v}$ to find out where it will go. The same idea applies to normal vectors: There's a different normal transformation at every point. In both cases, it's the function H that leads to problems. The "vector" transformation for any function U is, in general, the derivative DU. In the case of a matrix transformation $T_{\mathbf{M}}$ that's being applied only to points of the $w = 1$ plane, the "vectors" lying in that plane all have $w = 0$, and so the matrix used to transform these vectors can have its third column set to be all zeroes (or can be just written as a 2×2 matrix operating on vectors with two entries), as we have seen earlier. But since

$$S = H \circ T_{\mathbf{M}}, \tag{10.128}$$

we have (using the multivariable chain rule)

$$DS(P) = DH(T_{\mathbf{M}}(P)) \cdot DT_{\mathbf{M}}(P). \tag{10.129}$$

Now, since $H(\begin{bmatrix} x \\ y \\ w \end{bmatrix}) = \begin{bmatrix} x/w \\ y/w \\ 1 \end{bmatrix}$, we know that

$$DH(\begin{bmatrix} x \\ y \\ w \end{bmatrix}) = \begin{bmatrix} 1/w & 0 & -x/w^2 \\ 0 & 1/w & -y/w^2 \\ 0 & 0 & 0 \end{bmatrix} \tag{10.130}$$

$$= \frac{1}{w^2} \begin{bmatrix} w & 0 & -x \\ 0 & w & -y \\ 0 & 0 & 0 \end{bmatrix} \tag{10.131}$$

and

$$DT_{\mathbf{M}}(P) = \mathbf{M} = \begin{bmatrix} 2 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}. \tag{10.132}$$

So, if $P = \begin{bmatrix} x, y, 1 \end{bmatrix}$ is a point of the $w = 1$ plane and $\mathbf{v} = \begin{bmatrix} s \\ t \\ 0 \end{bmatrix}$ is a vector in that plane, then $S(P) = \begin{bmatrix} 2x-1 \\ y \\ x \end{bmatrix}$ and

$$DS(P)(\mathbf{v}) = DH(\begin{bmatrix} 2x-1 \\ y \\ x \end{bmatrix}) \cdot DT(P)\mathbf{v} = \begin{bmatrix} s \\ t \\ 0 \end{bmatrix} \tag{10.133}$$

$$= \frac{1}{x^2} \begin{bmatrix} x & 0 & -(2x+1) \\ 0 & x & -y \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} s \\ t \\ 0 \end{bmatrix} \tag{10.134}$$

$$= \frac{1}{x^2} \begin{bmatrix} -1 & 0 & -x \\ -y & x & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} s \\ t \\ 0 \end{bmatrix} = \begin{bmatrix} -s/x^2 \\ (tx - sy)/x^2 \\ 0 \end{bmatrix}. \tag{10.135}$$

Evidently, the "vector" transformation depends on the point $(x, y, 1)$ at which it's applied. The normal transform, being the inverse transpose of the vector transform, has the same dependence on the point of application.

10.14 Transformations versus Interpolation

When you rotate a book on your desk by 30° counterclockwise, the book is rotated by each intermediate amount between zero and 30°. But when we "rotate" the house in our diagram by 30°, we simply compute the final position of each point of the house. In no sense has it passed through any intermediate positions. In the more extreme case of rotation by 180°, the resultant transformation is exactly the same as the "uniform scale by -1" transformation. And in the case of rotation by 360°, the resultant transformation is the identity.

This reflects a *limitation in modeling.* The use of matrix transformations to model transformations of ordinary objects captures only the relationship between initial and final positions, and not the means by which the object got from the initial to the final position.

Much of the time, this distinction is unimportant: We want to put an object into a particular pose, so we apply some sequence of transformations to it (or its parts). But sometimes it can be quite significant: We might instead want to show the object being transformed from its initial state to its final state. An easy, but rarely useful, approach is to linearly interpolate each point of the object from its initial to its final position. If we do this for the "rotation by 180°" example, at the halfway point the entire object is collapsed to a single point; if we do it from the "rotation by 360°" example, the object never appears to move at all! The problem is that what we *really* want is to find interpolated versions of our *descriptions* of the transformation rather than of the transformations themselves. (Thus, to go from the initial state to "rotated by 360" we'd apply "rotate by s" to the initial state, for each value of s from 0 to 360.)

But sometimes students confuse a transformation like "multiplication by the identity matrix" with the way it was specified, "rotate by 360°," and they can be frustrated with the impossibility of "dividing by two" to get a rotation by 180°, for instance. This is particularly annoying when one has access only to the matrix form of the transformation, rather than the initial specification; in that case, as the examples show, there's no hope for a general solution to the problem of "doing a transformation partway." On the other hand, there *is* a solution that often gives reasonable results in practice, especially for the problem of interpolating two rather similar transformations (e.g., interpolating between rotating by 20° and rotating by 30°), which often arises. We'll discuss this in Chapter 11.

10.15 Discussion and Further Reading

We've introduced three classes of basic transformations: *linear,* which you've already encountered in linear algebra; *affine,* which includes translations and can be seen as a subset of the linear transformations in xyw-space, restricted to the $w = 1$ plane; and *projective,* which arises from general linear transformations on xyw-space, restricted to the $w = 1$ plane and then followed by the homogenization operation that divides through by w. We've shown how to represent each kind of transformation by matrix multiplication, but we urge you to separate the idea of a transformation from the matrix that represents it.

For each category, there's a theorem about uniqueness: A linear transformation on the plane is determined by its values on two independent vectors; an affine transformation is determined by its values on any three noncollinear points;

a projective transformation is determined by its values on any four points, no three of which are collinear. In the next chapter we'll see analogous results for 3-space, and in the following one we'll see how to use these theorems to build a library for representing transformations so that you don't have to spend a lot of time building individual matrices.

Even though matrices are not as easy for humans to interpret as "This transformation sends the points A, B, and C to A', B', and C'," the matrix representation of a transformation is very valuable, mostly because composition of transformation is equivalent to multiplication of matrices; performing a complex sequence of transformations on many points can be converted to multiplying the points' coordinates by a *single* matrix.

10.16 Exercises

Exercise 10.1: Use the 2D test bed to write a program to demonstrate windowing transforms. The user should click and drag two rectangles, and you should compute the transform between them. Subsequent clicks by the user within the first rectangle should be shown as small dots, and the corresponding locations in the second rectangle should also be shown as dots. Provide a Clear button to let the user restart.

Exercise 10.2: Multiply $\mathbf{M} = \begin{bmatrix} a & c \\ b & d \end{bmatrix}$ by the expression given in Equation 10.17 for its inverse to verify that the product really is the identity.

Exercise 10.3: Suppose that $\mathbf{M}$ is an $n \times n$ square matrix with SVD $\mathbf{M} = \mathbf{UDV^T}$.
(a) Why is $\mathbf{V^TV}$ the identity?
(b) Let i be any number from 1 to n. What is $\mathbf{V^T}\mathbf{v}_i$, where $\mathbf{v}_i$ denotes the ith column of $\mathbf{V}$? Hint: Use part (a).
(c) What's $\mathbf{DV^T}\mathbf{v}_i$?
(d) What's $\mathbf{M}\mathbf{v}_i$ in terms of $\mathbf{u}_i$ and d_i, the ith diagonal entry of D?
(e) Let $\mathbf{M}' = d_1\mathbf{u}_1\mathbf{v}_1^{\mathbf{T}} + \ldots + d_n\mathbf{u}_n\mathbf{v}_n^{\mathbf{T}}$. Show that $\mathbf{M}'\mathbf{v}_i = d_i\mathbf{u}_i$.
(f) Explain why $\mathbf{v}_i, i = 1, \ldots, n$ are linearly independent, and thus span $\mathbf{R}^n$.
(g) Conclude that $\mathbf{w} \mapsto \mathbf{Mw}$ and $\mathbf{w} \mapsto \mathbf{M}'\mathbf{w}$ agree on n linearly independent vectors, and hence must be the same linear transformation of $\mathbf{R}^n$.
(h) Conclude that $\mathbf{M}' = \mathbf{M}$. Thus, the singular-value decomposition proves the theorem that every matrix can be written as a sum of **outer products** (i.e., matrices of the form $\mathbf{vw^T}$).

Exercise 10.4: (a) If P, Q, and R are noncollinear points in the plane, show that $Q - P$ and $R - P$ are linearly independent vectors.
(b) If $\mathbf{v}_1$ and $\mathbf{v}_2$ are linearly independent points in the plane, and A is any point in the plane, show that $A, B = A + \mathbf{v}_1$ and $C = A + \mathbf{v}_2$ are noncollinear points. This shows that the two kinds of affine frames are equivalent.
(c) Two forms of an affine frame in 3-space are (i) four points, no three coplanar, and (ii) one point and three linearly independent vectors. Show how to convert from one to the other, and also describe a third possible version (Three points and one vector? Two points and two vectors? You choose!) and show its equivalence as well.

Exercise 10.5: We said that if the columns of the matrix $\mathbf{M}$ are $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k \in R^n$, and they are pairwise orthogonal unit vectors, then $\mathbf{M^TM} = \mathbf{I}_k$, the $k \times k$ identity matrix.

(a) Explain why, in this situation, $k \leq n$.
(b) Prove the claim that $\mathbf{M^T M} = \mathbf{I}_k$.

Exercise 10.6: An image (i.e., an array of grayscale values between 0 and 1, say) can be thought of as a large matrix, $\mathbf{M}$ (indeed, this is how we usually represent images in our programs). Use a linear algebra library to compute the SVD $\mathbf{M} = \mathbf{UDV^T}$ of some image $\mathbf{M}$. According to the decomposition theorem described in Exercise 10.3, this describes the image as a sum of outer products of many vectors. If we replace the last 90% of the diagonal entries of $\mathbf{D}$ with zeroes to get a new matrix $\mathbf{D}'$, then the product $\mathbf{M}' = \mathbf{UD'V}$ deletes 90% of the terms in this sum of outer products. In doing so, however, it deletes the *smallest* 90% of the terms. Display $\mathbf{M}'$ and compare it to $\mathbf{M}$. Experiment with values other than 90%. At what level do the two images become indistinguishable? You may encounter values less than 0 and greater than 1 during the process described in this exercise. You should simply clamp these values to the interval $[0, 1]$.

Exercise 10.7: The **rank** of a matrix is the number of linearly independent columns of the matrix.
(a) Explain why the outer product of two nonzero vectors always has rank one.
(b) The decomposition theorem described in Exercise 10.3 expresses a matrix $\mathbf{M}$ as a sum of rank one matrices. Explain why the sum of the first p such outer products has rank p (assuming $d_1, d_2, \ldots, d_p \neq 0$). In fact, this sum $\mathbf{M}_p$ is the rank p matrix that's closest to $\mathbf{M}$, in the sense that the sum of the squares of the entries of $\mathbf{M} - \mathbf{M}_p$ is as small as possible. (You need not prove this.)

Exercise 10.8: Suppose that $T : \mathbf{R^2} \to \mathbf{R^2}$ is a linear transformation represented by the 2×2 matrix $\mathbf{M}$, that is, $T(\mathbf{x}) = \mathbf{Mx}$. Let $K = \max_{\mathbf{x} \in \mathbf{S}^1} \|T(\mathbf{x})\|^2$, that is, K is the maximum squared length of all unit vectors transformed by $\mathbf{M}$.
(a) If the SVD of $\mathbf{M}$ is $\mathbf{M} = \mathbf{UDV^T}$, show that $K = d_1^2$.
(b) What is the *minimum* squared length among all such vectors, in terms of $\mathbf{D}$?
(c) Generalize to $\mathbf{R^3}$.

Exercise 10.9: Show that three distinct points P, Q, and R in the Euclidean plane are collinear if and only if the corresponding vectors ($\mathbf{v}_P = \begin{bmatrix} P_x \\ P_y \\ 1 \end{bmatrix}$, etc.) are linearly dependent, by showing that if $\alpha_P \mathbf{v}_P + \alpha_Q \mathbf{v}_Q + \alpha_R \mathbf{v}_R = \mathbf{0}$ with not all the αs being 0, then
(a) *none* of the αs are 0, and
(b) the point Q is an affine combination of P and R; in particular, $Q = -\frac{\alpha_P}{\alpha_Q} P - \frac{\alpha_R}{\alpha_P} R$, so Q must lie on the line between P and R.
(c) Argue why dependence and collinearity are trivially the same if two or more of the points P, Q, and R are identical.

Exercise 10.10: It's good to be able to recognize the transformation represented by a matrix by looking at the matrix; for instance, it's easy to recognize a 3×3 matrix that represents a translation in homogeneous coordinates: Its last row is $\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$ and its upper-left 2×2 block is the identity. Given a 3×3 matrix representing a transformation in homogeneous coordinates,
(a) how can you tell whether the transformation is affine or not?
(b) How can you tell whether the transformation is linear or not?
(c) How can you tell whether it represents a rotation about the origin?
(d) How can you tell if it represents a uniform scale?

Exercise 10.11: Suppose we have a linear transformation $T : \mathbf{R^2} \to \mathbf{R^2}$, and two coordinate systems with bases $\{\mathbf{u}_1, \mathbf{u}_2\}$ and $\{\mathbf{v}_1, \mathbf{v}_2\}$; all four basis vectors are unit vectors, $\mathbf{u}_2$ is 90° counterclockwise from $\mathbf{u}_1$, and similarly $\mathbf{v}_2$ is 90° counterclockwise from $\mathbf{v}_1$. You can write down the matrix $\mathbf{M}_u$ for T in the u-coordinate system and the matrix $\mathbf{M}_v$ for T in the v-coordinate system.

(a) If $\mathbf{M}_u$ is a rotation matrix $\begin{bmatrix} \cos\theta & \sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$, what can you say about $\mathbf{M}_v$?

(b) If $\mathbf{M}_u$ is a uniform scaling matrix, that is, a multiple of the identity, what can you say about $\mathbf{M}_v$?

(c) If $\mathbf{M}_u$ is a *non*uniform scaling matrix of the form $\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}$, with $a \neq b$, what can you say about $\mathbf{M}_v$?

Chapter 15

Ray Casting and Rasterization

15.1 Introduction

Previous chapters considered modeling and interacting with 2D and 3D scenes using an underlying renderer provided by WPF. Now we focus on writing our own physically based 3D renderer.

Rendering is integration. To compute an image, we need to compute how much light arrives at each pixel of the image sensor inside a virtual camera. Photons transport the light energy, so we need to simulate the physics of the photons in a scene. However, we can't possibly simulate *all* of the photons, so we need to sample a few of them and generalize from those to estimate the integrated arriving light. Thus, one might also say that rendering is sampling. We'll tie this integration notion of sampling to the alternative probability notion of sampling presently.

In this chapter, we look at two strategies for sampling the amount of light transported along a ray that arrives at the image plane. These strategies are called **ray casting** and **rasterization.** We'll build software renderers using each of them. We'll also build a third renderer using a hardware rasterization API. All three renderers can be used to sample the light transported to a point from a specific direction. A point and direction define a ray, so in graphics jargon, such sampling is typically referred to as "sampling along a ray," or simply "sampling a ray."

There are many interesting rays along which to sample transport, and the methods in this chapter generalize to all of them. However, here we focus specifically on sampling rays within a cone whose apex is at a point light source or a pinhole camera aperture. The techniques within these strategies can also be modified and combined in interesting ways. Thus, the essential idea of this chapter is that rather than facing a choice between distinct strategies, you stand to gain a set of tools that you can modify and apply to any rendering problem. We emphasize two aspects in the presentation: the principle of sampling as a mathematical tool and the practical details that arise in implementing real renderers.

Of course, we'll take many chapters to resolve the theoretical and practical issues raised here. Since graphics is an active field, some issues will not be thoroughly resolved even by the end of the book. In the spirit of servicing both principles and practice, we present some ideas first with pseudocode and mathematics and then second in actual compilable code. Although minimal, that code follows reasonable software engineering practices, such as data abstraction, to stay true to the feel of a real renderer. If you create your own programs from these pieces (which you should) and add the minor elements that are left as exercises, then you will have three working renderers at the end of the chapter. Those will serve as a scalable code base for your implementation of other algorithms presented in this book.

The three renderers we build will be simple enough to let you quickly understand and implement them in one or two programming sessions each. By the end of the chapter, we'll clean them up and generalize the designs. This generality will allow us to incorporate changes for representing complex scenes and the data structures necessary for scaling performance to render those scenes.

We assume that throughout the subsequent rendering chapters you are implementing each technique as an extension to one of the renderers that began in this chapter. As you do, we recommend that you adopt two good software engineering practices.

1. Make a copy of the renderer before changing it (this copy becomes the **reference renderer**).
2. Compare the image result after a change to the preceding, reference result.

Techniques that enhance performance should generally not reduce image quality. Techniques that enhance simulation accuracy should produce noticeable and measurable improvements. By comparing the "before" and "after" rendering performance and image quality, you can verify that your changes were implemented correctly.

Comparison begins right in this chapter. We'll consider three rendering strategies here, but all should generate identical results. We'll also generalize each strategy's implementation once we've sketched it out. When debugging your own implementations of these, consider how incorrectly mismatched results between programs indicate potential underlying program errors. This is yet another instance of the Visual Debugging principle.

15.2 High-Level Design Overview

We start with a high-level design in this section. We'll then pause to address the practical issues of our programming infrastructure before reducing that high-level design to the specific sampling strategies.

15.2.1 Scattering

Light that enters the camera and is measured arrives from points on surfaces in the scene that either scattered or emitted the light. Those points lie along the rays that we choose to sample for our measurement. Specifically, the points casting light into the camera are the intersections in the scene of rays, whose origins are points on the image plane, that passed through the camera's aperture.

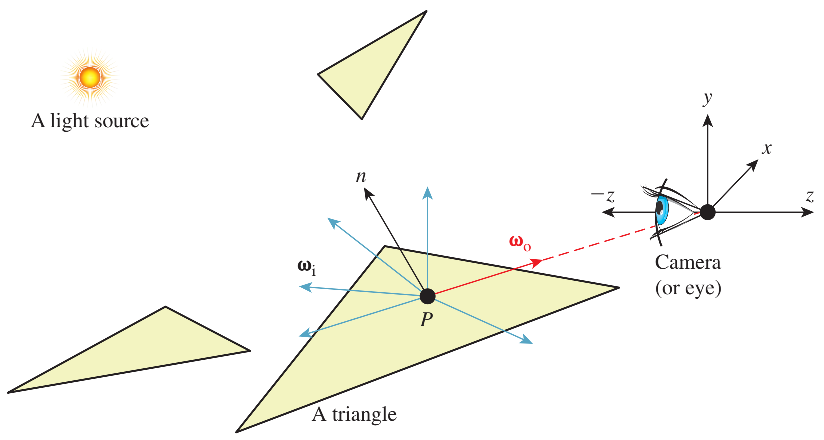


Figure 15.1: A specific surface location P that is visible to the camera, incident light at P from various directions $\{\omega_i\}$*, and the exitant direction* ω_o *toward the camera.*

To keep things simple, we assume a pinhole camera with a virtual image plane *in front* of the center of projection, and an instantaneous exposure. This means that there will be no blur in the image due to defocus or motion. Of course, an image with a truly zero-area aperture and zero-time exposure would capture zero photons, so we take the common graphics approximation of estimating the result of a *small* aperture and exposure from the limiting case, which is conveniently possible in simulation, albeit not in reality.

We also assume that the virtual sensor pixels form a regular square grid and estimate the value that an individual pixel would measure using a single sample at the center of that pixel's square footprint. Under these assumptions, our sampling rays are the ones with origins at the center of projection (i.e., the pinhole) and directions through each of the sensor-pixel centers.[1]

Finally, to keep things simple we chose a coordinate frame in which the center of projection is at the origin and the camera is looking along the negative z-axis. We'll also refer to the center of projection as the eye. See Section 15.3.3 for a formal description and Figure 15.1 for a diagram of this configuration.

The light that arrived at a specific sensor pixel from a scene point P came from some direction. For example, the direction from the brightest light source in the scene provided a lot of light. But not all light arrived from the brightest source. There may have been other light sources in the scene that were dimmer. There was also probably a lot of light that previously scattered at other points and arrived at P indirectly. This tells us two things. First, we ultimately have to consider all possible directions from which light may have arrived at P to generate a correct image. Second, if we are willing to accept some sampling error, then we can select a finite number of discrete directions to sample. Furthermore, we can

1. For the advanced reader, we acknowledge Alvy Ray Smith's "a pixel is not a little square"—that is, no sample is useful without its reconstruction filter—but contend that Smith was so successful at clarifying this issue that today "sample" now is properly used to describe the point-sample data to which Smith referred, and "pixel" now is used to refer to a "little square area" of a display or sensor, whose value may be estimated from samples. We'll generally use "sensor pixel" or "display pixel" to mean the physical entity and "pixel" for the rectangular area on the image plane.

probably rank the importance of those directions, at least for lights, and choose a subset that is likely to minimize sampling error.

Inline Exercise 15.1: We don't expect you to have perfect answers to these, but we want you to think about them now to help develop intuition for this problem: What kind of errors could arise from sampling a finite number of directions? What makes them errors? What might be good sampling strategies? How do the notions of expected value and variance from statistics apply here? What about statistical independence and bias?

Let's start by considering all possible directions for incoming light in pseudocode and then return to the ranking of discrete directions when we later need to implement directional sampling concretely.

To consider the points and directions that affect the image, our program has to look something like Listing 15.1.

