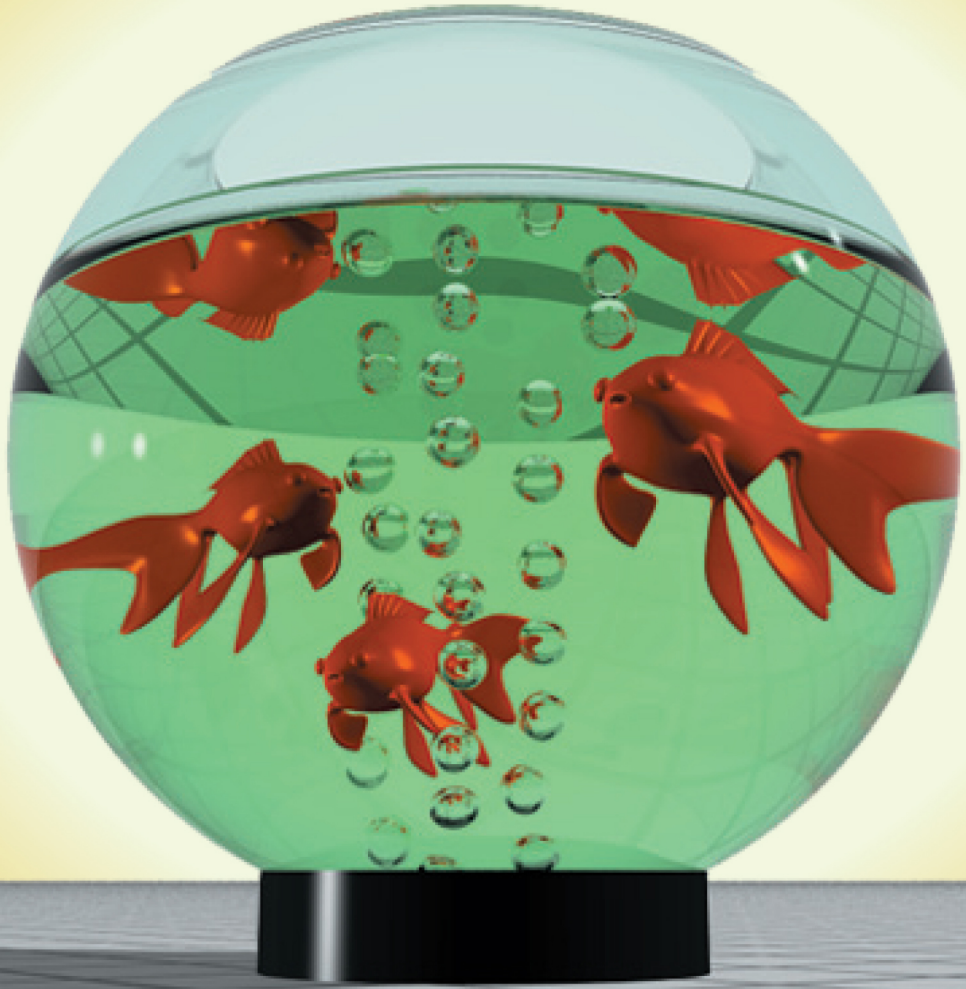
Ray Tracing from the Ground Up



Kevin Suffern

CRC Press
Taylor & Francis Group
AN A K PETERS BOOK

Ray Tracing from the Ground Up

Kevin Suffern

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

AN A K PETERS BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

First issued in hardback 2019

ISBN-13: 978-1-56881-272-4 (hbk)

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Library of Congress Cataloging-in-Publication Data

Suffern, Kevin G.
Ray tracing from the ground up / Kevin G. Suffern.
p. cm.
Includes bibliographical references and index.
ISBN 978-1-56881-272-4 (alk. paper)
1. Computer graphics. I. Title.
T385.S7995 2007
006.6--dc22 2007021706

In memory of
Silvia Gladys Suffern, 1910-2003
Lucy Suffern, 1977-1997

To Eileen, I could not have written this without you.

Contents

	Foreword	ix
	Preface	xiii
	Acknowledgments	xix
1	Ray Tracer Design and Programming	1
2	Some Essential Mathematics	17
3	Bare-Bones Ray Tracing	45
4	Antialiasing	81
5	Sampling Techniques	93
6	Mapping Samples to a Disk	119
7	Mapping Samples to a Hemisphere	125
8	Perspective Viewing	133
9	A Practical Viewing System	151
10	Depth of Field	167
11	Nonlinear Projections	181
12	Stereoscopy	197
13	Theoretical Foundations	217
14	Lights and Materials	245

15	Specular Reflection	279
16	Shadows	293
17	Ambient Occlusion	309
18	Area Lights	325
19	Ray-Object Intersections	353
20	Affine Transformations	397
21	Transforming Objects	417
22	Regular Grids	443
23	Triangle Meshes	473
24	Mirror Reflection	493
25	Glossy Reflection	529
26	Global Illumination	543
27	Simple Transparency	561
28	Realistic Transparency	593
29	Texture Mapping	643
30	Procedural Textures	671
31	Noise-Based Textures	691
	References	733
	Index	745

Foreword

Computer graphics involves simulating the distribution of light in a 3D environment. There are only a few fundamentally different algorithms that have survived the test of time. They can be loosely classified into projective algorithms and image-space algorithms. The former class projects each geometric primitive onto the image plane, with local shading taking care of the appearance of objects. This class of algorithm is still widely used because it is amenable to pipeline processing and therefore to hardware implementation as evidenced by all modern graphics cards.

Image-space algorithms compute the color of each pixel by figuring out where the light came from for that pixel. Here, the basic operation is to determine the nearest object along a line of sight. Following light back along a line has given this basic operation and the associated image-synthesis algorithm their name: ray tracing.

In 1980, ray tracing was at the forefront of science. The quality of the images that can be computed with ray tracing was an eye opener, as it naturally includes light paths such as specular reflection and transmission, which are difficult to compute with projective algorithms. Some shapes are easier to intersect rays with than others, and in those early days, spheres featured heavily in ray-traced images. Hence, old images often contained shiny spheres to demonstrate the power of ray tracing.

A vast amount of research was then expended to make ray tracing both more tractable and to include more features. Variants were introduced, for instance, that compute diffuse inter-reflection, caustics, and/or participating media. To speed up image generation, many data structures were developed that spatially sort the 3D geometry. Spatial subdivision algorithms allow a very substantial reduction of the candidate set of objects that need to be

intersected to find the nearest object for each ray. Ray tracing is also amenable to parallel processing and has therefore attracted a substantial amount of research in that area.

All of this work moved ray tracing from being barely tractable, to just about doable for those who had state-of-the-art computers and plenty of time to kill. High-quality rendering tends to take a whole night to complete, mostly because this allows artists to start a new rendering before going home, to find the finished image ready when they arrive at work the next day. This, by the way, still holds true. For many practical applications, hardware and algorithmic improvements are used for rendering larger environments, or to include more advanced shading, rather than to reduce the computation time.

On the other hand, more than 25 years after its introduction, ray tracing has found a new lease on life in the form of interactive and real-time implementations. Such rendering speeds are obtained by using a combination of super-fast modern hardware, parallel processing, state-of-the-art algorithms, and a healthy dose of old-fashioned low-level engineering. Recent advances have enabled ray tracing to be a useful alternative for real-time rendering of animated scenes, as well as huge scenes that do not fit into main memory. In addition, there is a trend towards the development of dedicated hardware for ray tracing.

All of this research has helped to push ray tracing from an interesting esoteric technique for image synthesis to a seriously viable algorithm for practical applications. If necessary, ray tracing can operate in real time. If desired, ray tracing can be physically based and can therefore be used in predictive lighting simulations. As a result, ray tracing is now used in earnest in the movie industry, but also, for instance, in the automotive industry and in scientific visualization. In addition, it forms the basis for several other graphics algorithms, including radiosity and photon mapping.

The practical importance of ray tracing as a lighting-simulation technique means that ray tracing needs to be taught to students, as well as to practitioners in industry. In addition, ray tracing is sufficiently multifaceted that teaching students all aspects of the algorithm will give them all manner of additional benefits: 3D modeling skills, mathematics skills, software engineering skills (writing a ray tracer is for many students the first time that they will have to manage a sizeable chunk of code), hands-on experience in object-oriented programming, and deeper insights into the physics of light, as well as knowledge of the behavior of materials.

It would be ideal to present a ray-tracing course to students at the undergraduate level for all of the above reasons, but also because a deep understanding of ray tracing will make it easier to grasp other image-synthesis algorithms.

For this, a book is required that explains all facets of ray tracing at the right pace, assuming only a very moderate amount of background knowledge.

I'm positively delighted that such a book now exists. *Ray Tracing from the Ground Up* not only covers all aspects of ray tracing, but does so at a level that allows both undergraduate and graduate students to appreciate the beauty and algorithmic elegance of ray tracing. At the same time, this book goes into more than sufficient detail to deserve a place on the bookshelves of many professionals as a reference work.

Kevin was gracious enough to let me read early drafts of several chapters when I was teaching a graduate-level ray-tracing course at the University of Central Florida. This has certainly taught me many of the lesser-known intricacies of ray tracing. Kevin himself has taught ray tracing to undergraduate students for many years, and it shows. This book, which grew out of his course notes, is remarkably easy to follow, especially given the complexity of the subject matter.

As such, I can heartily recommend this book to both professionals as well as students and teachers. Whether you are only interested in rendering a collection of shiny spheres or want to create stunning images of highly complicated and realistic environments, this book will show you how. Whether its intended use is as a ray-tracing reference or as the basis of a course on ray tracing, this book is essential reading.

Erik Reinhard
University of Bristol
University of Central Florida

Preface

Where Did This Book Come From?

Since the early 1990s, I have had the privilege of teaching an introductory ray-tracing course at the University of Technology, Sydney, Australia. This book is the outcome of all of those years in the classroom. The ray tracer presented here has been developed and taught over the years, during which time my students have provided invaluable feedback, bug reports, ideas, and wonderful images. The book's manuscript has evolved from the teaching notes for the course and has been written (and re-written) chapter by chapter as the ray tracer, and my teaching of it, have developed. Writing in a teaching context with the feedback provided by students has helped me produce a book that is, I hope, much more understandable than it would have been had I written it in isolation. I like to call the iterative processes of programming, writing, and teaching ray tracing the *ray-tracing circle*.

What Is Ray Tracing?

Ray tracing is a computer-graphics technique that creates images by shooting rays. It's illustrated in Figure 1, which shows a camera, a window with pixels, two rays, and two objects. The rays go through pixels and are tested for intersection with the objects. When a ray hits an object, the ray tracer works out how much light is reflected back along the ray to determine the color of the pixel. By using enough pixels, the ray tracer can produce an image of the objects. If the objects are reflective, the rays can bounce off of them and hit other objects.