Listing 15.1: High-level rendering structure.

```
for each visible point $P$ with direction $\omega_o$ from it to pixel center $(x,y)$:
    $sum$ = 0
    for each incident light direction $\omega_i$ at $P$:
        $sum$ += light scattered at $P$ from $\omega_i$ to $\omega_o$
    $pixel$[$x$, $y$] = $sum$
```

15.2.2 Visible Points

Now we devise a strategy for representing points in the scene, finding those that are visible and scattering the light incident on them to the camera.

For the scene representation, we'll work within some of the common rendering approximations described in Chapter 14. None of these are so fundamental as to prevent us from later replacing them with more accurate models.

Assume that we only need to model surfaces that form the boundaries of objects. "Object" is a subjective term; a **surface** is technically the interface between volumes with homogeneous physical properties. Some of these objects are what everyday language recognizes as such, like a block of wood or the water in a pool. Others are not what we are accustomed to considering as objects, such as air or a vacuum.

We'll model these surfaces as triangle meshes. We ignore the surrounding medium of air and assume that all the meshes are closed so that from the outside of an object one can never see the inside. This allows us to consider only single-sided triangles. We choose the convention that the vertices of a triangular face, seen from the outside of the object, are in counterclockwise order around the face. To approximate the shading of a smooth surface using this triangle mesh, we model the surface normal at a point on a triangle pointing in the direction of the barycentric interpolation of prespecified normal vectors at its vertices. These normals only affect shading, so silhouettes of objects will still appear polygonal.

Chapter 27 explores how surfaces scatter light in great detail. For simplicity, we begin by assuming all surfaces scatter incoming light equally in all directions, in a sense that we'll make precise presently. This kind of scattering is called Lambertian, as you saw in Chapter 6, so we're rendering a Lambertian surface. The color of a surface is determined by the relative amount of light scattered at each wavelength, which we represent with a familiar RGB triple.

This surface mesh representation describes all the *potentially* visible points at the set of locations $\{P\}$. To render a given pixel, we must determine which potentially visible points project to the center of that pixel. We then need to select the scene point closest to the camera. That point is the *actually* visible point for the pixel center. The radiance—a measure of light that's defined precisely in Section 26.7.2, and usually denoted with the letter L—arriving from that point and passing through the pixel is proportional to the light incident on the point and the point's reflectivity.

To find the nearest potentially visible point, we first convert the outer loop of Listing 15.1 (see the next section) into an iteration over both pixel centers (which correspond to rays) and triangles (which correspond to surfaces). A common way to accomplish this is to replace "`for each visible point`" with two nested loops, one over the pixel centers and one over the triangles. Either can be on the outside. Our choice of which is the new outermost loop has significant structural implications for the rest of the renderer.

15.2.3 Ray Casting: Pixels First

Listing 15.2: Ray-casting pseudocode.

```
for each pixel position $(x, y)$:
    let $R$ be the ray through $(x, y)$ from the eye
    for each triangle $T$:
        let $P$ be the intersection of $R$ and $T$ (if any)
        $sum$ = 0
        for each direction:
           $sum$ += ...
        if $P$ is closer than previous intersections at this pixel:
            $pixel$[x, y] = $sum$
```

Consider the strategy where the outermost loop is over pixel centers, shown in Listing 15.2. This strategy is called **ray casting** because it creates one ray per pixel and casts it at every surface. It generalizes to an algorithm called **ray tracing,** in which the innermost loop recursively casts rays at each direction, but let's set that aside for the moment.

Ray casting lets us process each pixel to completion independently. This suggests parallel processing of pixels to increase performance. It also encourages us to keep the entire scene in memory, since we don't know which triangles we'll need at each pixel. The structure suggests an elegant way of eventually processing the aforementioned indirect light: Cast more rays from the innermost loop.

15.2.4 Rasterization: Triangles First

Now consider the strategy where the outermost loop is over triangles shown in Listing 15.3. This strategy is called **rasterization,** because the inner loop is typically implemented by marching along the rows of the image, which are called **rasters.** We could choose to march along columns as well. The choice of rows is historical and has to do with how televisions were originally constructed. Cathode ray tube (CRT) displays scanned an image from left to right, top to bottom, the way that English text is read on a page. This is now a widespread convention:

Unless there is an explicit reason to do otherwise, images are stored in row-major order, where the element corresponding to 2D position (x, y) is stored at index `(x + y * width)` in the array.

Listing 15.3: Rasterization pseudocode; O denotes the origin, or eyepoint.

```
 1  for each pixel position $(x,y)$:
 2      $closest[x,y] = \infty$
 3
 4  for each triangle $T$:
 5      for each pixel position $(x,y)$:
 6          let $R$ be the ray through $(x,y)$ from the eye
 7          let $P$ be the intersection of $R$ and $T$
 8          if $P$ exists:
 9              $sum$ = 0
10              for each direction:
11                  $sum$ += ...
12                  if the distance to $P$ is less than $closest[x,y]$:
13                      $pixel[x,y] = sum$
14                      $closest[x,y] = |P - O|$
```

Rasterization allows us to process each triangle to completion independently.[2] This has several implications. It means that we can render much larger scenes than we can hold in memory, because we only need space for one triangle at a time. It suggests triangles as the level of parallelism. The properties of a triangle can be maintained in registers or cache to avoid memory traffic, and only one triangle needs to be memory-resident at a time. Because we consider adjacent pixels consecutively for a given triangle, we can approximate derivatives of arbitrary expressions across the surface of a triangle by finite differences between pixels. This is particularly useful when we later become more sophisticated about sampling strategies because it allows us to adapt our sampling rate to the rate at which an underlying function is changing in screen space.

Note that the conditional on line 12 in Listing 15.3 refers to the closest *previous* intersection at a pixel. Because that intersection was from a different triangle, that value must be stored in a 2D array that is parallel to the image. This array did not appear in our original pseudocode or the ray-casting design. Because we now touch each pixel many times, we must maintain a data structure for each pixel that helps us resolve visibility between visits. Only two distances are needed for comparison: the distance to the current point and to the previously closest point. We don't care about points that have been previously considered but are farther away than the closest, because they are hidden behind the closest point and can't affect the image. The *closest* array stores the distance to the previously closest point at each pixel. It is called a **depth buffer** or a **z-buffer.** Because computing the distance to a point is potentially expensive, depth buffers are often implemented to encode some other value that has the same comparison properties as distance along a ray. Common choices are $-z_P$, the z-coordinate of the point P, and $-1/z_P$. Recall that the camera is facing along the negative z-axis, so these are related to distance from the $z = 0$ plane in which the camera sits.

2. If you're worried that to process one triangle we have to loop through all the pixels in the image, even though the triangle does not cover most of them, then your worries are well founded. See Section 15.6.2 for a better strategy. We're starting this way to keep the code as nearly parallel to the ray-casting structure as possible.

For now we'll use the more intuitive choice of distance from P to the origin, $|P - O|$.

The depth buffer has the same dimensions as the image, so it consumes a potentially significant amount of memory. It must also be accessed atomically under a parallel implementation, so it is a potentially slow synchronization point. Chapter 36 describes alternative algorithms for resolving visibility under rasterization that avoid these drawbacks. However, depth buffers are by far the most widely used method today. They are extremely efficient in practice and have predictable performance characteristics. They also offer advantages beyond the sampling process. For example, the known depth at each pixel at the end of 3D rendering yields a "2.5D" result that enables compositing of multiple render passes and post-processing filters, such as artificial defocus blur.

This depth comparison turns out to be a fundamental idea, and it is now supported by special fixed-function units in graphics hardware. A huge leap in computer graphics performance occurred when this feature emerged in the early 1980s.

15.3 Implementation Platform

15.3.1 Selection Criteria

The kinds of choices discussed in this section are important. We want to introduce them now, and we want them all in one place so that you can refer to them later. Many of them will only seem natural to you after you've worked with graphics for a while. So read this section now, set it aside, and then read it again in a month.

In your studies of computer graphics you will likely learn many APIs and software design patterns. For example, Chapters 2, 4, 6, and 16 teach the 2D and 3D WPF APIs and some infrastructure built around them.

Teaching that kind of content is expressly *not* a goal of this chapter. This chapter is about creating algorithms for sampling light. The implementation serves to make the algorithms concrete and provide a test bed for later exploration. Although learning a specific platform is not a goal, learning the issues to consider when evaluating a platform *is* a goal; in this section we describe those issues.

We select one specific platform, a subset of the G3D Innovation Engine [http://g3d.sf.net] Version 9, for the code examples. You may use this one, or some variation chosen after considering the same issues weighed by your own goals and computing environment. In many ways it is better if your platform—language, compiler, support classes, hardware API—is *not* precisely the same as the one described here. The platform we select includes only a minimalist set of support classes. This keeps the presentation simple and generic, as suits a textbook. But you're developing software on today's technology, not writing a textbook that must serve independent of currently popular tools.

Since you're about to invest a lot of work on this renderer, a richer set of support classes will make both implementation and debugging easier. You can compile our code directly against the support classes in G3D. However, if you have to rewrite it slightly for a different API or language, this will force you to actually read every line and consider why it was written in a particular manner. Maybe your chosen language has a different syntax than ours for passing a parameter by value instead of reference, for example. In the process of redeclaring a parameter to make this syntax change, you should think about why the parameter was passed by value in the first place, and whether the computational overhead or software abstraction of doing so is justified.

To avoid distracting details, for the low-level renderers we'll write the image to an array in memory and then stop. Beyond a trivial PPM-file writing routine, we will not discuss the system-specific methods for saving that image to disk or displaying it on-screen in this chapter. Those are generally straightforward, but verbose to read and tedious to configure. The PPM routine is a proof of concept, but it is for an inefficient format and requires you to use an external viewer to check each result. G3D and many other platforms have image-display and image-writing procedures that can present the images that you've rendered more conveniently.

For the API-based hardware rasterizer, we will use a lightly abstracted subset of the OpenGL API that is representative of most other hardware APIs. We'll intentionally skip the system-specific details of initializing a hardware context and exploiting features of a particular API or GPU. Those transient aspects can be found in your favorite API or GPU vendor's manuals.

Although we can largely ignore the surrounding platform, we must still choose a programming language. It is wise to choose a language with reasonably high-level abstractions like classes and operator overloading. These help the algorithm shine through the source code notation.

It is also wise to choose a language that can be compiled to efficient native code. That is because even though performance should not be the ultimate consideration in graphics, it is a fairly important one. Even simple video game scenes contain millions of polygons and are rendered for displays with millions of pixels. We'll start with one triangle and one pixel to make debugging easier and then quickly grow to hundreds of each in this chapter. The constant overhead of an interpreted language or a managed memory system cannot affect the asymptotic behavior of our program. However, it can be the difference between our renderer producing an image in two seconds or two hours . . . and debugging a program that takes two hours to run is very unpleasant.

Computer graphics code tends to combine high-level classes containing significant state, such as those representing scenes and objects, with low-level classes (a.k.a. "records", "structs") for storing points and colors that have little state and often expose that which they do contain directly to the programmer. A real-time renderer can easily process billions of those low-level classes per second. To support that, one typically requires a language with features for efficiently creating, destroying, and storing such classes. Heap memory management for small classes tends to be expensive and thwart cache efficiency, so stack allocation is typically the preferred solution. Language features for passing by value and by constant reference help the programmer to control cloning of both large and small class instances.

Finally, hardware APIs tend to be specified at the machine level, in terms of bytes and pointers (as abstracted by the C language). They also often require manual control over memory allocation, deallocation, types, and mapping to operate efficiently.

To satisfy the demands of high-level abstraction, reasonable performance for hundreds to millions of primitives and pixels, and direct manipulation of memory, we work within a subset of C++. Except for some minor syntactic variations, this subset should be largely familiar to Java and Objective C++ programmers. It is a superset of C and can be compiled directly as native (nonmanaged) C#. For all of these reasons, and because there is a significant tools and library ecosystem built for it, C++ happens to be the dominant language for implementing renderers today. Thus, our choice is consistent with showing you how renderers are really implemented.

Note that many hardware APIs also have wrappers for higher-level languages, provided by either the API vendor or third parties. Once you are familiar with the basic functionality, we suggest that it may be more productive to use such a wrapper for extensive software development on a hardware API.

15.3.2 Utility Classes

This chapter assumes the existence of obvious utility classes, such as those sketched in Listing 15.4. For these, you can use equivalents of the WPF classes, the Direct3D API versions, the built-in GLSL, Cg, and HLSL shading language versions, or the ones in G3D, or you can simply write your own. Following common practice, the `Vector3` and `Color3` classes denote the axes over which a quantity varies, but not its units. For example, `Vector3` always denotes three spatial axes but may represent a unitless direction vector at one code location and a position in meters at another. We use a type alias to at least distinguish points from vectors (which are differences of points).

Listing 15.4: Utility classes.

```
#define INFINITY (numeric_limits<float>::infinity())

class Vector2 { public: float x, y;    ... };
class Vector3 { public: float x, y, z; ... };
typedef Vector2 Point2;
typedef Vector3 Point3;
class Color3 { public: float r, g, b;  ... };
class Radiance3 Color3;
class Power3 Color3;

class Ray {
private:
    Point3  m_origin;
    Vector3 m_direction;

public:
    Ray(const Point3& org, const Vector3& dir) :
       m_origin(org), m_direction(dir) {}

    const Point3& origin() const { return m_origin; }
    const Vector3& direction() const { return m_direction; }
    ...
};
```

Observe that some classes, such as `Vector3`, expose their representation through public member variables, while others, such as `Ray`, have a stronger abstraction that protects the internal representation behind methods. The exposed classes are the workhorses of computer graphics. Invoking methods to access their fields would add significant syntactic distraction to the implementation of any function. Since the byte layouts of these classes must be known and fixed to interact directly with hardware APIs, they cannot be strong abstractions and it makes sense to allow direct access to their representation. The classes that protect their representation are ones whose representation we may (and truthfully, will) later want to change. For example, the internal representation of `Triangle` in this listing is an array of vertices. If we found that we computed the edge vectors or face normal frequently, then it might be more efficient to extend the representation to explicitly store those values.

For images, we choose the underlying representation to be an array of `Radiance3`, each array entry representing the radiance incident at the center of one pixel in the image. We then wrap this array in a class to present it as a 2D structure with appropriate utility methods in Listing 15.5.

Listing 15.5: An `Image` *class.*

```
class Image {
private:
   int                    m_width;
   int                    m_height;
   std::vector<Radiance3> m_data;

   int PPMGammaEncode(float radiance, float displayConstant) const;

public:

   Image(int width, int height) :
        m_width(width), m_height(height), m_data(width * height) {}

   int width() const { return m_width; }

   int height() const { return m_height; }

   void set(int x, int y, const Radiance3& value) {
      m_data[x + y * m_width] = value;
   }

   const Radiance3& get(int x, int y) const {
      return m_data[x + y * m_width];
   }

   void save(const std::string& filename, float displayConstant=15.0f) const;
};
```

Under C++ conventions and syntax, the `&` following a type in a declaration indicates that the corresponding variable or return value will be passed by reference. The `m_` prefix avoids confusion between member variables and methods or parameters with similar names. The `std::vector` class is the dynamic array from the standard library.

One could imagine a more feature-rich image class with bounds checking, documentation, and utility functions. Extending the implementation with these is a good exercise.

The `set` and `get` methods follow the historical row-major mapping from a 2D to a 1D array. Although we do not need it here, note that the reverse mapping from a 1D index `i` to the 2D indices `(x, y)` is

```
x = i % width; y = i / width
```

where `%` is the C++ integer modulo operation.

When `width` is a power of two, that is, `width` $= 2^k$, it is possible to perform both the forward and reverse mappings using bitwise operations, since

$$a \mod 2^k = a \,\&\, (2^k - 1) \tag{15.1}$$

$$a/2^k = a \gg k \tag{15.2}$$

$$a \cdot 2^k = a \ll k, \tag{15.3}$$

for fixed-point values. Here we use » as the operator to shift the bits of the left operand to the right by the value of the right operand, and & as the bitwise AND operator.

This is one reason that many graphics APIs historically required power-of-two image dimensions (another is MIP mapping). One can always express a number that is not a power of two as the sum of multiple powers of two. In fact, that's what binary encoding does! For example, $640 = 512 + 128$, so $x + 640y = x + (y \ll 9) + (y \ll 7)$.

Inline Exercise 15.2: Implement forward and backward mappings from integer (x, y) pixel locations to 1D array indices i, for a typical HD resolution of 1920×1080, using only bitwise operations, addition, and subtraction.

Familiarity with the bit-manipulation methods for mapping between 1D and 2D arrays is important now so that you can understand other people's code. It will also help you to appreciate how hardware-accelerated rendering might implement some low-level operations and why a rendering API might have certain constraints. However, this kind of micro-optimization will not substantially affect the performance of your renderer at this stage, so it is not yet worth including.

Our `Image` class stores physically meaningful values. The natural measurement of the light arriving along a ray is in terms of **radiance,** whose definition and precise units are described in Chapter 26. The image typically represents the light *about* to fall onto each pixel of a sensor or area of a piece of film. It doesn't represent the sensor response process.

Displays and image files tend to work with arbitrarily scaled 8-bit display values that map nonlinearly to radiance. For example, if we set the display pixel value to 64, the display pixel does not emit twice the radiance that it does when we set the same pixel to 32. This means that we cannot display our image faithfully by simply rescaling radiance to display values. In fact, the relationship involves an exponent commonly called gamma, as described briefly below and at length in Section 28.12.

Assume some multiplicative factor `d` that rescales the radiance values in an image so that the largest value we wish to represent maps to 1.0 and the smallest maps to 0.0. This fills the role of the camera's shutter and aperture. The user will select this value as part of the scene definition. Mapping it to a GUI slider is often a good idea.

Historically, most images stored 8-bit values whose meanings were ill-specified. Today it is more common to specify what they mean. An image that actually stores radiance values is informally said to store **linear radiance,** indicating that the pixel value varies linearly with the radiance (see Chapter 17). Since the radiance range of a typical outdoor scene with shadows might span six orders of magnitude, the data would suffer from perceptible quantization artifacts were it reduced to eight bits per channel. However, human perception of brightness is roughly logarithmic. This means that distributing precision nonlinearly can reduce the perceptual error of a small bit-depth approximation. **Gamma encoding** is a common practice for distributing values according to a fractional power law, where $1/\gamma$ is the power. This encoding curve roughly matches the logarithmic response curve of the human visual system. Most computer displays accept input already gamma-encoded along the sRGB standard curve, which is about $\gamma = 2.2$. Many image file formats, such as PPM, also default to this gamma encoding. A routine that maps a radiance value to an 8-bit display value with a gamma value of 2.2 is:

```
int Image::PPMGammaEncode(float radiance, float d) const {
    return int(pow(std::min(1.0f, std::max(0.0f, radiance * d)),
                   1.0f / 2.2f) * 255.0f);
}
```

Note that $x^{1/2.2} \approx \sqrt{x}$. Because they are faster than arbitrary exponentiation on most hardware, square root and square are often employed in real-time rendering as efficient $\gamma = 2.0$ encoding and decoding methods.

The `save` routine is our bare-bones method for exporting data from the renderer for viewing. It saves the image in human-readable PPM format [P+10] and is implemented in Listing 15.6.

Listing 15.6: Saving an image to an ASCII RGB PPM file.

```
void Image::save(const std::string& filename, float d) const {
    FILE* file = fopen(filename.c_str(), "wt");
    fprintf(file, "P3 %d %d 255\n", m_width, m_height);
    for (int y = 0; y < m_height; ++y) {
        fprintf(file, "\n# y = %d\n", y);
        for (int x = 0; x < m_width; ++x) {
            const Radiance3& c(get(x, y));
            fprintf(file, "%d %d %d\n",
                    PPMGammaEncode(c.r, d),
                    PPMGammaEncode(c.g, d),
                    PPMGammaEncode(c.b, d));
        }
    }
    fclose(file);
}
```

This is a useful snippet of code beyond its immediate purpose of saving an image. The structure appears frequently in 2D graphics code. The outer loop iterates over rows. It contains any kind of per-row computation (in this case, printing the row number). The inner loop iterates over the columns of one row and performs the per-pixel operations. Note that if we wished to amortize the cost of computing `y * m_width` inside the `get` routine, we could compute that as a per-row operation and merely accumulate the 1-pixel offsets in the inner loop. We do not do so in this case because that would complicate the code without providing a measurable performance increase, since writing a formatted text file would remain slow compared to performing one multiplication per pixel.

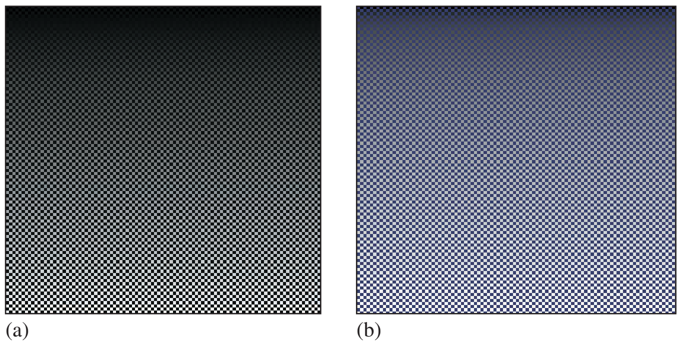


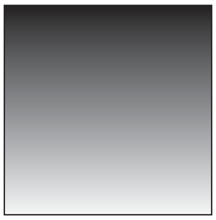
Figure 15.2: A pattern for testing the `Image` *class. The pattern is a checkerboard of* 1*-pixel squares that alternate between* $1/10\ \mathrm{W}/(\mathrm{m}^2\ \mathrm{sr})$ *in the blue channel and a vertical gradient from* 0 *to* 10*. (a) Viewed with* `deviceGamma` = 1.0 *and* `displayConstant` = 1.0*, which makes dim squares appear black and gives the appearance of a linear change in* brightness*. (b) Displayed more correctly with* `deviceGamma` = 2.0*, where the linear* radiance *gradient correctly appears as a nonlinear brightness ramp and the dim squares are correctly visible. (The conversion to a printed image or your online image viewer may further affect the image.)*

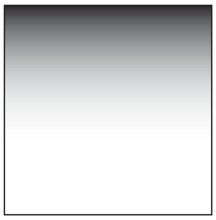
The PPM format is slow for loading and saving, and consumes lots of space when storing images. For those reasons, it is rarely used outside academia. However, it is convenient for data interchange between programs. It is also convenient for debugging small images for three reasons. The first is that it is easy to read and write. The second is that many image programs and libraries support it, including Adobe Photoshop and xv. The third is that we can open it in a text editor to look directly at the (gamma-corrected) pixel values.

After writing the image-saving code, we displayed the simple pattern shown in Figure 15.2 as a debugging aid. If you implement your own image saving or display mechanism, consider doing something similar. The test pattern alternates dark blue pixels with ones that form a gradient. The reason for creating the single-pixel checkerboard pattern is to verify that the image was neither stretched nor cropped during display. If it was, then one or more thin horizontal or vertical lines would appear. (If you are looking at this image on an electronic display, you may see such patterns, indicating that your viewing software is indeed stretching it.) The motivation for the gradient is to determine whether gamma correction is being applied correctly. A linear radiance gradient should appear as a nonlinear brightness gradient, when displayed correctly. Specifically, it should primarily look like the brighter shades. The pattern on the left is shown without gamma correction. The gradient appears to have linear brightness, indicating that it is not displayed correctly. The pattern on the right is shown with gamma correction. The gradient correctly appears to be heavily shifted toward the brighter shaders.

Note that we made the darker squares blue, yet in the left pattern—without gamma correction—they appear black. That is because gamma correction helps make darker shades more visible, as in the right image. This hue shift is another argument for being careful to always implement gamma correction, beyond the tone shift. Of course, we don't know the exact characteristics of the display (although one can typically determine its gamma exponent) or the exact viewing conditions of the room, so precise color correction and tone mapping is beyond our ability here. However, the simple act of applying gamma correction arguably captures some of the most important aspects of that process and is computationally inexpensive and robust.

Inline Exercise 15.3: Two images are shown below. Both have been gamma-encoded with $\gamma = 2.0$ for printing and online display. The image on the left is a gradient that has been rendered to give the impression of linear *brightness*. It should appear as a linear color ramp. The image on the right was rendered with linear *radiance* (it is the checkerboard on the right of Figure 15.2 without the blue squares). It should appear as a nonlinear color ramp. The image was rendered at 200×200 pixels. What equation did we use to compute the value (in $[0,1]$) of the pixel at (x,y) for the gradient image on the left?


Linear brightness


Linear radiance

15.3.3 Scene Representation

Listing 15.7 shows a `Triangle` class. It stores each triangle by explicitly storing each vertex. Each vertex has an associated normal that is used exclusively for shading; the normals do not describe the actual geometry. These are sometimes called **shading normals.** When the vertex normals are identical to the normal to the plane of the triangle, the triangle's shading will appear consistent with its actual geometry. When the normals diverge from this, the shading will mimic that of a curved surface. Since the silhouette of the triangle will still be polygonal, this effect is most convincing in a scene containing many small triangles.

Listing 15.7: Interface for a `Triangle` *class.*

```
class Triangle {
private:
    Point3    m_vertex[3];
    Vector3   m_normal[3];
    BSDF      m_bsdf;

public:

    const Point3& vertex(int i) const { return m_vertex[i]; }
    const Vector3& normal(int i) const { return m_normal[i]; }
    const BSDF& bsdf() const { return m_bsdf; }
    ...
};
```

We also associate a `BSDF` class value with each triangle. This describes the material properties of the surface modeled by the triangle. It is described in Section 15.4.5. For now, think of this as the color of the triangle.

The representation of the triangle is concealed by making the member variables private. Although the implementation shown contains methods that simply return those member variables, you will later use this abstraction boundary to create a more efficient implementation of the triangle. For example, many triangles may share the same vertices and bidirectional scattering distribution functions (BSDFs), so this representation is not very space-efficient. There are also properties of the triangle, such as the edge lengths and geometric normal, that we will find ourselves frequently recomputing and could benefit from storing explicitly.

Inline Exercise 15.4: Compute the size in bytes of one `Triangle`. How big is a 1M triangle mesh? Is that reasonable? How does this compare with the size of a stored mesh file, say, in the binary 3DS format or the ASCII OBJ format? What are other advantages, beyond space reduction, of sharing vertices between triangles in a mesh?

Listing 15.8 shows our implementation of an omnidirectional point light source. We represent the power it emits at three wavelengths (or in three wavelength bands), and the center of the emitter. Note that emitters are infinitely small in our representation, so they are not themselves visible. If we wish to see the source appear in the final rendering we need to either add geometry around it or explicitly render additional information into the image. We will do neither explicitly in this chapter, although you may find that these are necessary when debugging your illumination code.

Listing 15.8: Interface for a uniform point luminaire—a light source.

```
class Light {
public:
    Point3    position;

    /** Over the entire sphere. */
    Power3    power;
};
```

Listing 15.9 describes the scene as sets of triangles and lights. Our choice of arrays for the implementation of these sets imposes an ordering on the scene. This is convenient for ensuring a reproducible environment for debugging. However, for now we are going to create that ordering in an arbitrary way, and that choice may affect performance and even our image in some slight ways, such as resolving ties between which surface is closest at an intersection. More sophisticated scene data structures may include additional structure in the scene and impose a specific ordering.

Listing 15.9: Interface for a scene represented as an unstructured list of triangles and light sources.

```
class Scene {
public:
    std::vector<Triangle> triangleArray;
    std::vector<Light>    lightArray;
};
```

Listing 15.10 represents our camera. The camera has a pinhole aperture, an instantaneous shutter, and artificial near and far planes of constant (negative) z values. We assume that the camera is located at the origin and facing along the $-z$-axis.

Listing 15.10: Interface for a pinhole camera at the origin.

```
class Camera {
public:
    float zNear;
    float zFar;
    float fieldOfViewX;

    Camera() : zNear(-0.1f), zFar(-100.0f), fieldOfViewX(PI / 2.0f) {}
};
```

We constrain the horizontal field of view of the camera to be `fieldOfViewX`. This is the measure of the angle from the center of the leftmost pixel to the center of the rightmost pixel along the horizon in the camera's view in radians (it is shown later in Figure 15.3). During rendering, we will compute the aspect ratio of the target image and implicitly use that to determine the vertical field of view. We could alternatively specify the vertical field of view and compute the horizontal field of view from the aspect ratio.

15.3.4 A Test Scene

We'll test our renderers on a scene that contains one triangle whose vertices are

`Point3(0,1,-2)`, `Point3(-1.9,-1,-2)`, and `Point3(1.6,-0.5,-2)`,

and whose vertex normals are

```
Vector3( 0.0f, 0.6f, 1.0f).direction(),
Vector3(-0.4f,-0.4f, 1.0f).direction(), and
Vector3( 0.4f,-0.4f, 1.0f).direction().
```

We create one light source in the scene, located at `Point3(1.0f,3.0f,1.0f)` and emitting power `Power3(10, 10, 10)`. The camera is at the origin and is facing along the $-z$-axis, with y increasing upward in screen space and x increasing to the right. The image has size 800×500 and is initialized to dark blue.

This choice of scene data was deliberate, because when debugging it is a good idea to choose configurations that use nonsquare aspect ratios, nonprimary colors, asymmetric objects, etc. to help find cases where you have accidentally swapped axes or color channels. Having distinct values for the properties of each vertex also makes it easier to track values through code. For example, on this triangle, you can determine which vertex you are examining merely by looking at its x-coordinate.

On the other hand, the camera is the standard one, which allows us to avoid transforming rays and geometry. That leads to some efficiency and simplicity in the implementation and helps with debugging because the input data maps exactly to the data rendered, and in practice, most rendering algorithms operate in the camera's reference frame anyway.

Inline Exercise 15.5: *Mandatory; do not continue until you have done this:* Draw a schematic diagram of this scene from three viewpoints.

1. The orthographic view from infinity facing along the x-axis. Make z increase to the right and y increase upward. Show the camera and its field of view.
2. The orthographic view from infinity facing along the $-y$-axis. Make x increase to the right and z increase downward. Show the camera and its field of view. Draw the vertex normals.
3. The perspective view from the camera, facing along the $-z$-axis; the camera should not appear in this image.

15.4 A Ray-Casting Renderer

We begin the ray-casting renderer by expanding and implementing our initial pseudocode from Listing 15.2. It is repeated in Listing 15.11 with more detail.

Listing 15.11: Detailed pseudocode for a ray-casting renderer.

```
for each pixel row $y$:
   for each pixel column $x$:
      let $R$ = ray through screen space position $(x+0.5, y+0.5)$
      $closest$ = $\infty$
      for each triangle $T$:
         $d$ = intersect($T$, $R$)
         if ($d$ < $closest$)
             $closest$ = $d$
             $sum$ = 0
             let $P$ be the intersection point
             for each direction $\omega_i$:
                $sum$ += light scattered at $P$ from $\omega_i$ to $\omega_o$

             $image[x, y] = sum$
```

The three loops iterate over every ray and triangle combination. The body of the for-each-triangle loop verifies that the new intersection is closer than previous observed ones, and then shades the intersection. We will abstract the operation of ray intersection and sampling into a helper function called `sampleRayTriangle`. Listing 15.12 gives the interface for this helper function.

Listing 15.12: Interface for a function that performs ray-triangle intersection and shading.

```
bool sampleRayTriangle(const Scene& scene, int x, int y,
    const Ray& R, const Triangle& T,
    Radiance3& radiance, float& distance);
```

The specification for `sampleRayTriangle` is as follows. It tests a particular ray against a triangle. If the intersection exists and is closer than all previously observed intersections for this ray, it computes the radiance scattered toward the viewer and returns `true`. The innermost loop therefore sets the value of pixel (x, y)

to the radiance `L_o` passing through its center from the closest triangle. Radiance from farther triangles is not interesting because it will (conceptually) be blocked by the back of the closest triangle and never reach the image. The implementation of `sampleRayTriangle` appears in Listing 15.15.

To render the entire image, we must invoke `sampleRayTriangle` once for each pixel center and for each triangle. Listing 15.13 defines `rayTrace`, which performs this iteration. It takes as arguments a box within which to cast rays (see Section 15.4.4). We use `L_o` to denote the radiance from the triangle; the subscript "o" is for "outgoing".

Listing 15.13: Code to trace one ray for every pixel between (`x0`, `y0`) *and* (`x1-1`, `y1-1`)*, inclusive.*

```
/** Trace eye rays with origins in the box from [x0, y0] to (x1, y1).*/
void rayTrace(Image& image, const Scene& scene,
  const Camera& camera, int x0, int x1, int y0, int y1) {

  // For each pixel
  for (int y = y0; y < y1; ++y) {
    for (int x = y0; x < x1; ++x) {

      // Ray through the pixel
      const Ray& R = computeEyeRay(x + 0.5f, y + 0.5f, image.width(),
                                   image.height(), camera);

      // Distance to closest known intersection
      float distance = INFINITY;
      Radiance3 L_o;

      // For each triangle
      for (unsigned int t = 0; t < scene.triangleArray.size(); ++t){
        const Triangle& T = scene.triangleArray[t];

        if (sampleRayTriangle(scene, x, y, R, T, L_o, distance)) {
          image.set(x, y, L_o);
        }
      }
    }
  }
}
```

To invoke `rayTrace` on the entire image, we will use the call:

```
rayTrace(image, scene, camera, 0, image.width(), 0, image.height());
```

15.4.1 Generating an Eye Ray

Assume the camera's center of projection is at the origin, $(0,0,0)$, and that, in the camera's frame of reference, the y-axis points upward, the x-axis points to the right, and the z-axis points out of the screen. Thus, the camera is facing along its own $-z$-axis, in a right-handed coordinate system. We can transform any scene to this coordinate system using the transformations from Chapter 11.

We require a utility function, `computeEyeRay`, to find the ray through the center of a pixel, which in screen space is given by $(x+0.5, y+0.5)$ for integers x and y. Listing 15.14 gives an implementation. The key geometry is depicted in

Figure 15.3. The figure is a top view of the scene in which x increases to the right and z increases downward. The near plane appears as a horizontal line, and the `start` point is on that plane, along the line from the camera at the origin to the center of a specific pixel.

To implement this function we needed to parameterize the camera by either the image plane depth or the desired field of view. Field of view is a more intuitive way to specify a camera, so we previously chose that parameterization when building the scene representation.

Listing 15.14: Computing the ray through the center of pixel (x, y) on a `width` $\times$ `height` *image.*

```
Ray computeEyeRay(float x, float y, int width, int height, const Camera& camera) {
    const float aspect = float(height) / width;

    // Compute the side of a square at z = -1 based on our
    // horizontal left-edge-to-right-edge field of view
    const float s = -2.0f * tan(camera.fieldOfViewX * 0.5f);

    const Vector3& start =
        Vector3( (x / width - 0.5f) * s,
                -(y / height - 0.5f) * s * aspect, 1.0f) * camera.zNear;

    return Ray(start, start.direction());
}
```

We choose to place the ray origin on the near (sometimes called hither) clipping plane, at $z =$ `camera.zNear`. We could start rays at the origin instead of the near plane, but starting at the near plane will make it easier for results to line up precisely with our rasterizer later.

The ray direction is the direction from the center of projection (which is at the origin, $(0, 0, 0)$) to the ray `start` point, so we simply normalize `start` point.

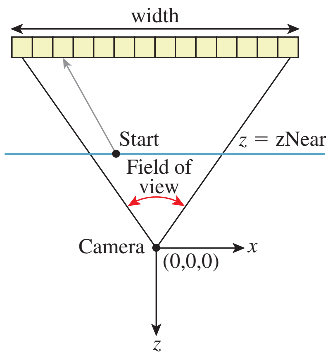


Figure 15.3: The ray through a pixel center in terms of the image resolution and the camera's horizontal field of view.

Inline Exercise 15.6: By the rules of Chapter 7, we should compute the ray direction as `(start - Vector3(0,0,0)).direction()`. That makes the camera position explicit, so we are less likely to introduce a bug if we later change the camera. This arises simply from strongly typing the code to match the underlying mathematical types. On the other hand, our code is going to be full of lines like this, and consistently applying correct typing might lead to more harm from obscuring the algorithm than benefit from occasionally finding an error. It is a matter of personal taste and experience (we can somewhat reconcile our typing with the math by claiming that `P.direction()` on a point `P` returns the direction to the point, rather than "normalizing" the point).

Rewrite `computeEyeRay` using the distinct `Point` and `Vector` abstractions from Chapter 7 to get a feel for how this affects the presentation and correctness. If this inspires you, it's quite reasonable to restructure all the code in this chapter that way, and doing so is a valuable exercise.

Note that the y-coordinate of the `start` is negated. This is because `y` is in 2D screen space, with a "$y =$ down" convention, and the ray is in a 3D coordinate system with a "$y =$ up" convention.

To specify the vertical field of view instead of the horizontal one, replace `fieldOfViewX` with `fieldOfViewY` and insert the line `s /= aspect`.

15.4.1.1 Camera Design Notes

The C++ language offers both functions and methods as procedural abstractions. We have presented `computeEyeRay` as a function that takes a `Camera` parameter to distinguish the "support code" `Camera` class from the ray-tracer-specific code that you are adding. As you move forward through the book, consider refactoring the support code to integrate auxiliary functions like this directly into the appropriate classes. (If you are using an existing 3D library as support code, it is likely that the provided camera class already contains such a method. In that case, it is worth implementing the method once as a function here so that you have the experience of walking through and debugging the routine. You can later discard your version in favor of a canonical one once you've reaped the educational value.)

A software engineering tip: Although we have chosen to forgo small optimizations, it is still important to be careful to use references (e.g., `Image&`) to avoid excess copying of arguments and intermediate results. There are two related reasons for this, and neither is about the performance of *this* program.

The first reason is that we want to be in the habit of avoiding excessive copying. A `Vector3` occupies 12 bytes of memory, but a full-screen `Image` is a few megabytes. If we're conscientious about never copying data unless we want copy semantics, then we won't later accidentally copy an `Image` or other large structure. Memory allocation and copy operations can be surprisingly slow and will bloat the memory footprint of our program. The time cost of copying data isn't just a constant overhead factor on performance. Copying the image once per pixel, in the inner loop, would change the ray caster's asymptotic run time from $\mathrm{O}(n)$ in the number of pixels to $\mathrm{O}(n^2)$.

The second reason is that experienced programmers rely on a set of idioms that are chosen to avoid bugs. Any deviation from those attracts attention, because it is a potential bug. One such convention in C++ is to pass each value as a const reference unless otherwise required, for the long-term performance reasons just described. So code that doesn't do so takes longer for an experienced programmer to review because of the need to check that there isn't an error or performance implication whenever an idiom is not followed. If you are an experienced C++ programmer, then such idioms help you to read the code. If you are not, then either ignore all the ampersands and treat this as pseudocode, or use it as an opportunity to become a better C++ programmer.

15.4.1.2 Testing the Eye-Ray Computation

We need to test `computeEyeRay` before continuing. One way to do this is to write a unit test that computes the eye rays for specific pixels and then compares them to manually computed results. That is always a good testing strategy. In addition to that, we can visualize the eye rays. Visualization is a good way to quickly see the result of many computations. It allows us to more intuitively check results, and to identify patterns of errors if they do not match what we expected.

In this section, we'll visualize the directions of the rays. The same process can be applied to the origins. The directions are the more common location for an error and conveniently have a bounded range, which make them both more important and easier to visualize.

A natural scheme for visualizing a direction is to interpret the (x, y, z) fields as (r, g, b) color triplets. The conversion of ray direction to pixel color is of course a gross violation of units, but it is a really useful debugging technique and we aren't expecting anything principled here anyway.

Because each ordinate is on the interval $[-1, 1]$, we rescale them to the range $[0, 1]$ by $r = (x + 1)/2$. Our image display routines also apply an exposure function, so we need to scale the resultant intensity down by a constant on the order of the inverse of the exposure value. Temporarily inserting the following line:

```
image.set(x, y, Color3(R.direction() + Vector3(1, 1, 1)) / 5);
```

into `rayTrace` in place of the `sampleRayTriangle` call should yield an image like that shown in Figure 15.4. (The factor of $1/5$ scales the debugging values to a reasonable range for our output, which was originally calibrated for radiance; we found a usable constant for this particular example by trial and error.) We expect the x-coordinate of the ray, which here is visualized as the color red, to increase from a minimum on the left to a maximum on the right. Likewise, the (3D) y-coordinate, which is visualized as green, should increase from a minimum at the bottom of the image to a maximum at the top. If your result varies from this, examine the pattern you observe and consider what kind of error could produce it. We will revisit visualization as a debugging technique later in this chapter, when testing the more complex intersection routine.

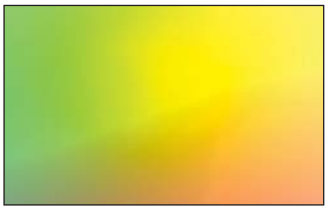


Figure 15.4: Visualization of eye-ray directions.

15.4.2 Sampling Framework: Intersect and Shade

Listing 15.15 shows the code for sampling a triangle with a ray. This code doesn't perform any of the heavy lifting itself. It just computes the values needed for `intersect` and `shade`.

Listing 15.15: Sampling the intersection and shading of one triangle with one ray.

```
bool sampleRayTriangle(const Scene& scene, int x, int y, const Ray& R,
     const Triangle& T, Radiance3& radiance, float& distance) {

    float weight[3];
    const float d = intersect(R, T, weight);

    if (d >= distance) {
        return false;
    }

    // This intersection is closer than the previous one
    distance = d;

    // Intersection point
    const Point3& P = R.origin() + R.direction() * d;

    // Find the interpolated vertex normal at the intersection
    const Vector3& n = (T.normal(0) * weight[0] +
                        T.normal(1) * weight[1] +
                        T.normal(2) * weight[2]).direction();

    const Vector3& w_o = -R.direction();

23      shade(scene, T, P, n, w_o, radiance);
24
25      // Debugging intersect: set to white on any intersection
26      //radiance = Radiance3(1, 1, 1);
27
28      // Debugging barycentric
29      //radiance = Radiance3(weight[0], weight[1], weight[2]) / 15;
30
31      return true;
32  }
```

The `sampleRayTriangle` routine returns `false` if there was no intersection closer than `distance`; otherwise, it updates `distance` and `radiance` and returns `true`.

When invoking this routine, the caller passes the `distance` to the closest currently known intersection, which is initially `INFINITY` (let `INFINITY = std::numeric_limits<T>::infinity()` in C++, or simply `1.0/0.0`). We will design the `intersect` routine to return `INFINITY` when no intersection exists between `R` and `T` so that a missed intersection will never cause `sampleRayTriangle` to return `true`.

Placing the `(d >= distance)` test before the shading code is an optimization. We would still obtain correct results if we always computed the shading before testing whether the new intersection is in fact the closest. This is an important optimization because the `shade` routine may be arbitrarily expensive. In fact, in a full-featured ray tracer, almost all computation time is spent inside `shade`, which recursively samples additional rays. We won't discuss further shading optimizations in this chapter, but you should be aware of the importance of an early termination when another surface is known to be closer.