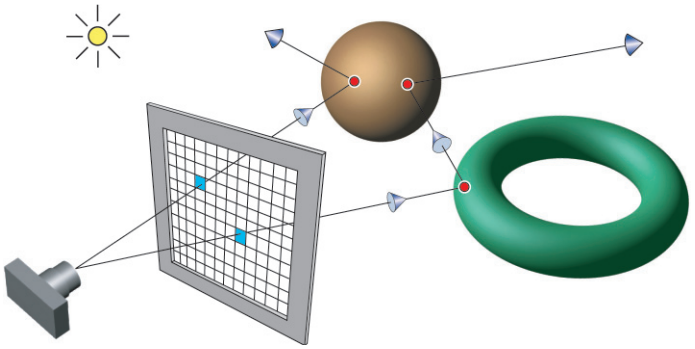


Figure 1. The ray-tracing process.

This process is conceptually simple, elegant, and powerful. For example, it allows ray tracing to accurately render reflections, transparent objects, shadows, and global illumination. Ray tracing can also render large triangle meshes more efficiently than other rendering techniques.

Why Is Ray Tracing Important?

The production of ever more realistic images is a trend of long standing in 3D computer graphics. This is a task at which ray tracing excels. A major application of ray tracing is the film industry, not just for visual effects, but for rendering whole movies. For example, the animated films *Ice Age*, *Ice Age 2*, and *Robots* were fully ray traced, as were the short films *Bunny* and *The Cathedral* (a.k.a. *Katedra*). *Ice Age* was nominated for the Academy Award for Best Animated Feature in 2002, *Bunny* won the Academy Award for Animated Short Film in 1998, and *The Cathedral* was nominated for the Academy Award for Animated Short Film in 2002. Ray tracing was also used in *Happy Feet* to render the penguins with ambient occlusion, and for reflection and refraction with the ocean surface (see Chapters 17, 27, and 28). *Happy Feet* won the Academy Award for Best Animated Feature in 2006.

The major software packages used in the visual-effects industry have built-in ray tracers, and there are numerous state-of-the art ray tracers available as plug-ins or stand-alone applications. These include Brazil (http://www.splutterfish.com/), Mental Ray (http://www.mentalimages.com/), finalRender (http://www.finalrender.com/), and Maxwell Render (http://www.maxwellrender.com/). Cinema 4D (http://www.maxon.net/) also has a state-of-the art ray tracer.

Real-time 3D computer games also have an increasing demand for realism. Although current PCs are not fast enough for real-time ray-traced games, this is likely to change in the next few years. The introduction of chips with specialized graphics processors on multiple cores that can be programmed using existing programming tools will make this possible. Because each ray can be traced independently, ray tracing can trivially use as many processors as are available. In fact, ray tracing has been described as being "embarrassingly parallelizable."[1] These hardware advances should also result in ray tracing being used more frequently in the visual-effects industry.

All this means that ray tracing has a great future, and within a few years you should be able to use the techniques you will learn in this book to write real-time ray-tracing applications such as games.

Graphics education also benefits greatly. My experience has been that getting students to write a ray tracer is the best way for them to understand how rendering algorithms work. Ray tracing's flexibility and ease of programming is the primary reason for this.

In a more general context, ray tracing helps us understand the appearance of the world around us. Because it simulates geometric optics, ray tracing can be used to render many familiar optical phenomena. The appearance of the fish and bubbles on the front cover is an example.

Book Features

This book provides a detailed explanation of how ray tracing works, a task that's accomplished with a combination of text, code samples, about 600 ray-traced images, and over 300 illustrations. Full color is used throughout. Almost all of the ray-traced images were produced with the software discussed here. The book also showcases the work of about 25 students. You can develop the ray tracer chapter by chapter.

Most chapters have questions and exercises at the end. The questions often ask you to think about ray-traced images; the exercises cover the implementation of the ray tracer and suggest ways to extend it. There are almost 400 questions and exercises.

Shading is described rigorously as solutions to the rendering equation and is specified in radiometric terms such as radiance (see Chapter 13).

1. Alan Norton, circa 1984, personal communication.

The book's website (http://www.raytracegroundup.com) contains several animations that demonstrate effects and processes that are difficult or impossible to see with static images.

Pathways through This Book

You don't have to read the chapters in order, or read all of the material in every chapter, or read all of the chapters. For example, Chapters 2 and 20 cover some of the mathematics you need for ray tracing, and you may already be familiar with this; you can read Chapter 19 on ray-object intersections, or parts of it, when you need to.

Chapters 1–4, 9, and 13–16 cover ray-tracing fundamentals, perspective viewing with a pinhole camera, theoretical foundations, and basic shading. Chapter 13 is heavy going mathematically but provides the essential theoretical foundations for the following chapters on shading. The good news is that you don't have to master all of the material in Chapter 13. Most of the complicated integrals in this and the following chapters can be expressed in a few simple lines of code, which are in the book.

Chapter 24 covers mirror reflection, Chapters 27 and 28 cover transparency, and Chapters 29–31 cover texturing. Although You will find many interesting things to explore here, and you can read the texturing chapters first, if you want to.

If you read the sampling chapters, Chapters 5–7, you will have the background to understand the different camera models in Chapters 10–12, ambient occlusion in Chapter 17, area lights in Chapter 18, glossy reflection in Chapter 25, and global illumination in Chapter 26.

Chapter 21 explains how to ray trace transformed objects, and Chapter 22 covers grid acceleration, which is the tool for ray tracing triangle meshes in Chapter 23.

What Knowledge and Skills Do You Need?

Because the ray tracer is written in C++, you should be reasonably proficient at C++ programming. A first course in C++ should be sufficient preparation, but there is a heavy emphasis on inheritance, dynamic binding, and polymorphism right from the start. That's a critical design element, as I'll explain in Chapter 1.

You should also be familiar with coordinate geometry, elementary trigonometry, and elementary vector and matrix algebra. Although there is some

calculus in the book, I usually just quote the results and give you the relevant code.

You don't need previous studies in computer graphics because the book is self-contained in this regard. Chapters 3, 8, and 13 cover the necessary graphics background material. As far as graphics output is concerned, ray tracing is a simple as possible—you just draw pixels into a window on your computer screen.

Intended Audience

The book is intended for computer-graphics students who have had at least an introductory course in C++. The book is suitable for both undergraduate and graduate courses.

It's also intended for anyone with the required background who wants to write a ray tracer or who wants find out how ray tracing works. This includes people working in the computer-graphics industry.

Online Resources

The book's website is at http://www.raytracegroundup.com, where you will find:

- the skeleton ray tracer described in Chapter 1;
- sample code;
- triangle mesh files in PLY format;
- image files in PPM format;
- the ray-traced images in JPEG format;
- additional images;
- C++ code for constructing scenes;
- animations;
- a place where you can post errata;
- useful links.

Topics Not Covered

Due to time and space constraints, here are some of the many topics that I have not discussed: high dynamic range (HDR) imaging with local tone-map-

ping operators; comparisons between grid acceleration and other acceleration schemes such as bounding volume hierarchies; an efficient global illumination algorithm; an efficient technique for rendering caustics; sub-surface scattering; bump mapping; the volumetric rendering of participating media. Although HDR imaging would require some serious retrofitting, the other topics could be implemented as add-ons. Any of these could be student assignments or projects.

Acknowledgments

It's traditional for authors to start by thanking their long-suffering families. This is not a tradition I'm about to break. First, my thanks go to my wife, Eileen, without whose support this book would not have been written. Eileen read the whole book about three times, and then the proofs, corrected innumerable grammar mistakes, improved my writing, and did a lot of work on the index. She did all the cooking and housework through this period, while completing her own diploma studies, and watched in despair as the number of urgent repair jobs on the house grew and grew. As a result, I probably have about a year's work of repairs awaiting me.

My son Chris provided expert help with Adobe Illustrator and did all the curves in the figures. Chris also did the photography and graphic design for Figure 29.26. I'm very grateful for Chris's help. My son Tim checked the references, for which I am also very grateful. And finally, my four-year-old grandson Broden gave me permission to use his photograph in Figure 29.26.

Paul Perry, a friend of 40 years, provided on more than one occasion advice and a quiet refuge in Melbourne for writing.

I would like to thank Erik Rienhard, who acted as my reviewer. Erik read the whole manuscript, found many mistakes, and made many suggestions that have improved this book. I'm honored that he also wrote the foreword. Frequent phone conversations with him were very helpful.

I would also like to thank Pete Shirley, who has helped and inspired me in several ways. For example, Pete suggested that I use orthonormal bases and use instances for ray tracing transformed objects. There is a lot of Pete's work in this book, and it is the better for it.

The following people trialed early versions of the manuscript in their ray-tracing courses: Dave Breen at Drexel University, Bob Futrelle at Northeastern University, Steve Parker at the University of Utah, Erik Reinhard at the University of Central Florida, and Pete Shirley at the University of Utah. I wish to thank all of them.

Special thanks must go to Alice Peters at A K Peters, who believed in this book, right from the start, was amazingly patient during interminable delays on my part, and worked extremely hard to get it published for SIGGRAPH 2007. I particularly wish to thank Kevin Jackson-Mead, Senior Editor at A K Peters, who expertly edited the manuscript. This was a huge job. I also thank the typesetter, Erica Schultz, for an expert job.

Thanks go to Darren Wotherspoon at Skye Design for the beautiful cover design.

Richard Raban also deserves special thanks. Richard, as my Department Head at the University of Technology, Sydney, gave me a lot of support and a working environment that made writing easier. Chris W. Johnson and Helen Lu at UTS also read several chapters and pointed out mistakes.

The following students graciously provided permission to use their ray-traced images: Steve Agland, Nathan Andrews, John Avery, Peter Brownlow, David Gardner, Peter Georges, Mark Howard, Tania Humphreys, Daniel Kaestli, Adeel Khan, Mark Langsworth, Lisa Lönroth, Alistair McKinley, Jimmy Nguyen, Riley Perry, Duy Tran, and Ving Wong. Thank you all for your beautiful images, which have enhanced this book.

Thanks also go to Tania Humphreys for modeling and rendering the Mirage 2000 device by Opti-Gone Associates, to Alistair McKinley for implementing the ray-visualization software that made Figure 27.24(b) possible, and to Jimmy Nguyen for performing a number of file conversions.

The following students pointed out errors in the manuscript: Deepak Chaudhary, Tim Cooper, Ronnie Sunde, Ksenia Szeweva, and Mark White.

I must thank Naomi Hatchman for allowing me to use her penguin model in Chapters 23 and 29 and for a lot of work in supplying files in various formats and triangle resolutions.

I also thank James McNess for permission to use his goldfish model, shown on the cover and in Chapters 23 and 28.

There are two critical pieces of software associated with this book. One is the skeleton ray tracer. I'd like to thank Peter Kohout and James McGregor for producing early versions. I particularly thank Sverre Kvåle for writing the current version. Sverre also converted all the code files to PC format and tested the sample code. I am most grateful for all his work and his great programming skills.