Note that the order of the triangles in the calling routine (`rayTrace`) affects the performance of the routine. If the triangles are in back-to-front order, then we will shade each one, only to reject all but the closest. This is the worst case. If the triangles are in front-to-back order, then we will shade the first and reject the rest without further shading effort. We could ensure the best performance always by separating `sampleRayTriangle` into two auxiliary routines: one to find the closest intersection and one to shade that intersection. This is a common practice in ray tracers. Keep this in mind, but do not make the change yet. Once we have written the rasterizer renderer, we will consider the space and time implications of such optimizations under both ray casting and rasterization, which gives insights into many variations on each algorithm.

We'll implement and test `intersect` first. To do so, comment out the call to `shade` on line 23 and uncomment either of the debugging lines below it.

15.4.3 Ray-Triangle Intersection

We'll find the intersection of the eye ray and a triangle in two steps, following the method described in Section 7.9 and implemented in Listing 15.16. This method first intersects the line containing the ray with the plane containing the triangle. It then solves for the barycentric weights to determine if the intersection is within the triangle. We need to ignore intersections with the back of the single-sided triangle and intersections that occur along the part of the line that is not on the ray.

The same weights that we use to determine if the intersection is within the triangle are later useful for interpolating per-vertex properties, such as shading

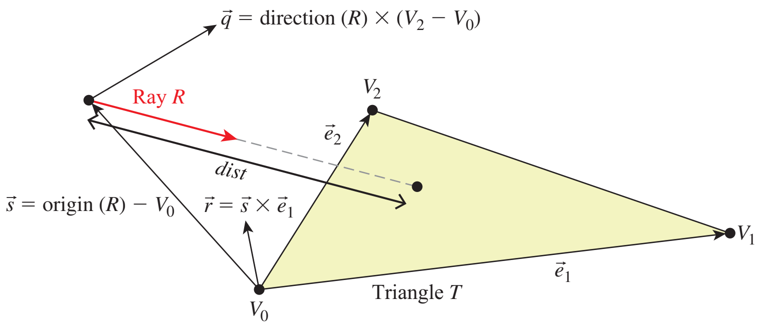

$\vec{q}$ = direction $(R) \times (V_2 - V_0)$
Ray R
V_2
$\vec{e}_2$
$dist$
$\vec{s}$ = origin $(R) - V_0$
$\vec{r} = \vec{s} \times \vec{e}_1$
V_1
$\vec{e}_1$
Triangle T
V_0

Figure 15.5: Variables for computing the intersection of a ray and a triangle (see Listing 15.16).

normals. We structure our implementation to return the weights to the caller. The caller could use either those or the distance traveled along the ray to find the intersection point. We return the distance traveled because we know that we will later need that anyway to identify the closest intersection to the viewer in a scene with many triangles. We return the barycentric weights for use in interpolation.

Figure 15.5 shows the geometry of the situation. Let R be the ray and T be the triangle. Let $\vec{e}_1$ be the edge vector from V_0 to V_1 and $\vec{e}_2$ be the edge vector from V_0 to V_2. Vector $\vec{q}$ is orthogonal to both the ray and $\vec{e}_2$. Note that if $\vec{q}$ is also orthogonal to $\vec{e}_1$, then the ray is parallel to the triangle and there is no intersection. If $\vec{q}$ is in the negative hemisphere of $\vec{e}_1$ (i.e., "points away"), then the ray travels away from the triangle.

Vector $\vec{s}$ is the displacement of the ray origin from V_0, and vector $\vec{r}$ is the cross product of $\vec{s}$ and $\vec{e}_1$. These vectors are used to construct the barycentric weights, as shown in Listing 15.16.

Variable `a` is the rate at which the ray is approaching the triangle, multiplied by twice the area of the triangle. This is not obvious from the way it is computed here, but it can be seen by applying a triple-product identity relation:

Let $d =$ `R.direction()`

Let $area = |\vec{e}_2 \times \vec{e}_1|/2$

$$a = \vec{e}_1 \cdot q = \vec{e}_1 \cdot d \times \vec{e}_2 = d \cdot \vec{e}_2 \times \vec{e}_1 = -(d \cdot n) \cdot 2 \cdot area, \tag{15.4}$$

since the direction of $\vec{e}2 \times \vec{e}1$ is opposite the triangle's geometric normal n. The particular form of this expression chosen in the implementation is convenient because the `q` vector is needed again later in the code for computing the barycentric weights.

There are several cases where we need to compare a value against zero. The two `epsilon` constants guard these comparisons against errors arising from limited numerical precision.

The comparison `a <= epsilon` detects two cases. If `a` is zero, then the ray is parallel to the triangle and never intersects it. In this case, the code divided by zero many times, so other variables may be infinity or not-a-number. That's irrelevant, since the first test expression will still make the entire test expression `true`. If `a` is negative, then the ray is traveling away from the triangle and will never intersect it. Recall that `a` is the rate at which the ray approaches the triangle, multiplied by the area of the triangle. If `epsilon` is too large, then intersections with triangles

Listing 15.16: Ray-triangle intersection (derived from [MT97])

```
float intersect(const Ray& R, const Triangle& T, float weight[3]) {
    const Vector3& e1 = T.vertex(1) - T.vertex(0);
    const Vector3& e2 = T.vertex(2) - T.vertex(0);
    const Vector3& q = R.direction().cross(e2);

    const float a = e1.dot(q);

    const Vector3& s = R.origin() - T.vertex(0);
    const Vector3& r = s.cross(e1);

    // Barycentric vertex weights
    weight[1] = s.dot(q) / a;
    weight[2] = R.direction().dot(r) / a;
    weight[0] = 1.0f - (weight[1] + weight[2]);

    const float dist = e2.dot(r) / a;

    static const float epsilon = 1e-7f;
    static const float epsilon2 = 1e-10;

    if ((a <= epsilon) || (weight[0] < -epsilon2) ||
        (weight[1] < -epsilon2) || (weight[2] < -epsilon2) ||
        (dist <= 0.0f)) {
        // The ray is nearly parallel to the triangle, or the
        // intersection lies outside the triangle or behind
        // the ray origin: "infinite" distance until intersection.
        return INFINITY;
    } else {
        return dist;
    }
}
```

will be missed at glancing angles, and this missed intersection behavior will be more likely to occur at triangles with large areas than at those with small areas. Note that if we changed the test to `fabs(a)<= epsilon`, then triangles would have two sides. This is not necessary for correct models of real, opaque objects; however, for rendering mathematical models or models with errors in them it can be convenient. Later we will depend on optimizations that allow us to quickly cull the (approximately half) of the scene representing back faces, so we choose to render single-sided triangles here for consistency.

The `epsilon2` constant allows a ray to intersect a triangle slightly outside the bounds of the triangle. This ensures that triangles that share an edge completely cover pixels along that edge despite numerical precision limits. If `epsilon2` is too small, then single-pixel holes will very occasionally appear on that edge. If it is too large, then all triangles will visibly appear to be too large.

Depending on your processor architecture, it may be faster to perform an early test and potential return rather than allowing not-a-number and infinity propagation in the ill-conditioned case where `a` ≈ 0. Many values can also be precomputed, for example, the edge lengths of the triangle, or at least be reused within a single intersection, for example, `1.0f / a`. There's a cottage industry of optimizing this intersection code for various architectures, compilers, and scene types (e.g., [MT97] for scalar processors versus [WBB08] for vector processors). Let's forgo those low-level optimizations and stick to high-level algorithmic decisions. In practice, most ray casters spend very little time in the ray intersection code anyway. The fastest way to determine if a ray intersects a triangle is to never ask that question in the first place. That is, in Chapter 37, we will introduce data structures that quickly and conservatively eliminate whole sets of triangles that the ray could not possibly intersect, without ever performing the ray-triangle intersection. So optimizing this routine now would only complicate it without affecting our long-term performance profile.

Our renderer only processes triangles. We could easily extend it to render scenes containing any kind of primitive for which we can provide a ray intersection solution. Surfaces defined by low-order equations, like the plane, rectangle, sphere, and cylinder, have explicit solutions. For others, such as bicubic patches, we can use root-finding methods.

15.4.4 Debugging

We now verify that the intersection code code is correct. (The code we've given you *is* correct, but if you invoked it with the wrong parameters, or introduced an error when porting to a different language or support code base, then you need to learn how to find that error.) This is a good opportunity for learning some additional graphics debugging tricks, all of which demonstrate the Visual Debugging principle.

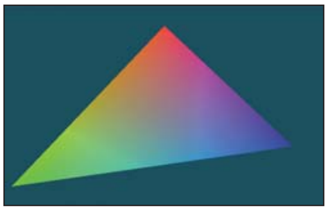


Figure 15.6: The single triangle scene visualized with color equal to barycentric weight for debugging the intersection code.

It would be impractical to manually examine every intersection result in a debugger or printout. That is because the `rayTrace` function invokes `intersect` thousands of times. So instead of examining individual results, we visualize the barycentric coordinates by setting the radiance at a pixel to be proportional to the barycentric coordinates following the Visual Debugging principle. Figure 15.6 shows the correct resultant image. If your program produces a comparable result, then your program is probably nearly correct.

What should you do if your result looks different? You can't examine every result, and if you place a breakpoint in `intersect`, then you will have to step through hundreds of ray casts that miss the triangle before coming to the interesting intersection tests.

This is why we structured `rayTrace` to trace within a caller-specified rectangle, rather than the whole image. We can invoke the ray tracer on a single pixel from `main()`, or better yet, create a debugging interface where clicking on a pixel with the mouse invokes the single-pixel trace on the selected pixel. By setting breakpoints or printing intermediate results under this setup, we can investigate why an artifact appears at a specific pixel. For one pixel, the math is simple enough that we can also compute the desired results by hand and compare them to those produced by the program.

In general, even simple graphics programs tend to have large amounts of data. This may be many triangles, many pixels, or many frames of animation. The processing for these may also be running on many threads, or on a GPU. Traditional debugging methods can be hard to apply in the face of such numerous data and massive parallelism. Furthermore, the graphics development environment may preclude traditional techniques such as printing output or setting breakpoints. For example, under a hardware rendering API, your program is executing on an embedded processor that frequently has no access to the console and is inaccessible to your debugger.

Fortunately, three strategies tend to work well for graphics debugging.

1. Use assertions liberally. These cost you nothing in the optimized version of the program, pass silently in the debug version when the program operates

correctly, and break the program at the test location when an assertion is violated. Thus, they help to identify failure cases without requiring that you manually step through the correct cases.

2. Immediately reduce to the minimal test case. This is often a single-triangle scene with a single light and a single pixel. The trick here is to find the combination of light, triangle, and pixel that produces incorrect results. Assertions and the GUI click-to-debug scheme work well for that.
3. Visualize intermediate results. We have just rendered an image of the barycentric coordinates of eye-ray intersections with a triangle for a 400,000-pixel image. Were we to print out these values or step through them in the debugger, we would have little chance of recognizing an incorrect value in that mass of data. If we see, for example, a black pixel, or a white pixel, or notice that the red and green channels are swapped, then we may be able to deduce the nature of the error that caused this, or at least know which inputs cause the routine to fail.

15.4.5 Shading

We are now ready to implement `shade`. This routine computes the incident radiance at the intersection point `P` and how much radiance scatters back along the eye ray to the viewer.

Let's consider only light transport paths directly from the source to the surface to the camera. Under this restriction, there is no light arriving at the surface from any directions except those to the lights. So we only need to consider a finite number of ω_i values. Let's also assume for the moment that there is always a line of sight to the light. This means that there will (perhaps incorrectly) be no shadows in the rendered image.

Listing 15.17 iterates over the light sources in the scene (note that we have only one in our test scene). For each light, the loop body computes the distance and direction to that light from the point being shaded. Assume that lights emit uniformly in all directions and are at finite locations in the scene. Under these assumptions, the incident radiance `L_i` at point `P` is proportional to the total power of the source divided by the square of the distance between the source and `P`. This is because at a given distance, the light's power is distributed equally over a sphere of that radius. Because we are ignoring shadowing, let the `visible` function always return `true` for now. In the future it will return `false` if there is no line of sight from the source to `P`, in which case the light should contribute no incident radiance.

The outgoing radiance to the camera, `L_o`, is the sum of the fraction of incident radiance that scatters in that direction. We abstract the scattering function into a BSDF. We implement this function as a class so that it can maintain state across multiple invocations and support an inheritance hierarchy. Later in this book, we will also find that it is desirable to perform other operations beyond invoking this function; for example, we might want to sample with respect to the probability distribution it defines. Using a class representation will allow us to later introduce additional methods for these operations.

The `evaluateFiniteScatteringDensity` method of that class evaluates the scattering function for the given incoming and outgoing angles. We always then take the product of this and the incoming radiance, modulated by the cosine

Listing 15.17: The single-bounce shading code.

```
void shade(const Scene& scene, const Triangle& T, const Point3& P,
   const Vector3& n, const Vector3& w_o, Radiance3& L_o) {

    L_o = Color3(0.0f, 0.0f, 0.0f);

    // For each direction (to a light source)
    for (unsigned int i = 0; i < scene.lightArray.size(); ++i) {
       const Light& light = scene.lightArray[i];

       const Vector3& offset = light.position - P;
       const float distanceToLight = offset.length();
       const Vector3& w_i = offset / distanceToLight;

       if (visible(P, w_i, distanceToLight, scene)) {
           const Radiance3& L_i = light.power / (4 * PI * square(distanceToLight));

             // Scatter the light
             L_o +=
                L_i *
                T.bsdf(n).evaluateFiniteScatteringDensity(w_i, w_o) *
                max(0.0, dot(w_i, n));
       }
    }
}
```

of the angle between `w_i` and `n` to account for the projected area over which incident radiance is distributed (by the Tilting principle).

15.4.6 Lambertian Scattering

The simplest implementation of the BSDF assumes a surface appears to be the same brightness independent of the viewer's orientation. That is, `evaluateFiniteScatteringDensity` returns a constant. This is called **Lambertian reflectance,** and it is a good model for matte surfaces such as paper and flat wall paint. It is also trivial to implement. Listing 15.18 gives the implementation (see Section 14.9.1 for a little more detail and Chapter 29 for a lot more). It has a single member, `lambertian`, that is the "color" of the surface. For energy conservation, this value should have all fields on the range $[0, 1]$.

Listing 15.18: Lambertian BSDF implementation, following Listing 14.6.

```
class BSDF {
public:
    Color3 k_L;

    /** Returns f = L_o / (L_i * w_i.dot(n)) assuming
    incident and outgoing directions are both in the
    positive hemisphere above the normal */
    Color3 evaluateFiniteScatteringDensity
      (const Vector3& w_i, const Vector3& w_o) const {
        return k_L / PI;
    }
};
```

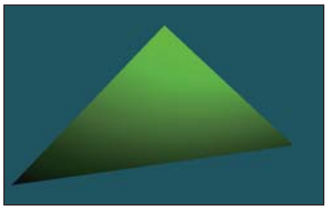


Figure 15.7: A green Lambertian triangle.

Figure 15.7 shows our triangle scene rendered with the Lambertian BSDF using `k_L=Color3(0.0f, 0.0f, 0.8f)`. Because our triangle's vertex normals are deflected away from the plane defined by the vertices, the triangle appears curved. Specifically, the bottom of the triangle is darker because the `w_i.dot(n)` term in line 20 of Listing 15.17 falls off toward the bottom of the triangle.

15.4.7 Glossy Scattering

The Lambertian surface appears dull because it has no highlight. A common approach for producing a more interesting shiny surface is to model it with something like the Blinn-Phong scattering function. An implementation of this function with the energy conservation factor from Sloan and Hoffmann [AMHH08, 257] is given in Listing 15.19. See Chapter 27 for a discussion of the origin of this function and alternatives to it. This is a variation on the shading function that we saw back in Chapter 6 in WPF, only now we are implementing it instead of just adjusting the parameters on a black box. The basic idea is simple: Extend the Lambertian BSDF with a large radial peak when the normal lies close to halfway between the incoming and outgoing directions. This peak is modeled by a cosine raised to a power since that is easy to compute with dot products. It is scaled so that the outgoing radiance never exceeds the incoming radiance and so that the sharpness and total intensity of the peak are largely independent parameters.

Listing 15.19: Blinn-Phong BSDF scattering density.

```
class BSDF {
public:
    Color3  k_L;
    Color3  k_G;
    float   s;
    Vector3 n;
    ...

    Color3 evaluateFiniteScatteringDensity(const Vector3& w_i,
        const Vector3& w_o) const {
        const Vector3& w_h = (w_i + w_o).direction();
        return
            (k_L + k_G * ((s + 8.0f) *
                powf(std::max(0.0f, w_h.dot(n)), s) / 8.0f)) /
            PI;

    }
};
```

For this BSDF, choose `lambertian` + `glossy` $<$ 1 at each color channel to ensure energy conservation, and `glossySharpness` typically in the range $[0, 2000]$. The `glossySharpness` is on a logarithmic scale, so it must be moved in larger increments as it becomes larger to have the same perceptual impact.

Figure 15.8 shows the green triangle rendered with the normalized Blinn-Phong BSDF. Here, `k_L=Color3(0.0f, 0.0f, 0.8f)`, `k_G=Color3(0.2f, 0.2f, 0.2f)`, and `s=100.0f`.

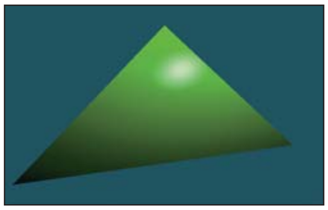


Figure 15.8: Triangle rendered with a normalized Blinn-Phong BSDF.

15.4.8 Shadows

The `shade` function in Listing 15.17 only adds the illumination contribution from a light source if there is an unoccluded line of sight between that source and the point P being shaded. Areas that *are* occluded are therefore darker. This absence of light is the phenomenon that we recognize as a shadow.

In our implementation, the line-of-sight visibility test is performed by the `visible` function, which is supposed to return `true` if and only if there is an unoccluded line of sight. While working on the shading routine we temporarily implemented `visible` to *always* return `true`, which means that our images contain no shadows. We now revisit the `visible` function in order to implement shadows.

We already have a powerful tool for evaluating line of sight: the `intersect` function. The light source is not visible from *P* if there is some intersection with another triangle. So we can test visibility simply by iterating over the scene again, this time using the *shadow ray* from *P* to the light instead of from the camera to *P*. Of course, we could also test rays from the light *to P*.

Listing 15.20 shows the implementation of `visible`. The structure is very similar to that of `sampleRayTriangle`. It has three major differences in the details. First, instead of shading the intersection, if we find any intersection we immediately return `false` for the visibility test. Second, instead of casting rays an infinite distance, we terminate when they have passed the light source. That is because we don't care about triangles past the light—they could not possibly cast shadows on *P*. Third and finally, we don't really start our shadow ray cast at *P*. Instead, we offset it slightly along the ray direction. This prevents the ray from reintersecting the surface containing *P* as soon as it is cast.

Listing 15.20: Line-of-sight visibility test, to be applied to shadow determination.

```
bool visible(const Vector3& P, const Vector3& direction, float
    distance, const Scene& scene){
    static const float rayBumpEpsilon = 1e-4;
    const Ray shadowRay(P + direction * rayBumpEpsilon, direction);

    distance -= rayBumpEpsilon;

    // Test each potential shadow caster to see if it lies between P and the light
    float ignore[3];
    for (unsigned int s = 0; s < scene.triangleArray.size(); ++s) {
        if (intersect(shadowRay, scene.triangleArray[s], ignore) < distance) {
            // This triangle is closer than the light
            return false;
        }
    }

    return true;
}
```

Our single-triangle scene is insufficient for testing shadows. We require one object to cast shadows and another to receive them. A simple extension is to add a quadrilateral "ground plane" onto which the green triangle will cast its shadow. Listing 15.21 gives code to create this scene. Note that this code also adds another triangle with the same vertices as the green one but the opposite winding order. Because our triangles are single-sided, the green triangle would not cast a shadow. We need to add the back of that surface, which will occlude the rays cast upward toward the light from the ground.

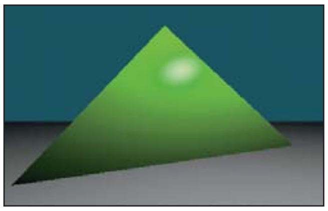


Figure 15.9: The green triangle scene extended with a two-triangle gray ground "plane." A back surface has also been added to the green triangle.

Inline Exercise 15.7: Walk through the intersection code to verify the claim that without the second "side," the green triangle would cast no shadow.

Figure 15.9 shows how the new scene should render *before* you implement shadows. If you do not see the ground plane under your own implementation, the most likely error is that you failed to loop over all triangles in one of the ray-casting routines.

Listing 15.21: Scene-creation code for a two-sided triangle and a ground plane.

```
void makeOneTriangleScene(Scene& s) { s.triangleArray.resize(1);

    s.triangleArray[0] =
      Triangle(Vector3(0,1,-2), Vector3(-1.9,-1,-2), Vector3(1.6,-0.5,-2),
         Vector3(0,0.6f,1).direction(),
         Vector3(-0.4f,-0.4f, 1.0f).direction(),
         Vector3(0.4f,-0.4f, 1.0f).direction(),
         BSDF(Color3::green() * 0.8f,Color3::white() * 0.2f, 100));

    s.lightArray.resize(1);
    s.lightArray[0].position = Point3(1, 3, 1);
    s.lightArray[0].power = Color3::white() * 10.0f;
}

void makeTrianglePlusGroundScene(Scene& s) {
    makeOneTriangleScene(s);

    // Invert the winding of the triangle
    s.triangleArray.push_back
      (Triangle(Vector3(-1.9,-1,-2), Vector3(0,1,-2),
          Vector3(1.6,-0.5,-2), Vector3(-0.4f,-0.4f, 1.0f).direction(),
          Vector3(0,0.6f,1).direction(), Vector3(0.4f,-0.4f, 1.0f).direction(),
          BSDF(Color3::green() * 0.8f,Color3::white() * 0.2f, 100)));

    // Ground plane
    const float groundY = -1.0f;
    const Color3 groundColor = Color3::white() * 0.8f;
    s.triangleArray.push_back
      (Triangle(Vector3(-10, groundY, -10), Vector3(-10, groundY, -0.01f),
          Vector3(10, groundY, -0.01f),
          Vector3::unitY(), Vector3::unitY(), Vector3::unitY(), groundColor));

    s.triangleArray.push_back
      (Triangle(Vector3(-10, groundY, -10), Vector3(10, groundY, -0.01f),
          Vector3(10, groundY, -10),
          Vector3::unitY(), Vector3::unitY(), Vector3::unitY(), groundColor));
}
```

Figure 15.10 shows the scene rendered with `visible` implemented correctly. If the `rayBumpEpsilon` is too small, then **shadow acne** will appear on the green triangle. This artifact is shown in Figure 15.11. An alternative to starting the ray artificially far from *P* is to explicitly exclude the previous triangle from the shadow ray intersection computation. We chose not to do that because, while appropriate for unstructured triangles, it would be limiting to maintain that custom ray intersection code as our scene became more complicated. For example, we would like to later abstract the scene data structure from a simple array of triangles. The abstract data structure might internally employ a hash table or tree and have complex methods. Pushing the notion of excluding a surface into such a data structure could complicate that data structure and compromise its general-purpose use. Furthermore, although we are rendering only triangles now,

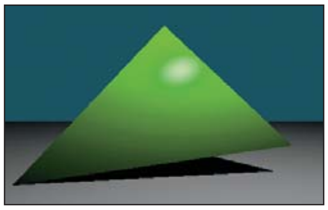


Figure 15.10: A four-triangle scene, with ray-cast shadows implemented via the `visible` function. The green triangle is two-sided.

we might wish to render other primitives in the future, such as spheres or implicit surfaces. Such primitives can intersect a ray multiple times. If we assume that the shadow ray never intersects the current surface, those objects would never self-shadow.

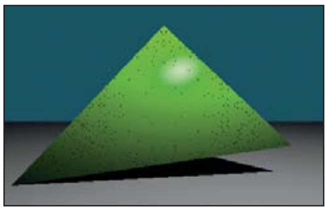


*Figure 15.11: The dark dots on the green triangle are **shadow acne** caused by self-shadowing. This artifact occurs when the shadow ray immediately intersects the triangle that was being shaded.*

15.4.9 A More Complex Scene

Now that we've built a renderer for one or two triangles, it is no more difficult to render scenes containing many triangles. Figure 15.12 shows a shiny, gold-colored teapot on a white ground plane. We parsed a file containing the vertices of the corresponding triangle mesh, appended those triangles to the `Scene`'s triangle array, and then ran the existing renderer on it. This scene contains about 100 triangles, so it renders about 100 times slower than the single-triangle scene. We can make arbitrarily more complex geometry and shading functions for the renderer. We are only limited by the quality of our models and our rendering performance, both of which will be improved in subsequent chapters.

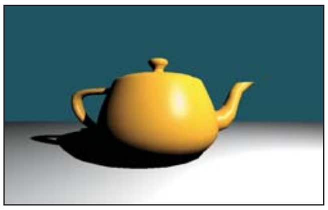


Figure 15.12: A scene composed of many triangles.

This scene looks impressive (at least, relative to the single triangle) for two reasons. First, we see some real-world phenomena, such as shiny highlights, shadows, and nice gradients as light falls off. These occurred naturally from following the geometric relationships between light and surfaces in our implementation.

Second, the image resembles a recognizable object, specifically, a teapot. Unlike the illumination phenomena, nothing in *our* code made this look like a teapot. We simply loaded a triangle list from a data file that someone (originally, Jim Blinn) happened to have manually constructed. This teapot triangle list is a classic model in graphics. You can download the triangle mesh version used here from `http://graphics.cs.williams.edu/data` among other sources. Creating models like this is a separate problem from rendering, discussed in Chapter 22 and many others. Fortunately, there are many such models available, so we can defer the modeling problem while we discuss rendering.

We can learn a lesson from this. A strength and weakness of computer graphics as a technical field is that often the data contributes more to the quality of the final image than the algorithm. The same algorithm rendered the teapot and the green triangle, but the teapot looks more impressive because the data is better. Often a truly poor approximation algorithm will produce stunning results when a master artist creates the input—the commercial success of the film and game industries has largely depended on this fact. Be aware of this when judging algorithms based on rendered results, and take advantage of it by importing good artwork to demonstrate your own algorithms.

15.5 Intermezzo

To render a scene, we needed to iterate over both triangles and pixels. In the previous section, we arbitrarily chose to arrange the pixel loop on the outside and the triangle loop on the inside. That yielded the ray-casting algorithm. The ray-casting algorithm has three nice properties: It somewhat mimics the underlying physics, it separates the visibility routine from the shading routine, and it leverages the same ray-triangle intersection routine for both eye rays and shadow rays.

Admittedly, the relationship between ray casting and physics at the level demonstrated here is somewhat tenuous. Real photons propagate along rays from the light source to a surface to an eye, and we traced that path backward. Real photons don't all scatter into the camera. Most photons from the light source scatter away from the camera, and much of the light that *is* scattered toward the camera from a surface didn't arrive at that surface directly from the light. Nonetheless, an algorithm for sampling light along rays is a very good starting point for sampling photons, and it matches our intuition about how light should propagate. You can probably imagine improvements that would better model the true scattering behavior of light. Much of the rest of this book is devoted to such models.

In the next section, we invert the nesting order of the loops to yield a **rasterizer algorithm.** We then explore the implications of that change. We already have a working ray tracer to compare against. Thus, we can easily test the correctness of our changes by comparing against the ray-traced image and intermediate results. We also have a standard against which to measure the properties of the new algorithm. As you read the following section and implement the program that it describes, consider how the changes you are making affect code clarity, modularity, and efficiency. Particularly consider efficiency in both a wall-clock time and an asymptotic run time sense. Think about applications for which one of rasterization and ray casting is a better fit than the other.

These issues are not restricted to our choice of the outer loop. All high-performance renderers subdivide the scene and the image in sophisticated ways. The implementer must choose how to make these subdivisions and for each must again revisit whether to iterate first over pixels (i.e., ray directions) or triangles. The same considerations arise at every level, but they are evaluated differently based on the expected data sizes at that level and the machine architecture.

15.6 Rasterization

We now move on to implement the rasterizing renderer, and compare it to the ray-casting renderer, observing places where each is more efficient and how the restructuring of the code allows for these efficiencies. The relatively tiny change turns out to have substantial impact on computation time, communication demands, and cache coherence.

15.6.1 Swapping the Loops

Listing 15.22 shows an implementation of `rasterize` that corresponds closely to `rayTrace` with the nesting order inverted. The immediate implication of inverting the loop order is that we must store the distance to the closest known intersection at each pixel in a large buffer (`depthBuffer`), rather than in a single float. This is because we no longer process a single pixel to completion before moving to another pixel, so we must store the intermediate processing state. Some implementations store the depth as a distance along the z-axis, or as the inverse of that distance. We choose to store distance along an eye ray to more closely match the ray-caster structure.

The same intermediate state problem arises for the ray `R`. We could create a buffer of rays. In practice, the rays are fairly cheap to recompute and don't justify storage, and we will soon see alternative methods for eliminating the per-pixel ray computation altogether.

Listing 15.22: Rasterizer implemented by simply inverting the nesting order of the loops from the ray tracer, but adding a `DepthBuffer`.

```
void rasterize(Image& image, const Scene& scene, const Camera& camera){

  const int w = image.width(), h = image.height();
  DepthBuffer depthBuffer(w, h, INFINITY);

  // For each triangle
  for (unsigned int t = 0; t < scene.triangleArray.size(); ++t) {
    const Triangle& T = scene.triangleArray[t];

    // Very conservative bounds: the whole screen
    const int x0 = 0;
    const int x1 = w;

    const int y0 = 0;
    const int y1 = h;

    // For each pixel
    for (int y = y0; y < y1; ++y) {
      for (int x = x0; x < x1; ++x) {
        const Ray& R = computeEyeRay(x, y, w, h, camera);

        Radiance3 L_o;
        float distance = depthBuffer.get(x, y);
        if (sampleRayTriangle(scene, x, y, R, T, L_o, distance)) {
          image.set(x, y, L_o);
          depthBuffer.set(x, y, distance);
        }
      }
    }
  }
}
```

The `DepthBuffer` class is similar to `Image`, but it stores a single float at each pixel. Buffers over the image domain are common in computer graphics. This is a good opportunity for code reuse through polymorphism. In C++, the main polymorphic language feature is the template, which corresponds to templates in C# and generics in Java. One could design a templated `Buffer` class and then instantiate it for `Radiance3`, `float`, or whatever per-pixel data was desired. Since methods for saving to disk or gamma correction may not be appropriate for all template parameters, those are best left to subclasses of a specific template instance.

For the initial rasterizer implementation, this level of design is not required. You may simply implement `DepthBuffer` by copying the `Image` class implementation, replacing `Radiance3` with `float`, and deleting the display and save methods. We leave the implementation as an exercise.

Inline Exercise 15.8: Implement `DepthBuffer` as described in the text.

After implementing Listing 15.22, we need to test the rasterizer. At this time, we trust our ray tracer's results. So we run the rasterizer and ray tracer on the same scene, for which they should generate identical pixel values. As before, if the results are not identical, then the differences may give clues about the nature of the bug.

15.6.2 Bounding-Box Optimization

So far, we implemented rasterization by simply inverting the order of the for-each-triangle and for-each-pixel loops in a ray tracer. This performs many ray-triangle intersection tests that will fail. This is referred to as **poor sample test efficiency**.

We can significantly improve sample test efficiency, and therefore performance, on small triangles by only considering pixels whose centers are near the projection of the triangle. To do this we need a heuristic for efficiently bounding each triangle's projection. The bound must be conservative so that we never miss an intersection. The initial implementation already used a very conservative bound. It assumed that every triangle's projection was "near" *every* pixel on the screen. For large triangles, that may be true. For triangles whose true projection is small in screen space, that bound is too conservative.

The best bound would be a triangle's true projection, and many rasterizers in fact use that. However, there are significant amounts of boilerplate and corner cases in iterating over a triangular section of an image, so here we will instead use a more conservative but still reasonable bound: the 2D axis-aligned bounding box about the triangle's projection. For a large nondegenerate triangle, this covers about twice the number of pixels as the triangle itself.

Inline Exercise 15.9: Why is it true that a large-area triangle covers at most about half of the samples of its bounding box? What happens for a *small* triangle, say, with an area smaller than one pixel? What are the implications for sample test efficiency if you know the size of triangles that you expect to render?

The axis-aligned bounding box, however, is straightforward to compute and will produce a significant speedup for many scenes. It is also the method favored by many hardware rasterization designs because the performance is very predictable, and for very small triangles the cost of computing a more accurate bound might dominate the ray-triangle intersection test.

The code in Listing 15.23 determines the bounding box of a triangle `T`. The code projects each vertex from the camera's 3D reference frame onto the plane $z = -1$, and then maps those vertices into the screen space 2D reference frame. This operation is handled entirely by the `perspectiveProject` helper function. The code then computes the minimum and maximum screen-space positions of the vertices and rounds them (by adding 0.5 and then casting the floating-point values to integers) to integer pixel locations to use as the for-each-pixel bounds.

The interesting work is performed by `perspectiveProject`. This inverts the process that `computeEyeRay` performed to find the eye-ray origin (before advancing it to the near plane). A direct implementation following that derivation is given in Listing 15.24. Chapter 13 gives a derivation for this operation as a matrix-vector product followed by a homogeneous division operation. That implementation is more appropriate when the perspective projection follows a series of other transformations that are also expressed as matrices so that the cost of the matrix-vector product can be amortized over all transformations. This version is potentially more computationally efficient (assuming that the constant subexpressions are precomputed) for the case where there are no other transformations; we also give this version to remind you of the derivation of the perspective projection matrix.

Listing 15.23: Projecting vertices and computing the screen-space bounding box.

```
Vector2 low(image.width(), image.height());
Vector2 high(0, 0);

for (int v = 0; v < 3; ++v) {
     const Vector2& X = perspectiveProject(T.vertex(v), image.width
         (), image.height(), camera);
     high = high.max(X);
     low = low.min(X);
}

const int x0 = (int)(low.x + 0.5f);
const int x1 = (int)(high.x + 0.5f);

const int y0 = (int)(low.y + 0.5f);
const int y1 = (int)(high.y + 0.5f);
```

Listing 15.24: Perspective projection.

```
Vector2 perspectiveProject(const Vector3& P, int width, int height,
     const Camera& camera) {
    // Project onto z = -1
    Vector2 Q(-P.x / P.z, -P.y / P.z);

    const float aspect = float(height) / width;

    // Compute the side of a square at z = -1 based on our
    // horizontal left-edge-to-right-edge field of view
    const float s = -2.0f * tan(camera.fieldOfViewX * 0.5f);

    Q.x = width * (-Q.x / s + 0.5f);
    Q.y = height * (Q.y / (s * aspect) + 0.5f);

    return Q;
}
```

Integrate the listings from this section into your rasterizer and run it. The results should exactly match the ray tracer and simpler rasterizer. Furthermore, it should be measurably faster than the simple rasterizer (although both are likely so fast for simple scenes that rendering seems instantaneous).

Simply verifying that the output matches is insufficient testing for this optimization. We're computing bounds, and we could easily have computed bounds that were way too conservative but still happened to cover the triangles for the test scene.

A good follow-up test and debugging tool is to plot the 2D locations to which the 3D vertices projected. To do this, iterate over all triangles again, after the scene has been rasterized. For each triangle, compute the projected vertices as before. But this time, instead of computing the bounding box, directly render the projected vertices by setting the corresponding pixels to white (of course, if there were bright white objects in the scene, another color, such as red, would be a better choice!). Our single-triangle test scene was chosen to be asymmetric. So this test should reveal common errors such as inverting an axis, or a half-pixel shift between the ray intersection and the projection routine.

15.6.3 Clipping to the Near Plane

Note that we can't apply `perspectiveProject` to points for which $z \geq 0$ to generate correct bounds in the invoking rasterizer. A common solution to this problem is to introduce some "near" plane $z = z_n$ for $z_n < 0$ and clip the triangle to it. This is the same as the near plane (`zNear` in the code) that we used earlier to compute the ray origin—since the rays began at the near plane, the ray tracer was also clipping the visible scene to the plane.

Clipping may produce a triangle, a degenerate triangle that is a line or point at the near plane, no intersection, or a quadrilateral. In the latter case we can divide the quadrilateral along one diagonal so that the output of the clipping algorithm is always either empty or one or two (possibly degenerate) triangles.

Clipping is an essential part of many rasterization algorithms. However, it can be tricky to implement well and distracts from our first attempt to simply produce an image by rasterization. While there are rasterization algorithms that never clip [Bli93, OG97], those are much more difficult to implement and optimize. For now, we'll ignore the problem and require that the entire scene is on the opposite side of the near plane from the camera. See Chapter 36 for a discussion of clipping algorithms.

15.6.4 Increasing Efficiency

15.6.4.1 2D Coverage Sampling

Having refactored our renderer so that the inner loop iterates over pixels instead of triangles, we now have the opportunity to substantially amortize much of the work of the ray-triangle intersection computation. Doing so will also build our insight for the relationship between a 3D triangle and its projection, and hint at how it is possible to gain the large constant performance factors that make the difference between offline and interactive rendering.

The first step is to transform the 3D ray-triangle intersection test by projection into a 2D point-in-triangle test. In rasterization literature, this is often referred to as **the visibility problem** or **visibility testing.** If a pixel center does not lie in the projection of a triangle, then the triangle is certainly "invisible" when we look through the center of projection of that pixel. However, the triangle might also be invisible for other reasons, such as a nearer triangle that occludes it, which is not considered here. Another term that has increasing popularity is more accurate: **coverage testing,** as in "Does the triangle *cover* the sample?" Coverage is a necessary but not sufficient condition for visibility.

We perform the coverage test by finding the 2D barycentric coordinates of every pixel center within the bounding box. If the 2D barycentric coordinates at a pixel center show that the pixel center lies within the projected triangle, then the 3D ray through the pixel center will also intersect the 3D triangle [Pin88]. We'll soon see that computing the 2D barycentric coordinates of several adjacent pixels can be done very efficiently compared to computing the corresponding 3D ray-triangle intersections.

15.6.4.2 Perspective-Correct Interpolation

For shading we will require the 3D barycentric coordinates of every ray-triangle intersection that we use, or some equivalent way of interpolating vertex attributes such as surface normals, texture coordinates, and per-vertex colors. We cannot directly use the 2D barycentric coordinates from the coverage test for shading. That is because the 3D barycentric coordinates of a point on the triangle and the 2D barycentric coordinates of the *projection* of that point within the *projection* of the triangle are generally not equal. This can be seen in Figure 15.13. The figure shows a square in 3D with vertices A, B, C, and D, viewed from an oblique perspective so that its 2D projection is a trapezoid. The centroid of the 3D square is point E, which lies at the intersection of the diagonals. Point E is halfway between 3D edges AB and CD, yet in the 2D projection it is clearly much closer to edge CD. In terms of triangles, for triangle ABC, the 3D barycentric coordinates of E must be $w_A = \frac{1}{2}, w_B = 0, w_C = \frac{1}{2}$. The projection of E is clearly not halfway along the 2D line segment between the projections of A and C. (We saw this phenomenon in Chapter 10 as well.)

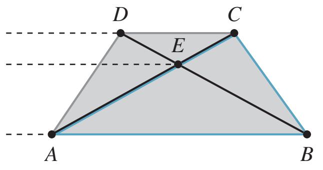


Figure 15.13: E is the centroid of square ABCD in 3D, but its projection is not the centroid of the projection of the square. This can be seen from the fact that the three dashed lines are not evenly spaced in 2D.

Fortunately, there is an efficient analog to 2D linear interpolation for projected 3D linear interpolation. This is interchangeably called **hyperbolic interpolation** [Bli93], **perspective-correct interpolation** [OG97], and **rational linear interpolation** [Hec90].

The perspective-correct interpolation method is simple. We can express it intuitively as, for each scalar vertex attribute u, linearly interpolate both $u' = u/z$ and $1/z$ in screen space. At each pixel, recover the 3D linearly interpolated attribute value from these by $u = u'/(1/z)$. See the following sidebar for a more formal explanation of why this works.

Let $u(x, y, z)$ be some scalar attribute (e.g., albedo, u texture coordinate) that varies linearly over the polygon. Two equivalent definitions may be more intuitive: (a) u is defined at vertices by specific values and varies by barycentric interpolation between them; (b) u has the form of a 3D plane equation, $u(x, y, z) = ax + by + cz + d$.

When the polygon is projected into screen space by the transformation $(x, y, z) \rightarrow (-x/z, -y/z, -1)$ for an image plane at $z = -1$, then **the function $-u(x, y, z)/z$ varies linearly in screen space.** Instead of linear interpolation in screen space, we need to perform a kind of "hyperbolic interpolation" to correctly evaluate u as follows.

Let P and Q be points on the 3D polygon, and let $u(P)$ and $u(Q)$ be some function that varies linearly across the plane of the 3D polygon evaluated at those points. Let $P' = -P/z_P$ be the projection of P and $Q' = -Q/z_Q$ be the projection of Q. At point M on line PQ that projects to $M' = \alpha P' + (1 - \alpha)Q'$, the value of $u(M)$ satisfies

$$\frac{u(M)}{-z_M} = \alpha \frac{u(P)}{-z_P} + (1 - \alpha) \frac{u(Q)}{-z_Q}, \tag{15.5}$$

while $-1/z_M$ satisfies

$$\frac{1}{-z_M} = \alpha \frac{1}{-z_P} + (1 - \alpha) \frac{1}{-z_Q}. \tag{15.6}$$

Solving for $u(M)$ yields

$$u(M) = \frac{\alpha \frac{u(P)}{-z_P} + (1 - \alpha) \frac{u(Q)}{-z_Q}}{\alpha \frac{1}{-z_P} + (1 - \alpha) \frac{1}{-z_Q}}. \tag{15.7}$$

Because for each screen raster (i.e., row of pixels) we hold P and Q constant and vary α linearly, we can simplify the expression above to define a directly parameterized function $u'(\alpha)$:

$$u'(\alpha) = \frac{\alpha \cdot z_Q \cdot u(P) + (1-\alpha)z_P \cdot u(Q)}{\alpha \cdot z_Q + (1-\alpha)z_P}. \tag{15.8}$$

This is often more casually, but memorably, phrased as "In screen space, the perspective-correct interpolation of u is the quotient of the *linear interpolation* of u/z by the linear interpolation of $1/z$."

We can apply the perspective-correct interpolation strategy to any number of per-vertex attributes, including the vertex normals and texture coordinates. That leaves us with input data for our `shade` function, which remains unchanged from its implementation in the ray tracer.

15.6.4.3 2D Barycentric Weights

To implement the perspective-correct interpolation strategy, we need only find an expression for the 2D barycentric weights at the center of each pixel. Consider the barycentric weight corresponding to vertex A of a point Q within a triangle ABC. Recall from Section 7.9 that this weight is the ratio of the distance from Q to the line containing BC to the distance from A to the line containing BC, that is, it is the relative distance across the triangle from the opposite edge. Listing 15.25 gives code for computing a barycentric weight in 2D.