The other is the website, built by HwaLeon (Ayo) Lee. Thank you, Ayo, for your generosity and professional development skills.

Ayo, James McGregor, Naomi, Peter, and Sverre have been students of mine; James McNess has been a student of Naomi.

Another student, Hong Son Nguyen, kindly produced a number of computer animations for the book and allowed me to put these on the website. Thank you, Hong.

My students, Naomi Hatchman, André Mazzone, Glen Sharah, Rangi Sutton, and the "UTS Amigos": Steve Agland, Peter Brownlow, Chris Cooper, Peter Georges, Justen Marshall, Adrian Paul, and Brian Smith, have inspired me by earning credits on Academy Award winning films for their outstanding visual-effects work. The films include *Happy Feet*, *The Lord of the Rings: The Return of the King*, and *The Matrix*.

I have also benefited greatly from discussions with Paul Bourke about nonlinear projections and stereoscopy. Paul read Chapters 11 and 12, and as a result of his feedback, these chapters are now more relevant to real-world applications. He also permitted me to use material in these chapters from his website at http://local.wasp.uwa.edu.au/~pbourke/. I thank Paul for all his help.

I also thank Paul Debevec for permission to use the Uffizi probe image from his website at http://www.debevec.org/Probes/ and for providing advice on light-probe mappings.

Henrik Jensen provided advice on various aspects of ray tracing.

The following people and organizations allowed me to use various information. Stephen Addleman from Cyberware: images of the horse, Isis, and Ganesh models from http://www.cyberware.com/; James Hastings-Trew: Earth images in Chapter 29 from his website at http://planetpixelemporium.com/planets.html; Phillipe Hurbain: sky images from http://www.philohome.com/ in Chapters 11 and 12; Michael Levin from Opti-Gone International: material from http://www.optigone.com/ about the Mirage 2000 in Chapter 12; Ric Lopez-Fabrega at Lopez-Fabrega Design: sky images from http://www.lfgrafix.com in Chapter 29; Morgan Kaufmann Publishers: images and code in Chapter 31; Steve Parker at the University of Utah: the Cornell-box image in Chapter 26; Greg Turk: PLY code and various PLY models of the Stanford bunny at http://www.cc.gatech.edu/projects/large_models/index.html. I thank you all.

Finally, I wish to thank Eric Haines and Pete Shirley for graciously providing endorsements for the book.

1 Ray Tracer Design and Programming

1.1 General Approaches
1.2 Inheritance
1.3 Language
1.4 Building Scenes
1.5 The User Interface
1.6 Skeleton Ray Tracer
1.7 Developing the Ray Tracer
1.8 Floats or Doubles
1.9 Efficiency Issues
1.10 Coding Style
1.11 Debugging

Image courtesy of Jimmy Nguyen
Skeleton model from Clemson University

A ray tracer with any reasonable set of features is a large and complex software system that must be designed carefully and developed in a systematic manner. This chapter gives you guidelines for the design and programming of a ray tracer. You can also find information on these topics in Glassner (1989), Wilt (1994), Shirley (2002), Shirley and Morley (2003), and Pharr and Humphreys (2004).

1.1 General Approaches

It's best to develop your ray tracer using object-oriented (OO) techniques for several reasons. The first is size and complexity. Object-oriented techniques are best for handling the design and implementation of large and complex systems, and ray tracers can certainly be large and complex. One of the largest was the Kilauea ray tracer, which consisted of about 700,000 lines of C++ code (Kato et al., 2001, Kato, 2002). Although the ray tracer I discuss here is

not nearly this large, it is large and complex enough for OO techniques to be essential for its development.

The second reason is extensibility, which is not really separate from the first, because large and complex software systems are developed by extending simpler systems. Ray tracers can be extended in many ways, for example, by adding new types of geometric objects. You should design your ray tracer so that adding new types of objects is as simple as possible. OO techniques allow you to do this without altering the existing code that renders the objects. Your ray tracer will also have to deal with different types of cameras, samplers, lights, BRDFs, materials, mappings, textures, noises, and bump maps. Adding a new type of any of these things should also be as simple as possible.

Let's look at how OO techniques facilitate these processes. The code that performs the ray-object intersections should not have to know the type of objects it deals with. Why? Because it would then have to explicitly identify the type of each object, and intersect it in a case or switch statement. This makes it more work to program because you must provide an identifier for each type, and add a new clause to the switch statement. To make matters worse, the switch statement may have to appear in more than one place in the ray tracer. It's far better for objects to be *anonymous* in the intersection part of the ray tracer. To do this you define a *uniform public interface* for the intersection (hit) functions, so that they are called the same way for all objects. The ray tracer, which can still identify the type of each object at run time, will then call the correct hit function.

You should also apply the same process to lights, materials, textures, etc. Except for build functions and `#include` statements, your ray tracer should not have to explicitly identify the type of anything that it deals with. From a design and development perspective, this is the most important aspect of your ray tracer code.

Kirk and Arvo (1988) discussed the above issues for the first time in a ray tracing context, including the use of a common user interface for all objects. This was the first paper on object oriented ray tracing; it contains many good ideas.

1.2 Inheritance

The best way to implement the above processes is to use *inheritance*. Figure 1.1 shows a sample geometric object inheritance chart. Provided you implement the objects correctly, *dynamic binding* guarantees that the correct hit functions are called. This is called *polymorphism*.

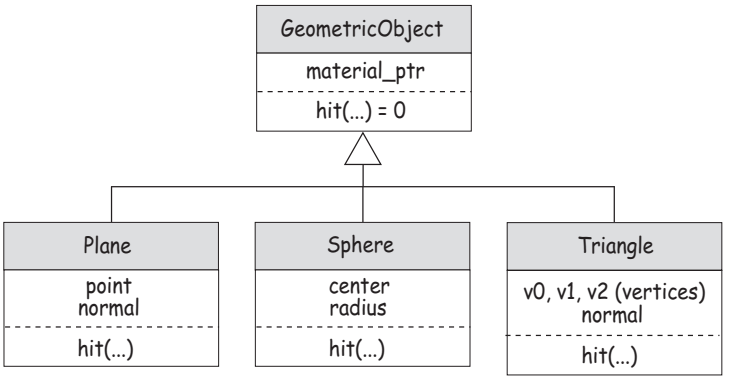


Figure 1.1 A sample object inheritance chart.

Code re-use is another benefit of inheritance. The fact that derived classes can use all the code in their base classes means that you only have to write the new parts when you add derived classes. For example, the code to handle the material only has to be written once in the `GeometricObject` class.

1.3 Language

The sample code in this book is C++ because it's the mainstream ray tracing language. The reasons are: its OO facilities, the ability to mix C and C++ code in the same program, and its computational efficiency. State of the art commercial ray tracers such as Brazil, Mental Ray, finalRender, and Maxwell Render are written in C++. Also, many major commercial rendering and animation packages, which were originally written in C, were re-written from the ground up in C++ during the 1990s. RenderMan plug-ins can also be written in C++.

Computational efficiency is important for ray tracers because of their extensive use of floating point calculations. C++ programs can be written efficiently because of the C heritage of C++, and the fact that they are fully compiled. If you are not careful, however, you can write inefficient C++ programs. The books by Meyers (1996, 2001, 2005), Lippman et al. (2005), and Bulka and Mayhew (2000) discuss computational efficiency in C++. Writing efficient code is, however, not as important as writing code that's easy to read, easy to maintain, and easy to extend. In other words, efficiency is not as important as good design. Fortunately, the two can go hand in hand in C++. Also, efficiency is not as important as using language features that make your job as a programmer easier. For example, there can be a cost penalty with inheritance in C++

because of the extra indirection of virtual function calls, and because dynamic binding prevents the inlining of virtual functions, but this is far outweighed by the benefits.

C++ can also use the Standard C library functions, and has excellent library facilities of its own, particularly the Standard Template Library (STL). See Lippman et al. (2005), Meyers (2001), and Ford and Topp (2001). C++ also has operator overloading that allows the code for many vector and matrix operations to be written in mathematical-like notation.

Public domain C code, such as code for solving cubic and quartic polynomial equations (Schwarze, 1990), can be simply incorporated into C++ programs, as can the efficient C macros for generating lattice noises by Peachey (2003). C++ also has an ANSI ISO standard that is platform independent.

Finally, there are good integrated development environments (IDEs) for C++ on all common computer platforms, but you should make sure that the code is compiled.

1.4 Building Scenes

Ray tracers render *scenes*. Before a scene can be rendered, the ray tracer needs everything to be specified: all the parameters for the camera, geometric objects, lights, materials, textures, etc. The most common way to do this is with a *scene description file*, but another way uses a *build function*. Each approach has its advantages and disadvantages.

Scene description files require a *scene description language* in which the files are written and a parser for the language. This can require a lot of programming. There are also limits to the type of scenes for which you can easily hand-craft a parser. Parsers that can handle control structures such as loops, branches, and recursive function calls are best written with lex and yacc, or Bison and Flex (Levine et al., 1992).

Build functions written in C++ don't require scene description files or parsing, and can exploit the full power of C++. They are functions that are included in your ray tracer code, and although they make it simpler and quicker to add new features, they require your ray tracer to always run in a development environment, at least in the simple approach that I've adopted.[1]

1. The executable produced by the development environment will, of course, run as a stand-alone application, but it can only render a single scene.

This is because the build function also has to be re-compiled each time you change the scene. This isn't a serious problem, for two reasons. First, C++ compilation is fast, and second, you will have to run your ray tracer this way while you are developing it. Your ray tracer would only need to run as a stand-alone program if you wanted to sell it, or distribute it. You could also run it as a stand-alone application if you finished it, but has anyone ever finished a ray tracer?

My experience from teaching both approaches is that build functions allow students more time to concentrate on the ray tracing itself. I therefore use build functions exclusively in this book, starting in Chapter 3. Wilt (1994) was the first ray-tracing book to use this approach.

Your ray tracer will still need to read some information from files. Examples include triangle mesh data and texture images, but I provide code for this.

1.5 The User Interface

This book is about writing the *engine* part of a ray tracer, because graphical user interface elements such as windows, dialogues, and menus, are operating system dependent. Technically, the only "user interface" your ray tracer needs is a command line—it can write the image out to a file, and you can use third-party software to look at it. You can use a command-line interface with Linux, Unix, and Mac OS X, but there's a potential problem. Although most of the scenes I discuss should only take seconds or minutes to render, scenes can take hours or more; in fact, there's no upper limit to ray tracer rendering times.[2] You don't want to wait hours, only to find out that you incorrectly typed one of the viewing parameters in the build function. You should therefore use image-viewing software that allows you to view partially rendered scenes.

From my perspective however, there's a more serious problem with this approach. If your ray tracer doesn't open a window and display the image as it's being rendered, you will miss out on one of life's great pleasures—watching scenes being ray traced. If you haven't done any ray tracing, you will just have to take my word on this for now.