Listing 15.25: Computing one barycentric weight in 2D.

```
/** Returns the distance from Q to the line containing B and A. */
float lineDistance2D(const Point2& A, const Point2& B, const Point2& Q) {
    // Construct the line align:
    const Vector2 n(A.y - B.y, B.x - A.x);
    const float d = A.x * B.y - B.x * A.y;
    return (n.dot(Q) + d) / n.length();
}

/** Returns the barycentric weight corresponding to vertex A of Q in triangle ABC */
float bary2D(const Point2& A, const Point2& B, const Point2& C, const Point2& Q) {
    return lineDistance2D(B, C, Q) / lineDistance2D(B, C, A);
}
```

Inline Exercise 15.10: Under what condition could `lineDistance2D` return `0`, or `n.length()` be `0`, leading to a division by zero? Change your rasterizer to ensure that this condition never occurs. Why does this not affect the final rendering? What situation does this correspond to in a ray caster? How did we resolve that case when ray casting?

The rasterizer structure now requires a few changes from our previous version. It will need the post-projection vertices of the triangle after computing the bounding box in order to perform interpolation. We could either retain them from the bounding-box computation or just compute them again when needed later. We'll recompute the values when needed because it trades a small amount of efficiency for a simpler interface to the bounding function, which makes the code easier to write and debug. Listing 15.26 shows the bounding box-function. The rasterizer must compute versions of the vertex attributes, which in our case are just the vertex normals, that are scaled by the $\frac{1}{z}$ value (which we call w) for the corresponding post-projective vertex. Both of those are per-triangle changes to the code. Finally, the inner loop must compute visibility from the 2D barycentric coordinates instead of from a ray cast. The actual shading computation remains unchanged from the original ray tracer, which is good—we're only looking at strategies for visibility, not shading, so we'd like each to be as modular as possible. Listing 15.27 shows the loop setup of the original rasterizer updated with the bounding-box and 2D barycentric approach. Listing 15.28 shows how the inner loops change.

Listing 15.26: Bounding box for the projection of a triangle, invoked by `rasterize3` *to establish the pixel iteration bounds.*

```
void computeBoundingBox(const Triangle& T, const Camera& camera,
                        const Image& image,
                        Point2 V[3], int& x0, int& y0, int& x1, int& y1) {

    Vector2 high(image.width(), image.height());
    Vector2 low(0, 0);

    for (int v = 0; v < 3; ++v) {
        const Point2& X = perspectiveProject(T.vertex(v), image.width(),
            image.height(), camera);
        V[v] = X;
        high = high.max(X);
        low = low.min(X);
    }

    x0 = (int)floor(low.x);
    x1 = (int)ceil(high.x);

    y0 = (int)floor(low.y);
    y1 = (int)ceil(high.y);
}
```

Listing 15.27: Iteration setup for a barycentric (edge align) rasterizer.

```
/** 2D barycentric evaluation w. perspective-correct attributes */
void rasterize3(Image& image, const Scene& scene,
       const Camera& camera){
  DepthBuffer depthBuffer(image.width(), image.height(), INFINITY);

  // For each triangle
  for (unsigned int t = 0; t < scene.triangleArray.size(); ++t) {
    const Triangle& T = scene.triangleArray[t];

    // Projected vertices
    Vector2 V[3];
    int x0, y0, x1, y1;
    computeBoundingBox(T, camera, image, V, x0, y0, x1, y1);

    // Vertex attributes, divided by -z
    float    vertexW[3];
    Vector3  vertexNw[3];
    Point3   vertexPw[3];
    for (int v = 0; v < 3; ++v) {
      const float w = -1.0f / T.vertex(v).z;
      vertexW[v] = w;
      vertexPw[v] = T.vertex(v) * w;
      vertexNw[v] = T.normal(v) * w;
    }

    // For each pixel
    for (int y = y0; y < y1; ++y) {
      for (int x = x0; x < x1; ++x) {
        // The pixel center
        const Point2 Q(x + 0.5f, y + 0.5f);
        ...

      }
    }
  }
}
```

Listing 15.28: Inner loop of a barycentric (edge align) rasterizer (see Listing 15.27 for the loop setup).

```
// For each pixel
for (int y = y0; y < y1; ++y) {
  for (int x = x0; x < x1; ++x) {
    // The pixel center
    const Point2 Q(x + 0.5f, y + 0.5f);

    // 2D Barycentric weights
    const float weight2D[3] =
      {bary2D(V[0], V[1], V[2], Q),
       bary2D(V[1], V[2], V[0], Q),
       bary2D(V[2], V[0], V[1], Q)};

    if ((weight2D[0]>0) && (weight2D[1]>0) && (weight2D[2]>0)) {
      // Interpolate depth
      float w = 0.0f;
      for (int v = 0; v < 3; ++v) {
        w += weight2D[v] * vertexW[v];
      }

      // Interpolate projective attributes, e.g., P', n'
      Point3 Pw;
      Vector3 nw;
      for (int v = 0; v < 3; ++v) {
        Pw += weight2D[v] * vertexPw[v];
        nw += weight2D[v] * vertexNw[v];
      }

      // Recover interpolated 3D attributes; e.g., P' -> P, n' -> n
      const Point3& P = Pw / w;
      const Vector3& n = nw / w;

      const float depth = P.length();
      // We could also use depth = z-axis distance: depth = -P.z

      // Depth test
      if (depth < depthBuffer.get(x, y)) {
        // Shade
        Radiance3  L_o;
        const Vector3& w_o = -P.direction();

        // Make the surface normal have unit length
        const Vector3& unitN = n.direction();
```

```
            shade(scene, T, P, unitN, w_o, L_o);

            depthBuffer.set(x, y, depth);
            image.set(x, y, L_o);
        }
      }
    }
}
```

To just test coverage, we don't need the magnitude of the barycentric weights. We only need to know that they are all positive. That is, that the current sample is on the positive side of every line bounding the triangle. To perform that test, we could use the distance from a point to a line instead of the full `bary2D` result. For this reason, this approach to rasterization is also referred to as testing the **edge aligns** at each sample. Since we need the barycentric weights for interpolation anyway, it makes sense to normalize the distances where they are computed. Our first instinct is to delay that normalization at least until after we know that the pixel is going to be shaded. However, even for performance, that is unnecessary—if we're going to optimize the inner loop, a much more significant optimization is available to us.

In general, barycentric weights vary linearly along any line through a triangle. The barycentric weight expressions are therefore linear in the loop variables `x` and `y`. You can see this by expanding `bary2D` in terms of the variables inside `lineDistance2D`, both from Listing 15.25. This becomes

$$\begin{aligned}\texttt{bary2D(A, B, C, Vector2(x, y))} &= \frac{(\texttt{n}\cdot\texttt{(x, y)}+\texttt{d})/|\texttt{n}|}{(\texttt{n}\cdot\texttt{C}+\texttt{d})/|\texttt{n}|} \\ &= r\cdot\texttt{x}+s\cdot\texttt{y}+t, \qquad (15.9)\end{aligned}$$

where the constants r, s, and t depend only on the triangle, and so are invariant across the triangle. We are particularly interested in properties invariant over horizontal and vertical lines, since those are our iteration directions.

For instance, `y` is invariant over the innermost loop along a scanline. Because the expressions inside the inner loop are constant in `y` (and all properties of `T`) and linear in `x`, we can compute them incrementally by accumulating derivatives with respect to `x`. That means that we can reduce all the computation inside the innermost loop and before the branch to three additions. Following the same argument for `y`, we can also reduce the computation that moves between rows to three additions. The only unavoidable operations are that for each sample that enters the branch for shading, we must perform three multiplications per scalar attribute; and we must perform a single division to compute $z = -1/w$, which is amortized over all attributes.

15.6.4.4 Precision for Incremental Interpolation

We need to think carefully about precision when incrementally accumulating derivatives rather than explicitly performing linear interpolation by the barycentric coordinates. To ensure that rasterization produces complementary pixel coverage for adjacent triangles with shared vertices ("watertight rasterization"), we must ensure that both triangles accumulate the same barycentric values at the shared edge as they iterate across their different bounding boxes. This means that we need an exact representation of the barycentric derivative. To accomplish this, we must first round vertices to some imposed precision (say, one-quarter of a pixel width), and must then choose a representation and maximum screen size that provide exact storage.

The fundamental operation in the rasterizer is a 2D dot product to determine the side of the line on which a point lies. So we care about the precision of a multiplication and an addition. If our screen resolution is $w \times h$ and we want $k \times k$ subpixel positions for snapping or antialiasing, then we need $\lceil \log_2(k \cdot \max(w, h)) \rceil$ bits to store each scalar value. At 1920×1080 (i.e., effectively 2048×2048) with 4×4 subpixel precision, that's 14 bits. To store the product, we need twice as many bits. In our example, that's 28 bits. This is too large for the 23-bit mantissa portion of the IEEE 754 32-bit floating-point format, which means that we cannot implement the rasterizer using the single-precision `float` data type. We can use a 32-bit integer, representing a 24.4 fixed-point value. In fact, within that integer's space limitations we can increase screen resolution to 8192×8192 at 4×4 subpixel resolution. This is actually a fairly low-resolution subpixel grid, however. In contrast, DirectX 11 mandates eight bits of subpixel precision in each dimension. That is because under low subpixel precision, the aliasing pattern of a diagonal edge moving slowly across the screen appears to jump in discrete steps rather than evolve slowly with motion.

15.6.5 Rasterizing Shadows

Although we are now rasterizing primary visibility, our `shade` routine still determines the locations of shadows by casting rays. Shadowing from a local point source is equivalent to "visibility" from the perspective of that source. So we can apply rasterization to that visibility problem as well.

A **shadow map** [Wil78] is an auxiliary depth buffer rendered from a camera placed at the light's location. This contains the same distance information as obtained by casting rays from the light to selected points in the scene. The shadow map can be rendered in one pass over the scene geometry *before* the camera's view is rendered. Figure 15.14 shows a visualization of a shadow map, which is a common debugging aid.

When a shadowing computation arises during rendering from the camera's view, the renderer uses the shadow map to resolve it. For a rendered point to be unshadowed, it must be simultaneously visible to both the light and the camera. Recall that we are assuming a pinhole camera and a point light source, so the

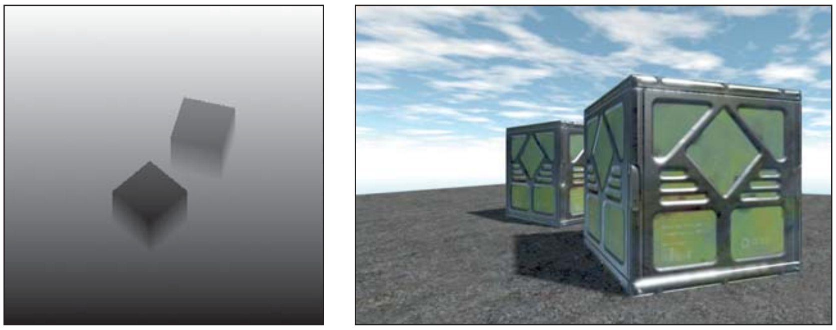


Figure 15.14: Left: A shadow map visualized with black = near the light and white = far from the light. Right: The camera's view of the scene with shadows.

paths from the point to each are defined by line segments of known length and orientation. Projecting the 3D point into the image space of the shadow map gives a 2D point. At that 2D point (or, more precisely, at a nearby one determined by rounding to the sampling grid for the shadow map) we previously stored the distance from the light to the first scene point, that is, the key information about the line segment. If that stored distance is equal to the distance from the 3D point to the 3D light source, then there must not have been any occluding surface and our point is lit. If the distance is less, then the point is in shadow because the light observes some other, shadow-casting, point first along the ray. This depth test must of course be conservative and approximate; we know there will be aliasing from both 2D discretization of the shadow map and its limited precision at each point.

Although we motivated shadow maps in the context of rasterization, they may be generated by or used to compute shadowing with both rasterization and ray casting renderers. There are often reasons to prefer to use the same visibility strategy throughout an application (e.g., the presence of efficient rasterization hardware), but there is no algorithmic constraint that we must do so.

When using a shadow map with triangle rasterization, we can amortize the cost of perspective projection into the shadow map over the triangle by performing most of the computational work at the vertices and then interpolating the results. The result must be interpolated in a perspective-correct fashion, of course. The key is that we want to be perspective-correct with respect to the matrix that maps points in world space to the shadow map, not to the viewport.

Recall the perspective-correct interpolation that we used for positions and texture coordinates (see previous sidebar, which essentially relied on linearly interpolating quantities of the form $\vec{u}/z$ and $w = -1/z$). If we multiply world-space vertices by the matrix that transforms them into 2D shadow map coordinates but do not perform the homogeneous division, then we have a value that varies linearly in the **homogeneous clip space** of the virtual camera at the light that produces the shadow map. In other words, we project each vertex into both the viewing camera's and the light camera's homogeneous clip space. We next perform the homogeneous division for the visible camera only and interpolate the four-component homogeneous vector representing the shadow map coordinate in a perspective-correct fashion in screen space. We next perform the perspective division for the shadow map coordinate at each pixel, paying only for the division and not the matrix product at each pixel. This allows us to transform to the light's projective view volume once per vertex and then interpolate those coordinates using the infrastructure already built for interpolating other elements. The reuse of a general interpolation mechanism and optimization of reducing transformations should naturally suggest that this approach is a good one for a hardware implementation of the graphics pipeline. Chapter 38 discusses how some of these ideas manifest in a particular graphics processor.

15.6.6 Beyond the Bounding Box

A triangle touching $O(n)$ pixels may have a bounding box containing $O(n^2)$ pixels. For triangles with all short edges, especially those with an area of about one pixel, rasterizing by iterating through all pixels in the bounding box is very efficient. Furthermore, the rasterization workload is very predictable for meshes of such triangles, since the number of tests to perform is immediately evident from the box bounds, and rectangular iteration is generally easier than triangular iteration.

For triangles with some large edges, iterating over the bounding box is a poor strategy because $n^2 \gg n$ for large n. In this case, other strategies can be more efficient. We now describe some of these briefly. Although we will not explore these strategies further, they make great projects for learning about hardware-aware algorithms and primary visibility computation.

15.6.6.1 Hierarchical Rasterization

Since the bounding-box rasterizer is efficient for small triangles and is easy to implement, a natural algorithmic approach is to recursively apply the bounding-box algorithm at increasingly fine resolution. This strategy is called **hierarchical rasterization** [Gre96].

Begin by dividing the entire image along a very coarse grid, such as into 16 $\times$ 16 macro-pixels that cover the entire screen. Apply a conservative variation of the bounding-box algorithm to these. Then subdivide the coarse grid and recursively apply the rasterization algorithm within all of the macro cells that overlapped the bounding box.

The algorithm could recur until the macro-pixels were actually a single pixel. However, at some point, we are able to perform a large number of tests either with Single Instruction Multiple Data (SIMD) operations or by using bitmasks packed into integers, so it may not always be a good idea to choose a single pixel as the base case. This is similar to the argument that you shouldn't quicksort all the way down to a length 1 array; for small problem sizes, the constant factors affect the performance more than the asymptotic bound.

For a given precision, one can precompute all the possible ways that a line passes through a tile of samples. These can be stored as bitmasks and indexed by the line's intercept points with the tile [FFR83, SW83]. For each line, using one bit to encode whether the sample is in the positive half-plane of the line allows an 8 $\times$ 8 pattern to fit in a single unsigned 64-bit integer. The bitwise AND of the patterns for the three line aligns defining the triangle gives the coverage mask for all 64 samples. One can use this trick to cull whole tiles efficiently, as well as avoiding per-sample visibility tests. (Kautz et al. [KLA04] extended this to a clever algorithm for rasterizing triangles onto hemispheres, which occurs frequently when sampling indirect illumination.) Furthermore, one can process multiple tiles simultaneously on a parallel processor. This is similar to the way that many GPUs rasterize today.

15.6.6.2 Chunking/Tiling Rasterization

A **chunking rasterizer,** a.k.a. a **tiling rasterizer,** subdivides the image into rectangular tiles, as if performing the first iteration of hierarchical rasterization. Instead of rasterizing a single triangle and performing recursive subdivision of the image, it takes *all* triangles in the scene and bins them according to which tiles they touch. A single triangle may appear in multiple bins.

The tiling rasterizer then uses some other method to rasterize within each tile. One good choice is to make the tiles 8 $\times$ 8 or some other size at which brute-force SIMD rasterization by a lookup table is feasible.

Working with small areas of the screen is a way to combine some of the best aspects of rasterization and ray casting. It maintains both triangle list and buffer memory coherence. It also allows triangle-level sorting so that visibility can be performed analytically instead of using a depth buffer. That allows both more efficient visibility algorithms and the opportunity to handle translucent surfaces in more sophisticated ways.

15.6.6.3 Incremental Scanline Rasterization

For each row of pixels within the bounding box, there is some location that begins the span of pixels covered by the triangle and some location that ends the span. The bounding box contains the triangle vertically and triangles are convex, so there is exactly one span per row (although if the span is small, it may not actually cover the *center* of any pixels).

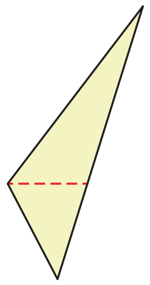


Figure 15.15: Dividing a triangle horizontally at its middle vertex.

A scanline rasterizer divides the triangle into two triangles that meet at a horizontal line through the vertex with the median vertical ordinate of the original triangle (see Figure 15.15). One of these triangles may have zero area, since the original triangle may contain a horizontal edge.

The scanline rasterizer computes the rational slopes of the left and right edges of the top triangle. It then iterates down these in parallel (see Figure 15.16). Since these edges bound the beginning and end of the span across each scanline, no explicit per-pixel sample tests are needed: Every pixel center between the left and right edges at a given scanline is covered by the triangle. The rasterizer then iterates up the bottom triangle from the bottom vertex in the same fashion. Alternatively, it can iterate down the edges of the bottom triangle toward that vertex.

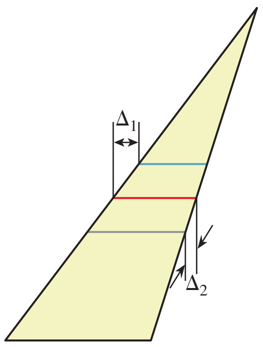


Figure 15.16: Each span's starting point shifts Δ_1 from that of the previous span, and its ending point shifts Δ_2.

The process of iterating along an edge is performed by a variant of either the **Digital Difference Analyzer (DDA)** or Bresenham line algorithm [Bre65], for which there are efficient floating-point and fixed-point implementations.

Pineda [Pin88] discusses several methods for altering the iteration pattern to maximize memory coherence. On current processor architectures this approach is generally eschewed in favor of tiled rasterization because it is hard to schedule for coherent parallel execution and frequently yields poor cache behavior.

15.6.6.4 Micropolygon Rasterization

Hierarchical rasterization recursively subdivided the *image* so that the triangle was always small relative to the number of macro-pixels in the image. An alternative is to maintain constant pixel size and instead subdivide the triangle. For example, each triangle can be divided into four similar triangles (see Figure 15.17). This is the rasterization strategy of the Reyes system [CCC87] used in one of the most popular film renderers, RenderMan. The subdivision process continues until the triangles cover about one pixel each. These triangles are called **micropolygons.** In addition to triangles, the algorithm is often applied to bilinear patches, that is, Bézier surfaces described by four control points (see Chapter 23).

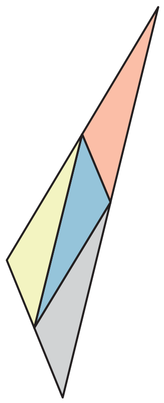


Figure 15.17: A triangle subdivided into four similar triangles.

Subdividing the geometry offers several advantages over subdividing the image. It allows additional geometric processing, such as displacement mapping, to be applied to the vertices after subdivision. This ensures that displacement is performed at (or slightly higher than) image resolution, effectively producing perfect level of detail. Shading can be performed at vertices of the micropolygons and interpolated to pixel centers. This means that the shading is "attached" to object-space locations instead of screen-space locations. This can cause shading features, such as highlights and edges, which move as the surface animates, to move more smoothly and with less aliasing than they do when we use screen-space shading. Finally, effects like motion blur and defocus can be applied by deforming the final shaded geometry before rasterization. This allows computation of shading at a rate proportional to visible geometric complexity but independent of temporal and lens sampling.

15.7 Rendering with a Rasterization API

Rasterization has been encapsulated in APIs. We've seen that although the basic rasterization algorithm is very simple, the process of increasing its performance can rapidly introduce complexity. Very-high-performance rasterizers can be very complex. This complexity leads to a desire to separate out the parts of the rasterizer that we might wish to change between applications while encapsulating the parts that we would like to optimize once, abstract with an API, and then never change again. Of course, it is rare that one truly is willing to never alter an algorithm again, so this means that by building an API for part of the rasterizer we are trading performance and ease of use in some cases for flexibility in others. Hardware rasterizers are an extreme example of an optimized implementation, where flexibility is severely compromised in exchange for very high performance.

There have been several popular rasterization APIs. Today, OpenGL and DirectX are among the most popular hardware APIs for real-time applications. RenderMan is a popular software rasterization API for offline rendering. The space in between, of software rasterizers that run in real time on GPUs, is currently a popular research area with a few open source implementations available [LHLW10, LK11, Pan11].

In contrast to the relative standardization and popularity enjoyed among rasterizer APIs, several ray-casting systems have been built and several APIs have been proposed, although they have yet to reach the current level of standardization and acceptance of the rasterization APIs.

This section describes the OpenGL-DirectX abstraction in general terms. We prefer generalities because the exact entry points for these APIs change on a fairly regular basis. The details of the current versions can be found in their respective manuals. While important for implementation, those details obscure the important ideas.