2. Steven Parker has an image on his website (http://www.cs.utah.edu/~sparker) that took two CPU years to ray trace on an eight-processor Silicon Graphics machine. This figure is reproduced in Chapter 26.

1.6 Skeleton Ray Tracer

To help you get started, there's a skeleton ray tracer on the book's website. This runs under Windows and has the following useful features:

- It renders the scene into a window, which can be of arbitrary size.
- It displays the rendering time.
- It can save images in a variety of file formats.
- As it's multi-threaded, it doesn't tie up the computer while it's rendering.

1.7 Developing the Ray Tracer

The following chapters will take you step by step through the process of developing a ray tracer in C++. Chapter 3, *Bare Bones Ray Tracing*, starts with the simplest possible ray tracer: one that only ray traces a single sphere and then adds multiple objects. In later chapters you can add facilities such as antialiasing, samplers, cameras, lights, materials, shadows, reflections, transparency, other types of objects, affine transformations, an acceleration scheme, triangle meshes, and textures. Each chapter is designed to allow you to add new capabilities to the existing ray tracer.

In the early chapters, certain classes are presented in simplified versions, because a lot of the data members and functions are not required until later chapters. An example is the `ShadeRec` class, which I'll be adding data members to throughout the book. To present the complete classes from the beginning would make them too complex. I've therefore employed code that, with few exceptions, uses only the facilities that you need at the time. Because of the fine-grained design of the ray tracer, and the layers of abstraction present, changes will involve additions to existing classes and adding new classes. In particular, the user interfaces of all member functions are frozen from the start, even though some parameters may not be required until later chapters. Examples are the `Plane:hit` and `Sphere:hit` functions discussed in Chapter 3. About the only change you will have to make to an existing function is one line in the main function.

The code samples I'll present and discuss in each chapter will be restricted to simplified class declarations and key functions, but numerous complete classes are on the book's website. The exercises ask you to add the features discussed in each chapter and implement additional features.

The most important thing here is that I'll provide you with a ray-tracer design that has the three principal objectives of extensibility, efficiency, and readability. Design is generally more difficult than programming.

1.8 Floats or Doubles

Doubles provide more numerical precision than floats, but they have twice the memory footprint. As suggested by Peter Shirley, I use doubles for all ray-object intersection calculations, where numerical accuracy is critical, and floats for shading calculations. This involves storing geometric object data members, and the components of all the utility classes as doubles. If you define a global type `typedef float FLOAT;` you can easily change between floats and doubles by changing a single word.

1.9 Efficiency Issues

Efficiency is critical to a ray tracer, as there's no upper limit to rendering times, even for simple scenes when we use sophisticated shading techniques. For example, the opening image in Chapter 26 took 2 CPU years to render! Below are a few relevant issues, mainly at the coding level. I'll discuss broader issues, such as acceleration schemes, in Chapters 22 and 23.

1.9.1 Small is Beautiful

I use a "small is beautiful" design philosophy applied to object sizes and executed code size. This involves a multitude of specialized inheritance hierarchies, objects, materials, lights, functions, etc.; each one designed for a specific task, and individually kept as small, simple, and efficient as possible. Here's an example. I use two hit functions for each object.[3] The first is the ordinary ray-object hit function discussed above, which returns the ray parameter at the nearest hit point, as well as other information required for shading. The second is for shadow rays, which doesn't return the shading information because we don't need it for shadow testing. This approach increases the size of the ray-tracer executable, but that rarely affects the speed of execution. We gain efficiencies by minimizing the code that's actually executed and the size of stored objects. This approach also creates more compilation units, but that's not a problem as most of your builds will be incremental.

The ideal is to be able to add new features with zero impact on the speed of the existing code, but that's often not possible. Instead, I try to add new features in such a way that they have minimum impact.

3. This example is common practice in ray tracing.

1.9.2 Data Storage

The geometric objects, lights, and sample points need to be stored in linear data structures. The STL `vector` class is the most suitable for this purpose because of its speed, see Bulka and Mayhew (2000), and because the collections of objects are usually static. When we know how many objects we are going to store in a `vector`, we can reserve the memory, which speeds up the construction since the underlying C array doesn't have to be resized. This applies to the sample points.

There's also the notational convenience of being able to access a vector element with array notation [ ], which allows a vector to be traversed with code like the following:

```
int  num_objects = objects.size();
for (int j = 0; j < num_objects; j++) {
     if (objects[j]->hit(ray, t, sr) && (t < tmin))
     ...
}
```

where `objects` is a `vector` of `GeometricObject` pointers. There's a small speed penalty for using this notation compared with using iterators.

We also need 2D and 3D data structures in ray tracing: images and sample points (Chapters 5–7) use 2D data structures, regular grids (Chapter 22) and lattice noises (Chapter 31) use 3D data structures. I store these in 1D arrays and use indexing to simulate the 2D and 3D structures. This is also common practice. An exception is the `Matrix` class which uses a 2D C array: `float m[4][4]`, to store the elements.

1.9.3 Pass by Reference

You should pass all compound objects into functions with references or constant references to avoid the construction and destruction of temporary objects. The following triangle constructor declaration is an example:

```
Triangle::Triangle(const Point3D & v1, const Point3D & v2,
   const Point3D & v3);
```

1.9.4 Don't Return by Reference

Don't use a reference to a complex class as the return type of function to try and save a temporary. As an example, the following is a `Matrix` member function for multiplying two matrices.

```
Matrix&
Matrix::operator* (const Matrix& mat) const {
     Matrix product;
     // compute the product ...
     return (product);
}
```

At best, this will cause memory leak, and at worst, the function won't work. Instead, use

```
Matrix
Matrix::operator* (const Matrix& mat) const
```

and let your compiler return the matrix in the most efficient way it can. See Meyers (2005), Item 23. Exceptions include the assignment operators = and *=, and /= that we require for a variety of structured data types.

1.9.5 Avoid Floating-Point Divides

Because a floating-point division requires many more machine cycles on Intel chips than a floating multiplication, it's best to avoid them when you can. One way is to define constants such as `const double invPI = 0.31830988618379067154;`, which avoids having to divide by π.

1.9.6 Use Inlining Judiciously

The judicious use of inlining can help your ray tracer run faster, but it can also have the opposite effect, and actually make it run slower, as well as make debugging difficult. Only inline small functions, don't inline constructors, destructors, or virtual functions, and remember that inlining is only a suggestion to your compiler. You usually have to place inline functions in a header file; see Meyers (2005), Item 33.

1.9.7 Utility Classes

Your ray tracer will need a number of utility classes such as `Vector3D`, `Point3D`, `Normal`, `Matrix`, `RGBColor`, and `ShadeRec`. Because of their ubiquitous use in the ray tracer, it's important that these classes are written as efficiently as possible. The code for these classes is on the book's website.

1.10 Coding Style

I've kept the C++ code as elementary as possible, consistent with getting the job done. Most of the C++ you require should therefore be covered in a first course, but you will have to know how to construct an inheritance hierarchy where the classes allocate memory dynamically. I've only used single inheritance. All code samples are ANSI Standard C++, and some ANSI Standard C.

1.10.1 Identifiers

Class names, data member names, and member function names use the following style, but with some exceptions.

- *Class names* start with upper case, all words in multi-word names start with upper case.
  Examples: `Sphere`, `PointLight`, `GeometricObject`.
- *Member function and data member names* are lower case with the second and subsequent words in multi-word names separated by underscores.
  Examples: `Sphere::center`, `World::add_object(...)`, `ShadeRec::local_hit_point`.
- *Pointer names* end with `_ptr` as in `Sphere* sphere_ptr = new Sphere;`, unless the identifier would be too long. Thus, `glossy_specular_brdf_ptr` is `glossy_specular_brdf`.
- *The names of functions that set or compute data members* usually contain the data-member name after an underscore, or indicate what is being set or computed.
  Examples: `Matte:set_cd()`, where `cd` is the name of the data member, `Camera::compute_uvw(void)`, where `uvw` stands for three data members called `u`, `v`, and `w`.

1.10.2 Concrete Data Types

One of the rules for writing correct C++ code is that classes that allocate memory dynamically must have their own copy constructor, assignment operator, and destructor; see, for example, Meyers (2005). To save space, the class declarations in the text usually don't list these functions, but they are in the electronic versions on the book's website. To provide a uniform class style, I've written most classes as concrete data types even when they don't allocate memory dynamically; see, for example, the `Ray` class declaration in Listing 3.1. Of course, you don't have to use this style.

1.10.3 Encapsulation

Class data members are generally private or protected, but there are exceptions based on their frequency of access. All data members of the following classes are public: `BBox`, `Matrix`, `Normal`, `Point3D`, `Ray`, `RGBColor`, `ShadeRec`, `Vector3D`, `ViewPlane`, and `World`.

1.10.4 Function Signatures

The way you write function signatures can affect the amount of typing you have to do in build functions and the efficiency of scene construction. As an example, consider the function that sets the diffuse color of a `Matte` material and has the following signature:

```
void
Matte::set_cd(const RGBColor& c);
```

This is nice and object-oriented, but every time you call it in a build function, you will have to write something like the following:

```
mattePtr->set_cd(RGBColor(r, g, b));
```

On the other hand, if you use the less object-oriented signature

```
void
MattePtr::set_cd(const float r, const float g, const float b);
```

you can write

```
mattePtr->set_cd(r, g, b);
```

This will not only save you a lot of typing over a lifetime of writing build functions, it's more efficient because it saves an `RGBColor` temporary. If all components of the color are the same, you can use a third version that takes a single float argument. This is particularly useful for setting colors to black, white, and grays. For most set functions of this type, it's best to write all three versions.

1.10.5 Changing a Function Signature

Yes, it may happen that you have to change the signature of a function that's called on a lot of objects with dynamic binding. It's happened to me several

times, the worst one being the ray-object hit function. Here's how you can do it with minimum disruption. Define the new function as a virtual (not pure virtual) function in the base class, and add it to the class you want to test it with. Then, add it to the other objects as you need to. When you've added it to all of the objects, you can make it pure virtual in the base class and get rid of the original version. This technique allows you to test the new function with one object type at a time.

1.10.6 Pure Virtual and Virtual Functions

If a member function has to be defined for every derived class in an inheritance hierarchy, you should declare it as pure virtual in the base class. The example is the function `GeometricObject::hit(...)`. If a function doesn't have to be defined for every derived class, you can declare it as virtual in the base class, and also define it there, to either do nothing, or return a default value. That will keep the C++ compiler happy, even if you don't define it in any derived class. An example is the function `GeometricObject::pdf(...)`, which I have only defined for two geometric objects out of about 30 in my ray tracer. The version in the base just returns 1.0, which is the default value.

1.10.7 File Structure

You should put each class declaration in a header file, such as **`Sphere.h`**, and prevent it from being #included more than once in each compilation unit (see Listing 1.1).

You should put the class definition in a separate file, such as **`Sphere.cpp`**.

```
#ifndef __SPHERE__
#define __SPHERE__

#include "GeometricObject.h"
class Sphere: public GeometricObject {
     // data member and member function declarations ...
};

// inlined functions ...
#endif
```

Listing 1.1. Code fragment from the file **`Sphere.h`**.

1.10.8 Project Structure

I put classes of each type such as geometric objects, materials, and lights, into separate groups, and I include all `.h` files in my IDE project for ease of access. This doubles the number of files in the project, but it's worth it.

1.11 Debugging

As debugging your ray tracer is something you will definitely have to do at some stage, here are a few tips.

1.11.1 Get to Know Your Debugger

First, make sure you are familiar with the debugger in your development environment. This should allow you to set break points, examine variables and the call stack, step through the program, and step into and out of functions.

1.11.2 Ray Trace Single-Pixel Images

Ray tracing has the following nice feature that often simplifies the debugging process: you can ray trace single pixel images. Here's an example. One of my students was implementing an axis-aligned box whose outline was correct, but whose shading was only ambient. This indicated that the hit function was correct but the normal was wrong. He debugged this by using a default box centered on the world origin, placing the camera on the z_w-axis, looking at the origin, and ray tracing a one pixel image with a breakpoint in the material's `shade` function. The box normal at the hit point should have been (0, 0, 1), but was (0, 0, –1) because he had made a mistake when he typed in the `Box::get_normal` function in Chapter 19.

1.11.3 Keep Track of Pixel Coordinates

Maintain two global variables, say `ph` and `pv`, that store the coordinates of the current pixel being rendered. If you find a problem with your image, for example a black dot, you can find the pixel coordinates using an image viewer, and then insert a statement `if (row == ph && column == pv) ...` at a relevant point in the code. Inside the `if` statement, you can place a dummy executable statement that you can put a debugger break point on or print out variables. This technique was also suggested by Erik Reinhard.

1.11.4 Use `cout`

It's often easiest to debug code in loops with `cout` statements. For example, you might be debugging a new sampling technique with 25 rays per pixel and need to look at the sample points for a single pixel. It's a lot quicker and more useful to see them all displayed in a window than having to manually step through the debugger 25 times.

1.11.5 Watch Out for Unallocated Memory

The most common error I get is an *unallocated memory* error caused by trying to access a pointer-based object that has not been constructed or allocated. This is known as sucking vacuum. A way to avoid this is to always check that a pointer is not null before accessing it, but this would slow down your ray tracer too much. I therefore don't do this, and instead live dangerously, but always run with the debugger on when I'm implementing anything new, which is almost all the time.

1.11.6 Simplify

When debugging, it's often best to simplify things as much as possible. Suppose you have just added a new type of object to an existing scene, and it's not working. It's easy to temporarily remove all the other objects by commenting out their `add_object` statements in the build function. You don't have to comment out any of their other code.

1.11.7 Transparency

Transparency code is often difficult to debug because you need recursion depths of at least three to correctly ray trace transparent objects, and the reflected and transmitted rays result in a binary tree of stack frames. It's particularly difficult if there are random errors, where only some pixels are incorrect. The only general guideline I can give you is to use single-pixel images, but fortunately, I've been through all this. As a result, I'm confident that the transparency code and images in Section 7 of the book are correct.

1.11.8 Use the Images Here

How do you know when an image is correct? Although there are no general answers, the hundreds of ray traced images in this book and on the book's website are here to help you.

Further Reading

There are many excellent books that cover C++ design and programming issues. The *C++ Primer*, Fourth Edition (2005) by Lippman, Lajoie, and Moo is comprehensive. *Object-Oriented Programming in C++*, Second Edition (2000) by Johnsonbaugh and Kalin is an excellent introductory book on C++ that emphasizes an object-oriented approach from the start.

If you already know some C++, you owe it to yourself to read Scott Meyers' three books on C++: *Effective C++*, Third Edition (2005), *More Effective C++* (1996), and *Effective STL* (2001). There's nothing quite like these books for their collective insights and wisdom. These books can also help you avoid some of the common mistakes that can make your ray tracers inefficient.

Efficient C++ by Bulka and Mayhew (2000) discuses the relative performance of the `vector` and `list` container classes and is an excellent book on writing efficient C++ programs.

Extreme Programming Explained (2004) by Kent Beck was written by the inventor of the methodology. Although extreme programming is a holistic and integrated set of practices for software development by small teams, it has a number of practices that are applicable to software development by individuals. For example, no matter how large your final application is going to be, you start with the smallest application that does something sensible and build incrementally from there. That's the approach I adopt here for ray-tracer development. In Chapter 3, you will start with the simplest ray tracer that actually does something and then add features to it. Beck's book is well worth a read. Extreme programming is now one of the *agile computing* methodologies.

Brian Kernighan has written two classic books on programming: *The C Programming Language* (1988) with Dennis Ritchie, the developer of C, and *The Unix Programming Environment* (1984) with Rob Pike. The *Practice of Programming* (1999) by Kernighan and Pike discusses many important aspects of programming, and is well worth reading.

An Introduction to Ray Tracing (1989), edited by Andrew Glassner, was the first book on ray tracing. Chapter 7 by Paul Heckbert is called *Writing a Ray Tracer* and discusses features, design issues, advocates an OO approach, and has sample code in C. *Object-Oriented Ray Tracing in C++* (1994) by Nicholas Wilt presents a ray tracer in C++. There are a lot of good ideas and code in this book. *Realistic Ray Tracing*, Second Edition (2003) by Shirley and Morley discusses how to write a modern ray tracer and includes C++ code. Although this book is small, it covers a lot of ground, including Monte Carlo ray tracing, to which Shirley has made significant contributions. *Fundamentals*

of Computer Graphics, Second Edition (Shirley et al., 2005), has a number of chapters on ray tracing and covers a lot of other material that's relevant to ray tracing.

Pharr and Humphreys (2004) discuss a ray tracer using Knuth's literate programming style. This is an excellent book, but the ray tracer is more advanced than the one presented here and is not designed to be implemented step by step by the readers.

2 Some Essential Mathematics

2.1 Sets
2.2 Intervals
2.3 Angles
2.4 Trigonometry
2.5 Coordinate Systems
2.6 Vectors
2.7 Points
2.8 Normals
2.9 Mathematical Surfaces
2.10 Solid Angle
2.11 Random Numbers
2.12 Orthonormal Bases and Frames
2.13 Geometric Series
2.14 The Dirac Delta Function

Image courtesy of Lisa Lönroth

Objectives

By the end of this chapter you should:

- be familiar with some of the mathematics you need for ray tracing;
- understand the difference between vectors, points, and normals;
- understand how to construct an orthonormal frame.

Ray tracing uses a lot of mathematics, from elementary coordinate geometry to multi-dimensional calculus. Fortunately, there are excellent books on all of these topics (see the Further Reading section). I'll present here the mathematical notation used in this book and a number of mathematical topics from a ray-tracing perspective. For example, I'll discuss three-dimensional coordinate systems defined the way we use them in ray tracing, and the difference between vectors, points, and normals. These topics are used throughout the book. Other topics such as barycentric coordinates, which are only used for ray tracing triangles, will be discussed in Chapter 19, where they are used. Except for the presentation of some integrals in Section 2.10, I won't discuss calculus, which is used in Chapter 13 and the subsequent shading chapters. I also won't discuss matrices, which are used in Chapters 20 and 21. Monte Carlo integration will be covered briefly in Chapter 13.

To keep this book to a reasonable size, I've had to draw the line on mathematics somewhere. Most of the topics in this chapter are therefore only covered briefly.

2.1 Sets

2.1.1 Definition and Notation

Definition 2.1. A *set* is an unordered collection of objects of the same type, with a rule for determining if a given object is in the set.

The objects in a set are known as its *elements*. Although the objects can be of any type, in ray tracing, we usually deal with sets of real (floating-point) numbers. Here are some common sets, all of which have an infinite number of elements:

- $\mathbb{R}$: the set of all real numbers. This is also known as the real number line and contains all real numbers from $-\infty$ to $+\infty$.
- $\mathbb{R}^+$: the set of non-negative real numbers, which includes zero.
- $\mathbb{R}^2$: the set of all points on the (x, y) plane.
- $\mathbb{R}^3$: the set of all points (x, y, z) in 3D space.

If s is an element of a set S, we use the notation $s \in S$, where $\in$ means *belongs to*. We also use a notation called *predicate form* to define sets, which is best explained with an example. The definition of $\mathbb{R}^+$ can be written as

$$\mathbb{R}^+ = \{x : x \in \mathbb{R} \text{ and } x \geq 0\}.$$

This can be read as follows: $\mathbb{R}^+$ is the set of all numbers x such that x is real and greater than or equal to zero. Here, the symbol ":" is read as *such that*. I'll use this notation to define intervals in the following section, because it's the most compact way for doing this.

2.1.2 Subsets

We often need to use sets whose elements belong to some larger set, which leads to the concept of *subsets*.

Definition 2.2. A set A is a *subset* of a set B if every element of A also belongs to B. Symbolically, we write this as $A \subseteq B$.

From the above examples, $\mathbb{R}^+ \subseteq \mathbb{R}$. As another example, consider the square S centered on the origin, where $S = \{(x, y) : -1 \leq x \leq 1, -1 \leq y \leq 1\}$. $S \subseteq \mathbb{R}^2$. Also, the set of integers $\mathbb{Z} = \{0, \pm 1, \pm 2, ...\} \subseteq \mathbb{R}$.

2.1.3 Ordered Pairs and the Cartesian Product of Sets

An *ordered pair* (x, y) of elements is a sequence of two elements in a definite order. A common example is a point on the (x, y) plane, where we always write the x-coordinate first, followed by the y-coordinate.

Definition 2.3. For two sets A and B, the set of all ordered pairs of elements (x, y) where $x \in A$ and $y \in B$ is known as the *Cartesian product* of the sets A and B, and is denoted by $A \times B$.

For example, the (x, y) plane is the Cartesian product of the real line with itself, that is, $\mathbb{R}^2 = \mathbb{R} \times \mathbb{R}$. Another example is $\mathbb{R}^3 = \mathbb{R} \times \mathbb{R} \times \mathbb{R}$.

2.2 Intervals

Definition 2.4. An *interval* is a subset of the real line that contains at least two numbers, and contains all of the real numbers lying between any two of its elements.

Intervals are represented geometrically by segments of the real line, and can be finite, semi-infinite, or infinite. Finite intervals have two endpoints that are finite numbers, while infinite intervals have *at most* one finite endpoint and stretch to infinity in one or both directions. In addition, intervals are said to be *open* if neither endpoint is included, *half-open* (or *half-closed*) if one of the endpoints is included, and *closed* if both endpoints are included. We use square brackets [ ] if the endpoints are included in the interval, and parentheses ( ) if they're not included. Infinite intervals can't be closed because no real number is equal to infinity. A standard notation is used for all the types of intervals, as shown in Table 2.1. This is re-written from Thomas and Finney (1996), p. 3.