15.7.1 The Graphics Pipeline

Consider the basic operations of any of our software rasterizer implementations:

1. (Vertex) Per-vertex transformation to screen space
2. (Rasterize) Per-triangle (clipping to the near plane and) iteration over pixels, with perspective-correct interpolation
3. (Pixel) Per-pixel shading
4. (Output Merge) Merging the output of shading with the current color and depth buffers (e.g., alpha blending)

These are the major stages of a rasterization API, and they form a sequence called the **graphics pipeline,** which was introduced in Chapter 1. Throughout the rest of this chapter, we refer to software that invokes API entry points as **host code** and software that is invoked as callbacks by the API as **device code.** In the context of a hardware-accelerated implementation, such as OpenGL on a GPU, this means that the C++ code running on the CPU is host code and the vertex and pixel shaders executing on the GPU are device code.

15.7.1.1 Rasterizing Stage

Most of the complexity that we would like such an API to abstract is in the rasterizing stage. Under current algorithms, rasterization is most efficient when implemented with only a few parameters, so this stage is usually implemented as a **fixed-function** unit. In hardware this may literally mean a specific circuit that can only compute rasterization. In software this may simply denote a module that accepts no parameterization.

15.7.1.2 Vertex and Pixel Stages

The per-vertex and per-pixel operations are ones for which a programmer using the API may need to perform a great deal of customization to produce the desired image. For example, an engineering application may require an orthographic projection of each vertex instead of a perspective one. We've already changed our per-pixel shading code three times, to support Lambertian, Blinn-Phong, and Blinn-Phong plus shadowing, so clearly customization of that stage is important. The performance impact of allowing nearly unlimited customization of vertex and pixel operations is relatively small compared to the benefits of that customization and the cost of rasterization and output merging. Most APIs enable customization of vertex and pixel stages by accepting callback functions that are executed for each vertex and each pixel. In this case, the stages are called **programmable** units.

A pipeline implementation with programmable units is sometimes called a **programmable pipeline.** Beware that in this context, the pipeline *order* is in fact fixed, and only the *units* within it are programmable. Truly programmable pipelines in which the order of stages can be altered have been proposed [SFB$^+$09] but are not currently in common use.

For historical reasons, the callback functions are often called **shaders** or **programs.** Thus, a **pixel shader** or "pixel program" is a callback function that will be executed at the per-pixel stage. For triangle rasterization, the pixel stage is often referred to as the **fragment** stage. A fragment is the portion of a triangle that overlaps the bounds of a pixel. It is a matter of viewpoint whether one is computing the shade of the fragment and sampling that shade at the pixel, or directly computing the shade at the pixel. The distinction only becomes important when computing visibility independently from shading. **Multi-sample anti-aliasing** (**MSAA**) is an example of this. Under that rasterization strategy, many visibility samples (with corresponding depth buffer and radiance samples) are computed within each pixel, but a single shade is applied to all the samples that pass the depth and visibility test. In this case, one truly is shading a fragment and not a pixel.

15.7.1.3 Output Merging Stage

The output merging stage is one that we might like to customize as consumers of the API. For example, one might imagine simulating translucent surfaces by blending the current and previous radiance values in the frame buffer. However, the output merger is also a stage that requires synchronization between potentially parallel instances of the pixel shading units, since it writes to a shared frame buffer. As a result, most APIs provide only limited customization at the output merge stage. That allows lockless access to the underlying data structures, since the implementation may explicitly schedule pixel shading to avoid contention at the frame buffer. The limited customization options typically allow the programmer to choose the operator for the depth comparison. They also typically allow a choice of compositing operator for color limited to linear blending, minimum, and maximum operations on the color values.

There are of course more operations for which one might wish to provide an abstracted interface. These include per-object and per-mesh transformations, tessellation of curved patches into triangles, and per-triangle operations like silhouette detection or surface extrusion. Various APIs offer abstractions of these within a programming model similar to vertex and pixel shaders.

Chapter 38 discusses how GPUs are designed to execute this pipeline efficiently. Also refer to your API manual for a discussion of the additional stages (e.g., tessellate, geometry) that may be available.

15.7.2 Interface

The interface to a software rasterization API can be very simple. Because a software rasterizer uses the same memory space and execution model as the host program, one can pass the scene as a pointer and the callbacks as function pointers or classes with virtual methods. Rather than individual triangles, it is convenient to pass whole meshes to a software rasterizer to decrease the per-triangle overhead.

For a hardware rasterization API, the host machine (i.e., CPU) and graphics device (i.e., GPU) may have separate memory spaces and execution models. In this case, shared memory and function pointers no longer suffice. Hardware rasterization APIs therefore must impose an explicit memory boundary and narrow entry points for negotiating it. (This is also true of the fallback and reference software implementations of those APIs, such as Mesa and DXRefRast.) Such an API requires the following entry points, which are detailed in subsequent subsections.

1. Allocate device memory.
2. Copy data between host and device memory.
3. Free device memory.
4. Load (and compile) a shading program from source.
5. Configure the output merger and other fixed-function state.
6. Bind a shading program and set its arguments.
7. Launch a **draw call,** a set of device threads to render a triangle list.

15.7.2.1 Memory Principles

The memory management routines are conceptually straightforward. They correspond to `malloc`, `memcpy`, and `free`, and they are typically applied to large arrays, such as an array of vertex data. They are complicated by the details necessary to achieve high performance for the case where data must be transferred per rendered frame, rather than once per scene. This occurs when streaming geometry for a scene that is too large for the device memory; for example, in a world large enough that the viewer can only ever observe a small fraction at a time. It also occurs when a data stream from another device, such as a camera, is an input to the rendering algorithm. Furthermore, hybrid software-hardware rendering and physics algorithms perform some processing on each of the host and device and must communicate each frame.

One complicating factor for memory transfer is that it is often desirable to adjust the data layout and precision of arrays during the transfer. The data structure for 2D buffers such as images and depth buffers on the host often resembles the "linear," row-major ordering that we have used in this chapter. On a graphics processor, 2D buffers are often wrapped along Hilbert or Z-shaped (Morton) curves, or at least grouped into small blocks that are themselves row-major (i.e., “block-linear”), to avoid the cache penalty of vertical iteration. The origin of a buffer may differ, and often additional padding is required to ensure that rows have specific memory alignments for wide vector operations and reduced pointer size.

Another complicating factor for memory transfer is that one would often like to overlap computation with memory operations to avoid stalling either the host or device. Asynchronous transfers are typically accomplished by semantically mapping device memory into the host address space. Regular host memory operations can then be performed as if both shared a memory space. In this case the programmer must manually synchronize both host and device programs to ensure that data is never read by one while being written by the other. Mapped memory is typically uncached and often has alignment considerations, so the programmer must furthermore be careful to control access patterns.

Note that memory transfers are intended for large data. For small values, such as scalars, 4×4 matrices, and even short arrays, it would be burdensome to explicitly allocate, copy, and free the values. For a shading program with twenty or so arguments, that would incur both runtime and software management overhead. So small values are often passed through a different API associated with shaders.

15.7.2.2 Memory Practice

Listing 15.30 shows part of an implementation of a triangle mesh class. Making rendering calls to transfer individual triangles from the host to the graphics device would be inefficient. So, the API forces us to load a large array of the geometry to the device once when the scene is created, and to encode that geometry as efficiently as possible.

Few programmers write directly to hardware graphics APIs. Those APIs reflect the fact that they are designed by committees and negotiated among vendors. They provide the necessary functionality but do so through awkward interfaces that obscure the underlying function of the calling code. Usage is error-prone because the code operates directly on pointers and uses manually managed memory.

For example, in OpenGL, the code to allocate a device array and bind it to a shader input looks something like Listing 15.29. Most programmers abstract these direct host calls into a vendor-independent, easier-to-use interface.

Listing 15.29: Host code for transferring an array of vertices to the device and binding it to a shader input.

```
// Allocate memory:
GLuint vbo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, hostVertex.size() * 2 * sizeof(Vector3), NULL,GL_STATIC_DRAW);
GLvoid* deviceVertex = 0;
GLvoid* deviceNormal = hostVertex.size() * sizeof(Vector3);

// Copy memory:
glBufferSubData(GL_ARRAY_BUFFER, deviceVertex, hostVertex.size() *
      sizeof(Point3), &hostVertex[0]);

// Bind the array to a shader input:
int vertexIndex = glGetAttribLocation(shader, "vertex");
glEnableVertexAttribArray(vertexIndex);
glVertexAttribPointer(vertexIndex, 3, GL_FLOAT, GL_FALSE, 0, deviceVertex);
```

Most programmers wrap the underlying hardware API with their own layer that is easier to use and provides type safety and memory management. This also has the advantage of abstracting the renderer from the specific hardware API. Most console, OS, and mobile device vendors intentionally use equivalent but incompatible hardware rendering APIs. Abstracting the specific hardware API into a generic one makes it easier for a single code base to support multiple platforms, albeit at the cost of one additional level of function invocation.

For Listing 15.30, we wrote to one such platform abstraction instead of directly to a hardware API. In this code, the `VertexBuffer` class is a managed memory array in device RAM and `AttributeArray` and `IndexArray` are subsets of a `VertexBuffer`. The "vertex" in the name means that these classes store per-vertex data. It does not mean that they store only vertex positions—for example, the `m_normal` array is stored in an `AttributeArray`. This naming convention is a bit confusing, but it is inherited from OpenGL and DirectX. You can either translate this code to the hardware API of your choice, implement the `VertexBuffer` and `AttributeArray` classes yourself, or use a higher-level API such as G3D that provides these abstractions.

Listing 15.30: Host code for an indexed triangle mesh (equivalent to a set of `Triangle` *instances that share a* `BSDF`*).*

```
class Mesh {
private:
    AttributeArray    m_vertex;
    AttributeArray    m_normal;
    IndexStream       m_index;

    shared_ptr<BSDF>  m_bsdf;

public:

    Mesh() {}

    Mesh(const std::vector<Point3>& vertex,
        const std::vector<Vector3>& normal,
        const std::vector<int>& index, const shared_ptr<BSDF>& bsdf) : m_bsdf(bsdf) {

        shared_ptr<VertexBuffer> dataBuffer =
          VertexBuffer::create((vertex.size() + normal.size()) *
            sizeof(Vector3) + sizeof(int) * index.size());
        m_vertex = AttributeArray(&vertex[0], vertex.size(), dataBuffer);
        m_normal = AttributeArray(&normal[0], normal.size(), dataBuffer);

        m_index = IndexStream(&index[0], index.size(), dataBuffer);
    }

    ...
};

/** The rendering API pushes us towards a mesh representation
    because it would be inefficient to make per-triangle calls. */
class MeshScene {
public:
    std::vector<Light>    lightArray;
    std::vector<Mesh>     meshArray;
};
```

Listing 15.31 shows how this code is used to model the triangle-and-ground-plane scene. In it, the process of uploading the geometry to the graphics device is entirely abstracted within the `Mesh` class.

Listing 15.31: Host code to create indexed triangle meshes for the triangle-plus-ground scene.

```
void makeTrianglePlusGroundScene(MeshScene& s) {
    std::vector<Vector3> vertex, normal;
    std::vector<int> index;

    // Green triangle geometry
    vertex.push_back(Point3(0,1,-2)); vertex.push_back(Point3(-1.9f,-1,-2));
        vertex.push_back(Point3(1.6f,-0.5f,-2));
    normal.push_back(Vector3(0,0.6f,1).direction()); normal.
        push_back(Vector3(-0.4f,-0.4f, 1.0f).direction()); normal.
        push_back(Vector3(0.4f,-0.4f, 1.0f).direction());
    index.push_back(0); index.push_back(1); index.push_back(2);
    index.push_back(0); index.push_back(2); index.push_back(1);
    shared_ptr<BSDF> greenBSDF(new PhongBSDF(Color3::green() * 0.8f,
                                             Color3::white() * 0.2f, 100));

    s.meshArray.push_back(Mesh(vertex, normal, index, greenBSDF));
    vertex.clear(); normal.clear(); index.clear();

    /////////////////////////////////////////////////////////
    // Ground plane geometry
    const float groundY = -1.0f;
    vertex.push_back(Point3(-10, groundY, -10)); vertex.push_back(Point3(-10,
        groundY, -0.01f));
    vertex.push_back(Point3(10, groundY, -0.01f)); vertex.push_back(Point3(10,
        groundY, -10));

    normal.push_back(Vector3::unitY()); normal.push_back(Vector3::unitY());
    normal.push_back(Vector3::unitY()); normal.push_back(Vector3::unitY());

    index.push_back(0); index.push_back(1); index.push_back(2);
    index.push_back(0); index.push_back(2); index.push_back(3);

    const Color3 groundColor = Color3::white() * 0.8f;
    s.meshArray.push_back(Mesh(vertex, normal, index, groundColor));

    //////////////////////////////////////////////////////////
    // Light source
    s.lightArray.resize(1);
    s.lightArray[0].position = Vector3(1, 3, 1);
    s.lightArray[0].power = Color3::white() * 31.0f;
}
```

15.7.2.3 Creating Shaders

The vertex shader must transform the input vertex in global coordinates to a homogeneous point on the image plane. Listing 15.32 implements this transformation. We chose to use the OpenGL Shading Language (GLSL). GLSL is representative of other contemporary shading languages like HLSL, Cg, and RenderMan. All of these are similar to C++. However, there are some minor syntactic differences between GLSL and C++ that we call out here to aid your reading of this example. In GLSL,

- Arguments that are constant over all triangles are passed as global ("uniform") variables.
- Points, vectors, and colors are all stored in `vec3` type.
- `const` has different semantics (compile-time constant).
- `in`, `out`, and `inout` are used in place of C++ reference syntax.
- `length`, `dot`, etc. are functions instead of methods on vector classes.

Listing 15.32: Vertex shader for projecting vertices. The output is in homogeneous space before the division operation. This corresponds to the `perspectiveProject` *function from Listing 15.24.*

```
#version 130

// Triangle vertices
in vec3 vertex;
in vec3 normal;

// Camera and screen parameters
uniform float fieldOfViewX;
uniform float zNear;
uniform float zFar;
uniform float width;
uniform float height;

// Position to be interpolated
out vec3 Pinterp;

// Normal to be interpolated
out vec3 ninterp;

vec4 perspectiveProject(in vec3 P) {
    // Compute the side of a square at z = -1 based on our
    // horizontal left-edge-to-right-edge field of view .
    float s = -2.0f * tan(fieldOfViewX * 0.5f);
    float aspect = height / width;

    // Project onto z = -1
    vec4 Q;
    Q.x = 2.0 * -Q.x / s;
    Q.y = 2.0 * -Q.y / (s * aspect);
    Q.z = 1.0;
    Q.w = -P.z;

    return Q;
}

void main() {
    Pinterp = vertex;
    ninterp = normal;

    gl_Position = perspectiveProject(Pinterp);
}
```

None of these affect the expressiveness or performance of the basic language. The specifics of shading-language syntax change frequently as new versions are released, so don't focus too much on the details. The point of this example is how the overall form of our original program is preserved but adjusted to the conventions of the hardware API.

Under the OpenGL API, the outputs of a vertex shader are a set of attributes and a vertex of the form $(x, y, a, -z)$. That is, a homogeneous point for which the perspective division has not yet been performed. The value $a/-z$ will be used for the depth test. We choose $a = 1$ so that the depth test is performed on $-1/z$, which is a positive value for the negative z locations that will be visible to the camera. We previously saw that any function that provides a consistent depth ordering can be used for the depth test. We mentioned that distance along the eye ray, $-z$, and $-1/z$ are common choices. Typically one scales the a value such that $-a/z$ is in the range $[0, 1]$ or $[-1, 1]$, but for simplicity we'll omit that here. See Chapter 13 for the derivation of that transformation.

Note that we did not scale the output vertex to the dimensions of the image, negate the y-axis, or translate the origin to the upper left in screen space, as we did for the software renderer. That is because by convention, OpenGL considers the upper-left corner of the screen to be at $(-1, 1)$ and the lower-right corner at $(1, -1)$.

We choose the 3D position of the vertex and its normal as our attributes. The hardware rasterizer will automatically interpolate these across the surface of the triangle in a perspective-correct manner. We need to treat the vertex as an attribute because OpenGL does not expose the 3D coordinates of the point being shaded.

Listings 15.33 and 15.34 give the pixel shader code for the `shade` routine, which corresponds to the shade function from Listing 15.17, and helper functions that correspond to the `visible` and `BSDF::evaluateFiniteScatteringDensity` routines from the ray tracer and software rasterizer. The output of the shader is in homogeneous space before the division operation. This corresponds to the `perspectiveProject` function from Listing 15.24. The interpolated attributes enter the shader as global variables `Pinterp` and `ninterp`. We then perform shading in exactly the same manner as for the software renderers.

Listing 15.33: Pixel shader for computing the radiance scattered toward the camera from one triangle illuminated by one light.

```
#version 130
// BSDF
uniform vec3    lambertian;
uniform vec3    glossy;
uniform float   glossySharpness;

// Light
uniform vec3    lightPosition;
uniform vec3    lightPower;

// Pre-rendered depth map from the light's position
uniform sampler2DShadow shadowMap;

// Point being shaded. OpenGL has automatically performed
// homogeneous division and perspective-correct interpolation for us.
in vec3             Pinterp;
in vec3             ninterp;

// Value we are computing
out vec3            radiance;

// Normalize the interpolated normal; OpenGL does not automatically
// renormalize for us.
vec3 n = normalize(ninterp);

```

```
vec3 shade(const in vec3 P, const in vec3 n) {
vec3 radiance              = vec3(0.0);

  // Assume only one light
  vec3 offset           = lightPosition - P;
  float distanceToLight = length(offset);
  vec3 w_i              = offset / distanceToLight;
  vec3 w_o              = -normalize(P);

  if (visible(P, w_i, distanceToLight, shadowMap)) {
      vec3 L_i = lightPower / (4 * PI * distanceToLight * distanceToLight);

      // Scatter the light.
      radiance +=
          L_i *
          evaluateFiniteScatteringDensity(w_i, w_o) *
          max(0.0, dot(w_i, n));
  }

  return radiance;
}

void main() {
    vec3 P = Pinterp;


    radiance = shade(P, n);
}
```

Listing 15.34: Helper functions for the pixel shader.

```
#define PI 3.1415927

bool visible(const in vec3 P, const in vec3 w_i, const in float distanceToLight,
    sampler2DShadow shadowMap) {
    return true;
}

/** Returns f(wi, wo). Same as BSDF::evaluateFiniteScatteringDensity
    from the ray tracer. */
vec3 evaluateFiniteScatteringDensity(const in vec3 w_i, const in vec3 w_o) {
    vec3 w_h = normalize(w_i + w_o);

    return (k_L +
            k_G * ((s + 8.0) * pow(max(0.0, dot(w_h, n)), s) / 8.0)) / PI;
}
```

However, there is one exception. The software renderers iterated over all the lights in the scene for each point to be shaded. The pixel shader is hardcoded to accept a single light source. That is because processing a variable number of arguments is challenging at the hardware level. For performance, the inputs to shaders are typically passed through registers, not heap memory. Register allocation is generally a major factor in optimization. Therefore, most shading compilers require the number of registers consumed to be known at compile time, which precludes passing variable length arrays. Programmers have developed three **forward-rendering** design patterns for working within this limitation. These use a single framebuffer and thus limit the total space required by the algorithm. A fourth and currently popular **deferred-rendering** method requires additional space.

1. Multipass Rendering: Make one **pass** per light over all geometry, summing the individual results. This works because light combines by superposition. However, one has to be careful to resolve visibility correctly on the first pass and then never alter the depth buffer. This is the simplest and most elegant solution. It is also the slowest because the overhead of launching a pixel shader may be significant, so launching it multiple times to shade the same point is inefficient.

2. Übershader: Bound the total number of lights, write a shader for that maximum number, and set the unused lights to have zero power. This is one of the most common solutions. If the overhead of launching the pixel shader is high and there is significant work involved in reading the BSDF parameters, the added cost of including a few unused lights may be low. This is a fairly straightforward modification to the base shader and is a good compromise between performance and code clarity.

3. Code Generation: Generate a set of shading programs, one for each number of lights. These are typically produced by writing another program that automatically generates the shader code. Load *all* of these shaders at runtime and bind whichever one matches the number of lights affecting a particular object. This achieves high performance if the shader only needs to be swapped a few times per frame, and is potentially the fastest method. However, it requires significant infrastructure for managing both the source code and the compiled versions of all the shaders, and may actually be slower than the conservative solution if changing shaders is an expensive operation.

If there are different BSDF terms for different surfaces, then we have to deal with all the permutations of the number of lights and the BSDF variations. We again choose between the above three options. This combinatorial explosion is one of the primary drawbacks of current shading languages, and it arises directly from the requirement that the shading compiler produce efficient code. It is not hard to design more flexible languages and to write compilers for them. But our motivation for moving to a hardware API was largely to achieve increased performance, so we are unlikely to accept a more general shading language if it significantly degrades performance.

4. Deferred Lighting: A deferred approach that addresses these problems but requires more memory is to separate the computation of *which* point will color each pixel from illumination computation. An initial rendering pass renders many parallel buffers that encode the shading coefficients, surface normal, and location of each point (often, assuming an übershader). Subsequent passes then iterate over the screen-space area conservatively affected by each light, computing and summing illumination. Two common structures for those lighting passes are multiple lights applied to large screen-space tiles and ellipsoids for individual lights that cover the volume within which their contribution is non-negligible.

For the single-light case, moving from our own software rasterizer to a hardware API did not change our `perspectiveProject` and `shade` functions substantially.