Type of Interval		Notation	Set Definition	Endpoints
Finite	open	(a, b)	$\{ x : a < x < b \}$	a and b are not in the interval
	closed	$[a, b]$	$\{ x : a \le x \le b \}$	a and b are both in the interval
	half-open	$[a, b)$	$\{ x : a \le x < b \}$	a is in the interval, b is not
	half-open	$(a, b]$	$\{ x : a < x \le b \}$	a is not in the interval, b is
Infinite	open	(a, ∞)	$\{ x : x > a \}$	a is not in the interval
	half-open	$[a, \infty)$	$\{ x : x \ge a \}$	a is in the interval
	open	$(-\infty, b)$	$\{ x : x < b \}$	b is not in the interval
	half-open	$(-\infty, b]$	$\{ x : x \le b \}$	b is in the interval
	open	$(-\infty, \infty)$	$\{ x : -\infty < x < \infty \}$	there are no endpoints

Table 2.1. Interval types, notation, and definitions.

I'll use interval notation extensively in the following chapters to specify the range of numbers that variables can take, for example, $x \in [-1, 1]$, $r \in [0, \infty)$.

The empty interval. The empty interval has no numbers that belong to it, and is denoted by $\varnothing$.

Intersection of intervals. The intersection of two intervals A and B is the set of numbers that belong to both intervals, and is denoted by $A \cap B$. Geometrically, the intersection is the part of the real line where the intervals overlap. For example, if $A = [0, 10]$, $B = [8, 15]$, $A \cap B = [8, 10]$, but if $A = [0, 10]$, $B = [11, 15]$, $A \cap B = \varnothing$.

Cartesian products of intervals. Since intervals are sets, their Cartesian product is defined the same way that it is for sets. The Cartesian product can be used to define finite and infinite planar areas. Here are some examples: the real plane is $\mathbb{R} \times \mathbb{R} = (-\infty, \infty) \times (-\infty, \infty)$; a rectangle is $[0, 1] \times [0, 2]$. When both intervals are the same, I'll often use the notation $A \times A = A^2$. For example, the generic unit square is $[-1, 1] \times [-1, 1] = [-1, 1]^2$.

2.3 Angles

2.3.1 Measurement

In 2D (x, y) coordinates, positive angles are measured from the positive x-axis, in a *counterclockwise* direction, and negative angles are measured *clockwise* from the positive x-axis (see Figure 2.1).

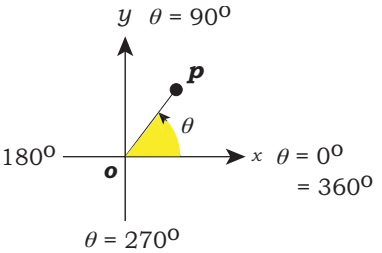


Figure 2.1. Angle definition.

Figure 2.1 also shows the values of the angle θ in degrees and radians along the positive and negative x- and y-axes. There are 360 degrees in a circle, and so along the positive x-axis, $\theta = 0$ degrees, but also $\theta = 360$ degrees. However, angles are not restricted to the range $0 \le \theta \le 2\pi$, because the line ***op*** can be rotated any number of times about the origin in the positive or negative direction.

2.3.2 Degrees and Radians

Angles can be specified in degrees or radians.

Definition 2.5. A radian is the angle subtended at the center of circle by an arc around the circumference whose length is equal to the radius.

Since there are 2π radians and 360 degrees in a circle,

$$1 \text{ radian} = 180° / \pi = 57.29577...°.$$

When specifying angles, most people find it easier to use degrees because these are the common everyday measurement of angles. For example, saying that we want to cut a piece of wood at 45° to its sides is more meaningful than saying that we want to cut it at 0.79 radians. It's important to realize, however, that the angle θ in the definitions of the trigonometric functions is *always* in radians. Consequently, you must convert into radians any angles specified in degrees, with the formula

$$\text{radians} = (180 / \pi) \text{ degrees}$$

before using the angles in trigonometric functions. Angles often have to be specified for ray-tracing purposes. For example, the viewing angles for fish-eye and panoramic cameras in Chapter 11, the stereo separation angles in Chapter 12, and in Chapter 19, several part objects will be defined in terms of angles. The user interface in each case will use degrees, while the code inside will convert the degrees to radians.

2.4 Trigonometry

Because trigonometry is used so extensively in ray tracing, I'll present here some of its definitions and relevant formulae.

2.4.1 Definitions

Consider the right-angled triangle in Figure 2.2. The trigonometric functions sine, cosine, and tangent of the angle θ are defined as

$$\sin \theta = a / c,$$
$$\cos \theta = b / c,$$
$$\tan \theta = a / b = \sin \theta \ / \cos \theta.$$

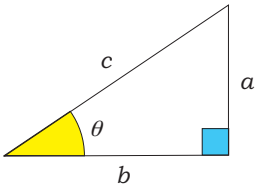


Figure 2.2. A right-angled triangle.

Pythagoras' theorem for the right-angled triangle is

$$c^2 = a^2 + b^2.$$

2.4.2 Relations

$$\begin{aligned} \sin^2\theta + \cos^2\theta &= 1, \\ \sin(\theta \pm \phi) &= \sin\theta\cos\phi \pm \cos\theta\sin\phi, \\ \cos(\theta \pm \phi) &= \cos\theta\cos\phi \mp \sin\theta\sin\phi. \end{aligned} \tag{2.1}$$

When $\phi = \pi/2$, Equations (2.1) become

$$\begin{aligned} \sin(\theta \pm \pi/2) &= \pm\cos\theta, \\ \cos(\theta \pm \pi/2) &= \mp\cos\theta. \end{aligned}$$

2.5 Coordinate Systems

Ray tracing uses the following 2D and 3D coordinate systems:

- world coordinates (3D);
- viewing coordinates (3D);
- object coordinates (2D and 3D);
- local shading coordinates (3D);
- view-plane coordinates (2D);
- texture coordinates (2D and 3D).

In later chapters, I'll discuss each of these coordinate systems and how to use them. Below, I'll just discuss the mathematical definitions of some common 3D coordinate systems in the way that we use them in ray tracing.

2.5.1 3D Cartesian Coordinates

Figure 2.3(a) illustrates 3D Cartesian coordinates, where we use an *ordered triple*[1] of real numbers (x, y, z) to specify the location of a point in 3D space. These coordinates are the perpendicular distances of the point along the three coordinate axes, measured from the origin. Cartesian coordinates are used to specify the location of points in infinite 3D space. Each pair of coordinate axes defines a *coordinate plane*, of which there are three: the (x, y) plane defined by the x- and y-axes, and the (x, z) and (y, z) planes.

1. An ordered triple is a sequence of three numbers in a definite order.

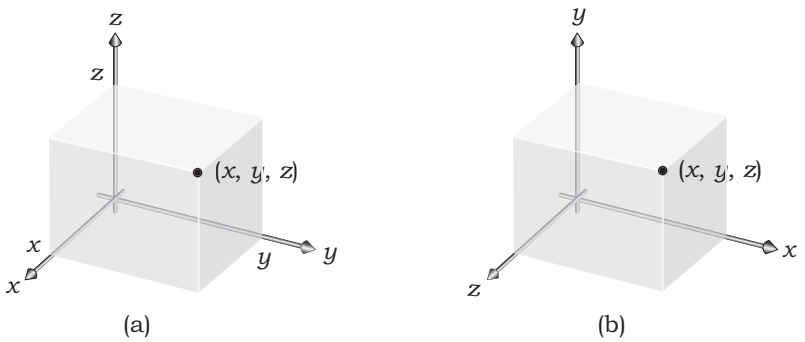


Figure 2.3. 3D Cartesian coordinates (a) as drawn in fields other than computer graphics; (b) as drawn in computer graphics.

If your background is in a field other than computing, for example, science or engineering, you are probably used to drawing the x-, y-, and z-axes as shown in Figure 2.3(a), with the z-axis pointing up. This is the convention in most fields, where the (x, y)-plane is horizontal. In 3D computer graphics, a different convention has been adopted for *world coordinates*, as shown in Figure 2.3(b), where the y-axis points up and the (x, z)-plane is horizontal.[2] It's important to realize that this is just a *drawing* convention; the coordinates in Figure 2.3(b) are still the same as in part (a).

Computer graphics also uses *right-handed* and *left-handed* coordinate systems, but I'll only use right-handed systems, as illustrated in Figure 2.3. The right-handedness of a coordinate system or set of basis vectors will come up later on, starting with orthonormal bases and frames in Section 2.12.

2.5.2 Cylindrical Coordinates

Cylindrical coordinates are based on Cartesian coordinates, but instead of using the x-, y-, and z-coordinates directly, we use a straight-line distance r, an angle ϕ, and the y-coordinate (see Figure 2.4). Because I'll use cylindrical coordinates to define circular cylinders in Chapter 19 with a vertical central

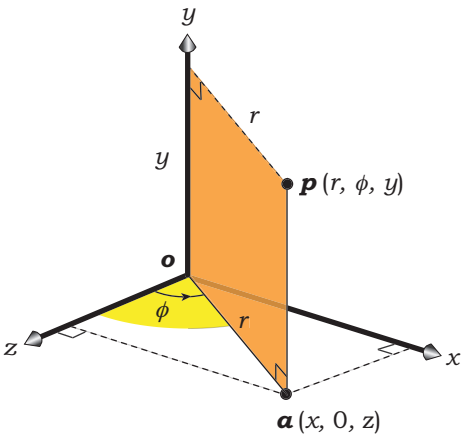


Figure 2.4. Definition of cylindrical coordinates.

2. This is also the convention adopted by the 3D rendering APIs OpenGL and RenderMan and the major 3D rendering and animation packages such as Maya, Houdini, and SoftImage.

axis in world coordinates, the Cartesian coordinates in Figure 2.4 also have the y-axis up.

The first coordinate r is the perpendicular distance between the point $\boldsymbol{p}$ (r, θ, ϕ) and the y-axis. Its value lies in the interval $r \in [0, \infty)$. To define the second coordinate ϕ, we first project $\boldsymbol{p}$ onto the (x, z) plane to get the point $\boldsymbol{a} = (x, 0, z)$. This also defines the orange vertical plane in Figure 2.4. The angle ϕ is defined to be the angle between the positive z-axis and the line $\boldsymbol{oa}$, measured counterclockwise in the (x, z) plane. Its value lies in the interval $\phi \in [0, 2\pi)$. This is known as the *azimuth angle*. The third coordinate is simply the y-coordinate.