However, our shade function was not particularly powerful. Although we did not choose to do so, in our software rasterizer, we could have executed arbitrary code inside the shade function. For example, we could have written to locations other than the current pixel in the frame buffer, or cast rays for shadows or reflections. Such operations are typically disallowed in a hardware API. That is because they interfere with the implementation's ability to efficiently schedule parallel instances of the shading programs in the absence of explicit (inefficient) memory locks.

This leaves us with two choices when designing an algorithm with more significant processing, especially at the pixel level. The first choice is to build a hybrid renderer that performs some of the processing on a more general processor, such as the host, or perhaps on a general computation API (e.g., CUDA, Direct Compute, OpenCL, OpenGL Compute). Hybrid renderers typically incur the cost of additional memory operations and the associated synchronization complexity.

The second choice is to frame the algorithm purely in terms of rasterization operations, and make multiple rasterization passes. For example, we can't conveniently cast shadow rays in most hardware rendering APIs today. But we can sample from a previously rendered shadow map.

Similar methods exist for implementing reflection, refraction, and indirect illumination purely in terms of rasterization. These avoid much of the performance overhead of hybrid rendering and leverage the high performance of hardware rasterization. However, they may not be the most natural way of expressing an algorithm, and that may lead to a net inefficiency and certainly to additional software complexity. Recall that changing the order of iteration from ray casting to rasterization increased the space demands of rendering by requiring a depth buffer to store intermediate results. In general, converting an arbitrary algorithm to a rasterization-based one often has this effect. The space demands might grow larger than is practical in cases where those intermediate results are themselves large.

Shading languages are almost always compiled into executable code at runtime, inside the API. That is because even within products from one vendor the underlying micro-architecture may vary significantly. This creates a tension within the compiler between optimizing the target code and producing the executable quickly. Most implementations err on the side of optimization, since shaders are often loaded once per scene. Beware that if you synthesize or stream shaders throughout the rendering process there may be substantial overhead.

Some languages (e.g., HLSL and CUDA) offer an initial compilation step to an intermediate representation. This eliminates the runtime cost of parsing and some trivial compilation operations while maintaining flexibility to optimize for a specific device. It also allows software developers to distribute their graphics applications without revealing the shading programs to the end-user in a human-readable form on the file system. For closed systems with fixed specifications, such as game consoles, it is possible to compile shading programs down to true machine code. That is because on those systems the exact runtime device is known at host-program compile time. However, doing so would reveal some details of the proprietary micro-architecture, so even in this case vendors do not always choose to have their APIs perform a complete compilation step.

15.7.2.4 Executing Draw Calls

To invoke the shaders we issue `draw` calls. These occur on the host side. One typically clears the framebuffer, and then, for each mesh, performs the following operations.

1. Set fixed function state.
2. Bind a shader.
3. Set shader arguments.
4. Issue the draw call.

These are followed by a call to send the framebuffer to the display, which is often called a **buffer swap.** An abstracted implementation of this process might look like Listing 15.35. This is called from a main rendering loop, such as Listing 15.36.

Listing 15.35: Host code to set fixed-function state and shader arguments, and to launch a draw call under an abstracted hardware API.

```
void loopBody(RenderDevice* gpu) {
    gpu->setColorClearValue(Color3::cyan() * 0.1f);
    gpu->clear();

    const Light& light = scene.lightArray[0];

    for (unsigned int m = 0; m < scene.meshArray.size(); ++m) {
        Args args;
        const Mesh& mesh = scene.meshArray[m];
        const shared_ptr<BSDF>& bsdf = mesh.bsdf();

        args.setUniform("fieldOfViewX",      camera.fieldOfViewX);
        args.setUniform("zNear",             camera.zNear);
        args.setUniform("zFar",              camera.zFar);

        args.setUniform("lambertian",        bsdf->lambertian);
        args.setUniform("glossy",            bsdf->glossy);
        args.setUniform("glossySharpness",   bsdf->glossySharpness);

        args.setUniform("lightPosition",     light.position);
        args.setUniform("lightPower",        light.power);

        args.setUniform("shadowMap",         shadowMap);

        args.setUniform("width",             gpu->width());
        args.setUniform("height",            gpu->height());

        gpu->setShader(shader);

        mesh.sendGeometry(gpu, args);
    }
    gpu->swapBuffers();
}
```

Listing 15.36: Host code to set up the main hardware rendering loop.

```
OSWindow::Settings osWindowSettings;
RenderDevice* gpu = new RenderDevice();
gpu->init(osWindowSettings);

// Load the vertex and pixel programs
shader = Shader::fromFiles("project.vrt", "shade.pix");

shadowMap = Texture::createEmpty("Shadow map", 1024, 1024,
    ImageFormat::DEPTH24(), Texture::DIM_2D_NPOT, Texture::Settings::shadow());
makeTrianglePlusGroundScene(scene);

```

```
// The depth test will run directly on the interpolated value in
// Q.z/Q.w, which is going to be smallest at the far plane
gpu->setDepthTest(RenderDevice::DEPTH_GREATER);
gpu->setDepthClearValue(0.0);

while (! done) {
    loopBody(gpu);
    processUserInput();
}

...
```

15.8 Performance and Optimization

We'll now consider several examples of optimization in hardware-based rendering. This is by no means an exhaustive list, but rather a set of model techniques from which you can draw ideas to generate your own optimizations when you need them.

15.8.1 Abstraction Considerations

Many performance optimizations will come at the price of significantly complicating the implementation. Weigh the performance advantage of an optimization against the additional cost of debugging and code maintenance. High-level algorithmic optimizations may require significant thought and restructuring of code, but they tend to yield the best tradeoff of performance for code complexity. For example, simply dividing the screen in half and asynchronously rendering each side on a separate processor nearly doubles performance at the cost of perhaps 50 additional lines of code that do not interact with the inner loop of the renderer.

In contrast, consider some low-level optimizations that we intentionally passed over. These include reducing common subexpressions (e.g., mapping all of those repeated divisions to multiplications by an inverse that is computed once) and lifting constants outside loops. Performing those destroys the clarity of the algorithm, but will probably gain only a 50% throughput improvement.

This is not to say that low-level optimizations are not worthwhile. But they are primarily worthwhile when you have completed your high-level optimizations; at that point you are more willing to complicate your code and its maintenance because you are done adding features.

15.8.2 Architectural Considerations

The primary difference between the simple rasterizer and ray caster described in this chapter is that the "for each pixel" and "for each triangle" loops have the opposite nesting. This is a trivial change and the body of the inner loop is largely similar in each case. But the trivial change has profound implications for memory access patterns and how we can algorithmically optimize each.

Scene triangles are typically stored in the heap. They may be in a flat 1D array, or arranged in a more sophisticated data structure. If they are in a simple data structure such as an array, then we can ensure reasonable memory coherence by iterating through them in the same order that they appear in memory. That produces efficient cache behavior. However, that iteration also requires substantial bandwidth because the entire scene will be processed for each pixel. If we use a more sophisticated data structure, then we likely will reduce bandwidth but also reduce memory coherence. Furthermore, adjacent pixels likely sample the same triangle, but by the time we have iterated through to testing that triangle again it is likely to have been flushed from the cache. A popular low-level optimization for a ray tracer is to trace a bundle of rays called a **ray packet** through adjacent pixels. These rays likely traverse the scene data structure in a similar way, which increases memory coherence. On a SIMD processor a single thread can trace an entire packet simultaneously. However, packet tracing suffers from computational coherence problems. Sometimes different rays in the same packet progress to different parts of the scene data structure or branch different ways in the ray intersection test. In these cases, processing multiple rays simultaneously on a thread gives no advantage because memory coherence is lost or both sides of the branch must be taken. As a result, fast ray tracers are often designed to trace packets through very sophisticated data structures. They are typically limited not by computation but by memory performance problems arising from resultant cache inefficiency.

Because frame buffer storage per pixel is often much smaller than scene structure per triangle, the rasterizer has an inherent memory performance advantage over the ray tracer. A rasterizer reads each triangle into memory and then processes it to completion, iterating over many pixels. Those pixels must be adjacent to each other in space. For a row-major image, if we iterate along rows, then the pixels covered by the triangle are also adjacent in memory and we will have excellent coherence and fairly low memory bandwidth in the inner loop. Furthermore, we can process multiple adjacent pixels, either horizontally or vertically, simultaneously on a SIMD architecture. These will be highly memory and branch coherent because we're stepping along a single triangle. There are many variations on ray casting and rasterization that improve their asymptotic behavior. However, these algorithms have historically been applied to only millions of triangles and pixels. At those sizes, constant factors like coherence still drive the performance of the algorithms, and rasterization's superior coherence properties have made it preferred for high-performance rendering. The cost of this coherence is that after even the few optimizations needed to get real-time performance from a rasterizer, the code becomes so littered with bit-manipulation tricks and highly derived terms that the elegance of a simple ray cast seems very attractive from a software engineering perspective. This difference is only magnified when we make the rendering algorithm more sophisticated. The conventional wisdom is that ray-tracing algorithms are elegant and easy to extend but are hard to optimize, and rasterization algorithms are very efficient but are awkward and hard to augment with new features. Of course, one can always make a ray tracer fast and ugly (which packet tracing succeeds at admirably) and a rasterizer extensible but slow (e.g., Pixar's RenderMan, which was used extensively in film rendering over the past two decades).

15.8.3 Early-Depth-Test Example

One simple optimization that can significantly improve performance, yet only minimally affects clarity, is an early depth test. Both the rasterizer and the ray-tracer structures sometimes shaded a point, only to later find that some other point was closer to the surface. As an optimization, we might first find the closest point before doing any shading, then go back and shade the point that was closest. In ray tracing, each pixel is processed to completion before moving to the next, so this involves running the entire visibility loop for one pixel, maintaining the shading inputs for the closest-known intersection at each iteration, and then shading after that loop terminates. In rasterization, pixels are processed many times, so we have to make a complete first pass to determine visibility and then a second pass to do shading. This is called an **early-depth pass** [HW96] if it primes `depthBuffer` so that only the surface that shades will pass the inner test. The process is called **deferred shading** if it also accumulates the shading parameters so that they do not need to be recomputed. This style of rendering was first introduced by Whitted and Weimer [WW82] to compute shading independent from visibility at a time when primary visibility computation was considered expensive. Within a decade it was considered a method to accelerate complex rendering toward real-time rendering (and the "deferred" term was coined) [MEP92], and today its use is widespread as a further optimization on hardware platforms that already achieve real time for complex scenes.

For a scene that has high **depth complexity** (i.e., in which many triangles project to the same point in the image) and an expensive shading routine, the performance benefit of an early depth test is significant. The cost of rendering a pixel without an early depth test is $\mathrm{O}(tv+ts)$, where t is the number of triangles, v is the time for a visibility test, and s is the time for shading. This is an upper bound. When we are lucky and always encounter the closest triangle first, the performance matches the lower bound of $\Omega(tv+s)$ since we only shade once. The early-depth optimization ensures that we are always in this lower-bound case. We have seen how rasterization can drive the cost of v very low—it can be reduced to a few additions per pixel—at which point the challenge becomes reducing the number of triangles tested at each pixel. Unfortunately, that is not as simple. Strategies exist for obtaining expected $\mathrm{O}(v \log t + s)$ rendering times for scenes with certain properties, but they significantly increase code complexity.

15.8.4 When Early Optimization Is Good

The domain of graphics raises two time-based exceptions to the general rule of thumb to avoid premature optimization. The more significant of these exceptions is that when low-level optimizations can accelerate a rendering algorithm just enough to make it run at interactive rates, it might be worth making those optimizations early in the development process. It is much easier to debug an interactive rendering system than an offline one. Interaction allows you to quickly experiment with new viewpoints and scene variations, effectively giving you a true 3D perception of your data instead of a 2D slice. If that lets you debug faster, then the optimization has increased your ability to work with the code despite the added complexity. The other exception applies when the render time is just at the threshold of your patience. Most programmers are willing to wait for 30 seconds for an image to render, but they will likely leave the computer or switch tasks if the render time is, say, more than two minutes. Every time you switch tasks or leave the computer you're amplifying the time cost of debugging, because on your return you have to recall what you were doing before you left and get back into the development flow. If you can reduce the render time to something you are willing to wait for, then you have cut your debugging time and made the process sufficiently more pleasant that your productivity will again rise despite increased code complexity. We enshrine these ideas in a principle:

✓ **THE EARLY OPTIMIZATION PRINCIPLE:** It's worth optimizing early if it makes the difference between an interactive program and one that takes several minutes to execute. Shortening the debugging cycle and supporting interactive testing are worth the extra effort.

15.8.5 Improving the Asymptotic Bound

To scale to truly large scenes, no linear-time rendering algorithm suffices. We must somehow eliminate whole parts of the scene without actually touching their data even once. Data structures for this are a classic area of computer graphics that continues to be a hot research topic. The basic idea behind most of these is the same as behind using tree and bucket data structures for search and sort problems. Visibility testing is primarily a search operation, where we are searching for the closest ray intersection with the scene. If we precompute a treelike data structure that orders the scene primitives in some way that allows conservatively culling a constant fraction of the primitives at each layer, we will approach $O(\log n)$-time visibility testing for the entire scene, instead of $O(n)$ in the number of primitives. When the cost of traversing tree nodes is sufficiently low, this strategy scales well for arbitrarily constructed scenes and allows an exponential increase in the number of primitives we can render in a fixed time. For scenes with specific kinds of structure we may be able to do even better. For example, say that we could find an indexing scheme or hash function that can divide our scene into $O(n)$ buckets that allow conservative culling with $O(1)$ primitives per bucket. This would approach $O(d)$-time visibility testing in the distance d to the first intersection. When that distance is small (e.g., in twisty corridors), the runtime of this scheme for static scenes becomes independent of the number of primitives and we can theoretically render arbitrarily large scenes. See Chapter 37 for a detailed discussion of algorithms based on these ideas.

15.9 Discussion

Our goal in this chapter was not to say, "You can build either a ray tracer or a rasterizer," but rather that rendering involves sampling of light sources, objects, and rays, and that there are broad algorithmic strategies you can use for accumulating samples and interpolating among them. This provides a stage for all future rendering, where we try to select samples efficiently and with good statistical characteristics.

For sampling the scene along eye rays through pixel centers, we saw that three tests—explicit 3D ray-triangle tests, 2D ray-triangle through incremental barycentric tests, and 2D ray-triangle through incremental edge equation tests—were mathematically equivalent. We also discussed how to implement them so that the mathematical equivalence was preserved even in the context of bounded-precision arithmetic. In each case we computed some value directly related to the barycentric weights and then tested whether the weights corresponded to a point on the interior of the triangle. It is essential that these are mathematically equivalent tests. Were they not, we would not expect all methods to produce the same image! Algorithmically, these approaches led to very different strategies. That is because they allowed amortization in different ways and provoked different memory access patterns.

Sampling is the core of physically based rendering. The kinds of design choices you faced in this chapter echo throughout all aspects of rendering. In fact, they are significant for all high-performance computing, spreading into fields as diverse as biology, finance, and weather simulation. That is because many interesting problems do not admit analytic solutions and must be solved by taking discrete samples. One frequently wants to take many of those samples in parallel to reduce computation latency. So considerations about how to sample over a complex domain, which in our case was the set product of triangles and eye rays, are fundamental to science well beyond image synthesis.

The ray tracer in this chapter is a stripped-down, no-frills ray tracer. But it still works pretty well. Ten years ago you would have had to wait an hour for the teapot to render. It will probably take at most a few seconds on your computer today. This performance increase allows you to more freely experiment with the algorithms in this chapter than people have been able to in the past. It also allows you to exercise clearer software design and to quickly explore more sophisticated algorithms, since you need not spend significant time on low-level optimization to obtain reasonable rendering rates.

Despite the relatively high performance of modern machines, we still considered design choices and compromises related to the tension between abstraction and performance. That is because there are few places where that tension is felt as keenly in computer graphics as at the primary visibility level, and without at least *some* care our renderers would still have been unacceptably slow. This is largely because primary visibility is driven by large constants—scene complexity and the number of pixels—and because primary visibility is effectively the tail end of the graphics pipeline.

Someday, machines may be fast enough that we don't have to make as many compromises to achieve acceptable rendering rates as we do today. For example, it would be desirable to operate at a purely algorithmic level without exposing the internal memory layout of our `Image` class. Whether this day arrives soon depends on both algorithmic and hardware advances. Previous hardware performance increases have in part been due to faster clock speeds and increased duplication of parallel processing and memory units. But today's semiconductor-based processors are incapable of running at greater clock speeds because they have hit the limits of voltage leakage and inductive capacitance. So future speedups will not come from higher clock rates due to better manufacturing processes on the same substrates. Furthermore, the individual wires within today's processors are close to one molecule in thickness, so we are near the limits of miniaturization for circuits. Many graphics algorithms are today limited by communication between parallel processing units and between memory and processors. That means that simply increasing the number of ALUs, lanes, or processing cores will not increase performance. In fact, increased parallelism can even decrease performance when runtime is dominated by communication. So we require radically new algorithms or hardware architectures, or much more sophisticated compilers, if we want today's performance with better abstraction.

There are of course design considerations beyond sample statistics and raw efficiency. For example, we saw that if you're sampling really small triangles, then micropolygons or tile rasterization seems like a good rendering strategy. However, what if you're sampling shapes that aren't triangles and can't easily be subdivided?

Shapes as simple as a sphere fall into this category. In that case, ray casting seems like a very good strategy because you can simply replace ray-triangle intersection with ray-sphere intersection. Any micro-optimization of a rasterizer must be evaluated compared to the question, "What if we could render one nontriangular shape, instead of thousands of small triangles?" At some point, the constants make working with more abstract models like spheres and spline surfaces more preferable than working with many triangles.

When we consider sampling visibility in not just space, but also exposure time and lens position, individual triangles become six-dimensional, nonpolyhedral shapes. While algorithms for rasterizing these have recently been developed, they are certainly more complicated than ray-sampling strategies. We've seen that small changes, such as inverting the order of two nested loops, can yield significant algorithmic implications. There are many such changes that one can make to visibility sampling strategies, and many that have been made previously. It is probably best to begin a renderer by considering the desired balance of performance and code manageability, the size of the triangles and target image, and the sampling patterns desired. One can then begin with the simplest visibility algorithm appropriate for those goals, and subsequently experiment with variations.

Many of these variations have already been tried and are discussed in the literature. Only a few of these are cited here. Appel presented the first significant 3D visibility solution of ray casting in 1968. Nearly half a century later, new sampling algorithms appear regularly in top publication venues and the industry is hard at work designing new hardware for visibility sampling. This means that the best strategies may still await discovery, so some of the variations you try should be of your own design!

15.10 Exercises

Exercise 15.1: Generalize the `Image` and `DepthBuffer` implementations into different instances of a single, templated buffer class.

Exercise 15.2: Use the equations from Section 7.8.2 to extend your ray tracer to also intersect spheres. A sphere does not define a barycentric coordinate frame or vertex normals. How will you compute the normal to the sphere?

Exercise 15.3: Expand the barycentric weight computation that is abstracted in the `bary2D` function so that it appears explicitly within the per-pixel loop. Then lift the computation of expressions that are constant along a row or column outside the corresponding loop. Your resultant code should contain a single division operation within the inner loop.

Exercise 15.4: Characterize the asymptotic performance of each algorithm described in Section 15.6. Under what circumstances would each algorithm be preferred, according to this analysis?

Exercise 15.5: Consider the "1D rasterization" problem of coloring the pixel centers (say, at integer locations) covered by a line segment lying on the real number line.

1. What is the longest a segment can be while covering no pixel centers? Draw the number line and it should be obvious.
2. If we rasterize by snapping vertices at real locations to the nearest integer locations, how does that affect your answer to the previous question?

(Hint: Nothing wider than 0.5 pixels can now hide between two pixel centers.)

3. If we rasterize in fixed point with 8-bit subpixel precision and snap vertices to that grid before rasterization, how does that affect your answer? (Hint: Pixel centers are now spaced every 256 units.)

Exercise 15.6: Say that we transform the final scene for our ray tracer by moving the teapot 10 cm and ground to the right by adding 10 cm to the *x*-ordinate of each vertex. We could also accomplish this by leaving the teapot in the original position and instead transforming the ray origins to the left by 10 cm. This is the Coordinate-System/Basis principle. Now, consider the case where we wish to render 1000 teapots with identical geometry but different positions and orientations. Describe how to modify your ray tracer to represent this scene without explicitly storing 1000 copies of the teapot geometry, and how to trace that scene representation. (This idea is called **instancing.**)

Exercise 15.7: One way to model scenes is with **constructive solid geometry** or **CSG:** building solid primitives like balls, filled cubes, etc., transformed and combined by boolean operations. For instance, one might take two unit spheres, one at the origin and one translated to $(1.7, 0, 0)$, and declare their intersection to be a "lens" shape, or their union to be a representation of a hydrogen molecule. If the shapes are defined by meshes representing their boundaries, finding a mesh representation of the union, intersection, or difference (everything in shape *A* that's *npt* in shape *B*) can be complex and costly. For ray casting, things are simpler.
(a) Show that if *a* and *b* are the intervals where the ray *R* intersects objects *A* and *B*, then *acupb* is where *R* intersects $A \cap B$; show similar statements for the intersection and difference.
(b) Suppose a CSG representation of a scene is described by a tree (where edges are transformations, leaves are primitives, and nodes are CSG operations like union, intersection, and difference); sketch a ray-intersect-CSG-tree algorithm. Note: Despite the simplicity of the ideas suggested in this exercise, the speedups offered by bounding-volume hierarchies described in Chapter 37 favor their use, at least for complex scenes.

Index

Page numbers followed by "f" indicate a figure; and those followed by "t" indicate a table.