With these definitions, it follows that the Cartesian coordinates of $\boldsymbol{p}$ can be expressed in terms of r, ϕ, and y as follows:

$$\begin{aligned} x &= r \sin \phi, \\ y &= y, \\ z &= r \cos \phi. \end{aligned} \tag{2.2}$$

2.5.3 Spherical Coordinates

Spherical coordinates are also based on Cartesian coordinates, but here, we specify locations with a distance r and two angles θ and ϕ, as shown in Figure 2.5. The coordinate r is now the straight-line distance from the origin $\boldsymbol{o}$ of the Cartesian coordinates to the point $\boldsymbol{p}$ (r, θ, ϕ). Again, its value lies in the interval $r \in [0, \infty)$. The second coordinate θ is the angle between the positive y-axis and the line $\boldsymbol{op}$. It's measured from the y-axis in the vertical plane defined by $(\boldsymbol{o}, \boldsymbol{p}, \boldsymbol{a})$, where $\boldsymbol{a}$ is again the projection of $\boldsymbol{p}$ onto the (x, z) plane, and its value lies in the interval $\theta \in [0, \pi]$. This is known as the *polar angle*. The third coordinate ϕ is the same azimuth angle as in cylindrical coordinates.

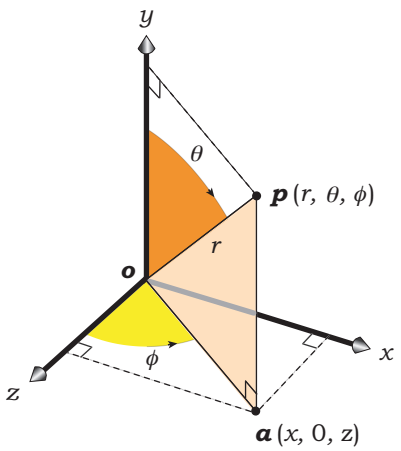


Figure 2.5. Definition of spherical coordinates.

With these definitions, it follows that the Cartesian coordinates of $\boldsymbol{p}$ can be expressed in terms of r, θ, and ϕ as follows:

$$\begin{aligned} x &= r \sin \theta \sin \phi, \\ y &= r \cos \theta, \\ z &= r \sin \theta \cos \phi. \end{aligned} \tag{2.3}$$

Spherical coordinates have a number of uses in ray tracing. Examples include the distribution of sample points on a hemisphere (Chapter 7), which has numerous shading applications, integration over a solid angle

(Section 2.10), and an intimate involvement with the theoretical foundations of ray tracing (Chapter 13).

2.6 Vectors

2.6.1 Definition and Representation

I'll present here some basic information about vectors, because we need this before I discuss points and normals in the following two sections. Vectors are used to represent many things in ray tracing, including ray directions, directions from hit points to light sources, reflection models, and orthonormal bases.

A vector is a *directed line segment*, as Figure 2.6 illustrates in 2D. A vector is defined by its length and direction, but not by its location in space. All vectors in Figure 2.6(a) are the same, while those in Figure 2.6(b) are all different, although they have the same lengths. We can represent a 3D vector by three floating-point numbers that are its projections onto the Cartesian (x, y, z) coordinate axes. See Figure 2.6(c) for a 2D representation, where the vector starts at the origin. These are the *components* of the vector, which are independent of the vector's location.

We denote a vector by a bold italic letter, for example, $\boldsymbol{u}$, and use the same letter for its components: u_x, u_y, and u_z. The *magnitude* of a vector is the length of its line segment, and is denoted by $\|\boldsymbol{u}\|$.

Vectors are represented by the class `Vector3D`, which stores the components as three doubles: `x`, `y`, and `z`.

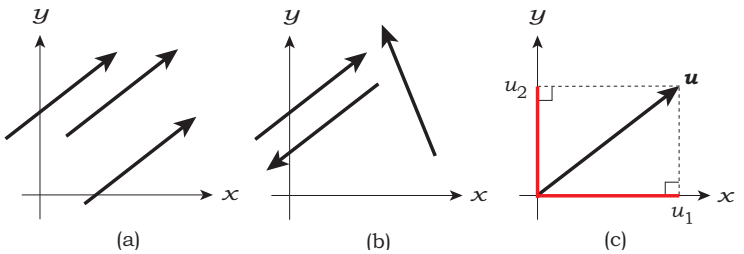


Figure 2.6. (a) Identical vectors defined by the same directed line segment; (b) different vectors defined by different line segments; (c) a vector's components are its projections (red) onto the coordinate axes.

2.6.2 Operations

Let

$$\boldsymbol{u} = (u_x, u_y, u_z),$$
$$\boldsymbol{v} = (v_x, v_y, v_z),$$

be two vectors. We need the following operations:

Operation	Definition	Return Type
$\boldsymbol{u} + \boldsymbol{v}$	$(u_x + v_x, u_y + v_y, u_z + v_z)$	vector
$\boldsymbol{u} - \boldsymbol{v}$	$(u_x - v_x, u_y - v_y, u_z - v_z)$	vector
$a\boldsymbol{u}$	(au_x, au_y, au_z)	vector
$\boldsymbol{u}a$	(au_x, au_y, au_z)	vector
$\boldsymbol{u} / a$	$(u_x/a, u_y/a, u_z/a)$	vector
$\boldsymbol{u} = \boldsymbol{v}$	(v_x, v_y, v_z)	vector reference
$\|\boldsymbol{u}\|$	$(u_x^2 + u_y^2 + u_z^2)^{1/2}$	double
$\|\boldsymbol{u}\|^2$	$u_x^2 + u_y^2 + u_z^2$	double

where a is a double. We also need the dot and cross products of two vectors, defined by

$\boldsymbol{u} \bullet \boldsymbol{v}$	$u_x v_x + u_y v_y + u_z v_z$	double	(2.4)
$\boldsymbol{u} \times \boldsymbol{v}$	$(u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x)$	vector	

Other useful operations are

$-\boldsymbol{u}$	$(-u_x, -u_y, -u_z)$	vector
$\boldsymbol{u}$ += $\boldsymbol{v}$	$(u_x + v_x, u_y + v_y, u_z + v_z)$	vector reference

The dot product in Equation (2.4) can also be written as

$$\boldsymbol{u} \bullet \boldsymbol{v} = \|\boldsymbol{u}\| \, \|\boldsymbol{v}\| \cos\theta,$$

where θ is the angle between $\boldsymbol{u}$ and $\boldsymbol{v}$. Although we won't use this to compute the value of $\boldsymbol{u} \bullet \boldsymbol{v}$, it's valuable for mathematical calculations. For example, if $\boldsymbol{u}$ and $\boldsymbol{u} \bullet \boldsymbol{v}$ are perpendicular, $\theta = \pi/2$, and $\boldsymbol{u} \bullet \boldsymbol{v} = 0$, since $\cos \pi/2 = 0$. I'll use this fact to write down Equation (2.6) for a plane in Section 2.9.1

We can use C++ operator overloading for many of the above operations, including *, which can be multiply overloaded for $\boldsymbol{u} \bullet \boldsymbol{v}$, $a\boldsymbol{u}$, and $\boldsymbol{u}a$. I use the operator ^ for the cross product, because the *outer product* of two n-dimensional vectors is denoted by $\boldsymbol{u} \wedge \boldsymbol{v}$, and this reduces to the cross product in 3D.

We also need to be able to *normalize* the vector. This converts it into a *unit vector*, which has length one. This is an important operation, as nearly all of the vectors used in ray tracing must be normalized at some stage. There are a couple of ways this can be written. One way doesn't return a value and has to be called in its own statement, such as `u.normalize();`, while the other returns a vector and can be called in expressions such as `u * v.hat()`. The second function is called `hat` because the common mathematical notation for unit vectors is to write them as $\hat{\boldsymbol{u}}$.

We must also be able to multiply a vector by a 4×4 matrix m on the left, to produce a new vector: $\boldsymbol{u}' = m\boldsymbol{u}$. Matrices are used to implement *affine transformations* on vectors, points, and normals, each of which transforms in a different way. For example, translation doesn't affect vectors. The details are in Chapters 20 and 21.

2.7 Points

Points represent locations in space, as indicated in Figure 2.7 for the point $\boldsymbol{a}$. Although each 3D point has a *location vector* with components (x, y, z) associated with it, and can be represented in the same way as vectors, points and vectors are not the same. For example, the dot and cross products don't make sense for points, neither does the magnitude of a point, and we can't add points. We can, however, add a vector to a point, which gives a new point displaced from the original by the components of the vector (see Figure 2.7(a)). We can also subtract points, because the result is the vector that joins them, as Figure 2.7(b) illustrates. The distance between two points makes sense, but it doesn't make sense for vectors.

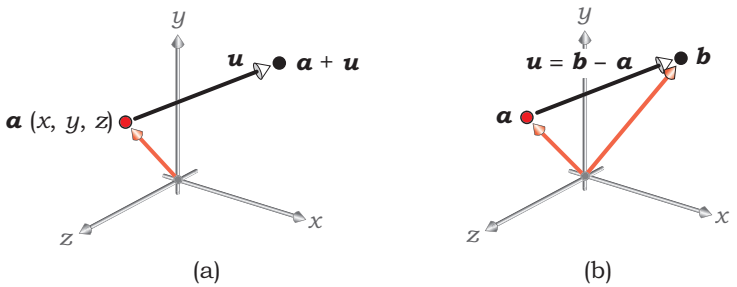


Figure 2.7. Each point has a location vector associated with it, as indicated by the red arrows. (a) Adding the vector $\boldsymbol{u}$ to the point $\boldsymbol{a}$ defines the new point $\boldsymbol{b} = \boldsymbol{a} + \boldsymbol{u}$; (b) subtracting $\boldsymbol{b}$ from $\boldsymbol{a}$ defines the vector $\boldsymbol{b} - \boldsymbol{a}$ that joins them.

Let

$$\boldsymbol{a} = (a_x, a_y, a_z)$$
$$\boldsymbol{b} = (b_x, b_y, b_z),$$

be two points, $\boldsymbol{u} = (u_1, u_2, u_3)$ be a vector, and c be a double. We need the following operations:

Operation	Definition	Return Type
$\boldsymbol{a} + \boldsymbol{u}$	$(a_x + u_x, a_y + u_y, a_z + u_z)$	point
$\boldsymbol{a} - \boldsymbol{u}$	$(a_x - u_x, a_y - u_y, a_z - u_z)$	point
$\boldsymbol{a} - \boldsymbol{b}$	$(a_x - b_x, a_y - b_y, a_z - b_z)$	vector
$\|\boldsymbol{a} - \boldsymbol{b}\|^2$	$(a_x - b_x)^2 + (a_y - b_y)^2 + (a_z - b_z)^2$	double
$\|\boldsymbol{a} - \boldsymbol{b}\|$	$[(a_x - b_x)^2 + (a_y - b_y)^2 + (a_z - b_z)^2]^{1/2}$	double
$\boldsymbol{a} = \boldsymbol{b}$	(b_x, b_y, b_z)	point reference
$c\boldsymbol{a}$	(ca_x, ca_y, ca_z)	double
$\boldsymbol{a}c$	(ca_x, ca_y, ca_z)	double

We also need a function that multiplies a point by a 4 × 4 matrix m on the left to return a new point, $\boldsymbol{a}' = m\boldsymbol{a}$. Finally, we need functions that return the distance and the square of the distance between two points.

2.8 Normals

Normals are also directed line segments and are therefore like vectors, but they behave differently. Because normals are always perpendicular to object surfaces, they must remain perpendicular when the objects are transformed. See Figure 2.8, which shows a sphere that's scaled to become an ellipsoid. This

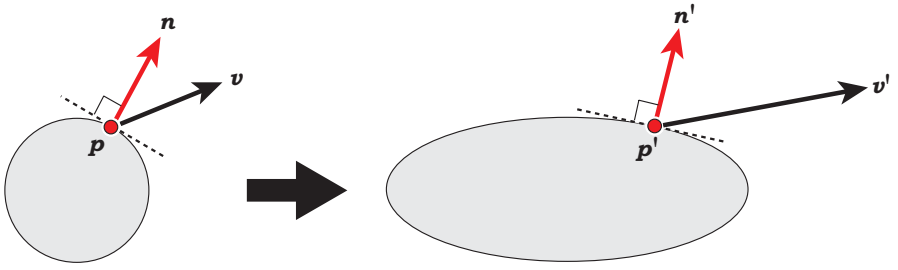


Figure 2.8. A vector, a point, and a normal are transformed differently when an object is transformed.

constraint is the reason that normals transform differently from vectors and points under affine transformations.

Because smooth surfaces have only a single (outward-pointing) normal at each point, it doesn't make sense to take the dot or cross products of normals with themselves, or subtract them. We also can't add a normal and a point. We must, however, be able to add normals, take the dot product of a normal and a vector, in either order, multiply a normal on the left or right by a scalar, and add a vector and a normal to give a vector. We must also be able to normalize a normal, and, you guessed it, multiply a normal on the left by a matrix to produce a transformed normal.

Consider two normals $\boldsymbol{n} = (n_x, n_y, n_z)$, $\boldsymbol{m} = (m_x, m_y, m_z)$, and a vector $\boldsymbol{u} = (u_x, u_y, u_z)$. We need the following operations:

Operation	Definition	Return Type
$-\boldsymbol{n}$	$(-n_x, -n_y, -n_z)$	normal
$\boldsymbol{n} + \boldsymbol{m}$	$(n_x + m_x, n_y + m_y, n_z + m_z)$	normal
$\boldsymbol{n} \bullet \boldsymbol{u}$	$n_x u_x + n_y u_y + n_z u_z$	double
$\boldsymbol{u} \bullet \boldsymbol{n}$	$n_x u_x + n_y u_y + n_z u_z$	double
$a\boldsymbol{n}$	(an_x, an_y, an_z)	normal
$\boldsymbol{n}a$	(an_x, an_y, an_z)	normal
$\boldsymbol{n} + \boldsymbol{u}$	$(n_x + u_x, n_y + u_y, n_z + u_z)$	vector
$\boldsymbol{u} + \boldsymbol{n}$	$(u_x + n_x, u_y + n_y, u_z + n_z)$	vector
$\boldsymbol{n} = \boldsymbol{m}$	(m_x, m_y, m_z)	normal reference
$\boldsymbol{n} \mathrel{+}= \boldsymbol{m}$	$(n_x + m_x, n_y + m_y, n_z + m_z)$	normal reference

At various places, we will have to make assignments between vectors, points, and normals. There are two ways that these can be programmed. The proper way is to use constructors, for example, `v = Vector(n)`, where a vector is constructed from a normal. For simplicity and efficiency, however, I've used straight assignment functions, where the above example is just `v = n`. Technically, this is bad programming practice, and the assignment operators can't test for self assignment.

The book's website contains the complete code for the classes `Vector3D`, `Point3D`, `Normal`, and `Matrix`. As these are important utility classes, I would rather you spent your time creating nice images, than implementing these. You will, however, have to implement a class that represents 2D points, as these are used in Chapters 5–7 to store sample points. This is considerably simpler than the `Point3D` class because the code doesn't involve vectors, normals, or matrices.

2.9 Mathematical Surfaces

We use mathematical definitions of surfaces to define the objects we ray trace. There are two basic ways of doing this: with implicit equations (which result in implicit surfaces) and with parametric equations (which result in parametric surfaces).

The objects we ray trace directly are all defined by implicit surfaces, because simple implicit surfaces are easy to ray trace (see Chapters 3 and 19). In contrast, even simple parametric surfaces are *difficult* to ray trace, because there's no easy way to calculate where a ray hits a parametric surface. Parametric surfaces are, however, an essential modeling tool in commercial rendering and animation packages (see the Further Reading section). For rendering purposes, the software packages convert the parametric surfaces to triangle meshes, a process known as *polygonization* or *tessellation*. Fortunately we *can* ray trace triangles and triangle meshes, as I'll discuss in Chapters 19 and 23.

All of the objects we ray trace can also be expressed as parametric surfaces, which is fortunate, because these have a number of uses that include ray tracing part objects and texture mapping.

2.9.1 Implicit Surfaces

An implicit surface is defined by an equation of the form

$$f(x, y, z) = 0, \tag{2.5}$$

where f is some arbitrary function of x, y, and z. We can express an implicit surface using set notation as $\{(x, y, z) : f(x, y, z) = 0\}$.

Implicit surfaces divide 3D space into two regions, where $f(x, y, z) < 0$ on one side of the surface, $f(x, y, z) > 0$ on the other side, and $f(x, y, z) = 0$ on the surface. Implicit surfaces are either *open* or *closed*, where an open surface extends to infinity, while a closed surface is finite in extent and has an inside and an outside. For example, planes and hyperboloids are open, while spheres and tori are closed. Implicit surfaces can also consist of a number of disconnected pieces. Figure 2.9 shows a cross section of a closed surface with the inside shaded. The inside can, however, be the region $f(x, y, z) < 0$ or $f(x, y, z) > 0$, depending on how the surface is defined.

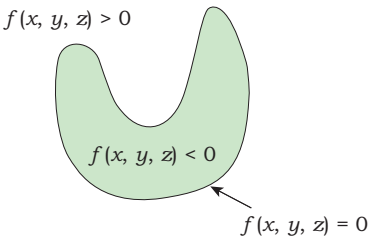


Figure 2.9. A closed implicit surface divides space into regions that are inside and outside the surface.

I'll illustrate implicit surfaces with a couple of simple examples that we'll ray trace in the following chapter.

Planes. A plane is an infinite flat sheet and is the simplest open surface. We define a plane by specifying a point $\boldsymbol{a}$ that lies on the plane, and a normal $\boldsymbol{n}$ to the plane. This defines the plane uniquely, because there is only one plane that passes through a given point and has the orientation specified by the normal. Since a plane is flat, all points on the surface have the same normal. Figure 2.10 shows a plane defined this way, where $\boldsymbol{p}$ is an arbitrary point on the plane, and I have only drawn the part of the plane that's in the first octant of the world coordinates (infinite planes are difficult to draw!).

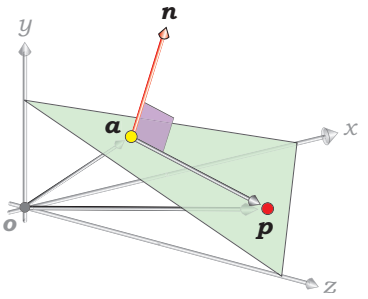


Figure 2.10. A plane is defined by a point $\boldsymbol{a}$, which determines where it is, and a normal $\boldsymbol{n}$, which determines its orientation.

The vector from $\boldsymbol{a}$ to $\boldsymbol{p}$ is $\boldsymbol{p} - \boldsymbol{a}$, and since this lies in the plane, it's perpendicular to the normal $\boldsymbol{n}$. We can therefore express the equation of a plane in terms of the dot product of $\boldsymbol{p} - \boldsymbol{a}$ and $\boldsymbol{n}$ as

$$(\boldsymbol{p} - \boldsymbol{a}) \bullet \boldsymbol{n} = 0. \tag{2.6}$$

If we write the points and normal in Equation (2.6) in component form ($\boldsymbol{p} = (x, y, z)$, $\boldsymbol{a} = (a_x, a_y, a_z)$, and $\boldsymbol{n} = (n_x, n_y, n_z)$) and use the component expression (2.4) for the dot product, we can express this equation as

$$Ax + By + Cz + D = 0. \tag{2.7}$$

This is the implicit equation of an arbitrary plane. Here, the coefficients $A = n_x$, $B = n_y$, and $C = n_z$ are the components of the normal, and $D = -\boldsymbol{a} \bullet \boldsymbol{n} = -a_x n_x - a_y n_y - a_z n_z$. An example is the (x, z) plane, whose implicit equation is as simple as we can get: $y = 0$.

Spheres. A sphere is the set of points that's within a constant specified distance r from a given point $\boldsymbol{c}$. Formally, a sphere $= \{\boldsymbol{p} : |\boldsymbol{p} - \boldsymbol{c}| \le r\}$, where $\boldsymbol{c}$ is the center of the sphere, and r is the radius (see Figure 2.11). The surface of the sphere, which is the part we're interested in for ray tracing, is defined by $\{\boldsymbol{p} : |\boldsymbol{p} - \boldsymbol{c}| = r\}$; that is, it's the set of points at distance r from $\boldsymbol{c}$. Technically, this is a *spherical shell*, but I'll refer to it as a sphere.

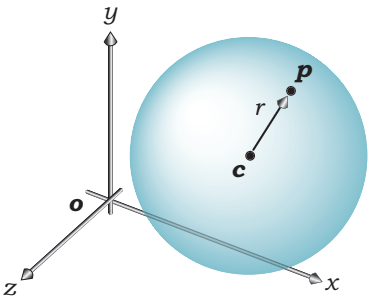


Figure 2.11. Sphere definition. The point $\boldsymbol{p}$ is on the surface of the sphere.

If $\boldsymbol{c} = (c_x, c_y, c_z)$, and $\boldsymbol{p} = (x, y, z)$ is a point on the surface of the sphere, the implicit equation of the surface can be written by inspection, since the square of the distance between $\boldsymbol{p}$ and

$\boldsymbol{c}$ is equal to the square of the radius. The implicit equation is therefore

$$(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 - r^2 = 0, \tag{2.8}$$

using Pythagoras' theorem in 3D. An example is a unit sphere (radius $r = 1$) centered on the origin, for which Equation (2.8) simplifies to

$$x^2 + y^2 + z^2 - 1 = 0.$$

2.9.2 Parametric Surfaces

A point on a parametric surface is expressed in the form

$$\boldsymbol{p}\,(u, v) = [f\,(u, v), g\,(u, v), h\,(u, v)], \tag{2.9}$$

where $f\,(u, v)$, $g\,(u, v)$, and $h\,(u, v)$ are explicit functions of the two parameters u and v. Since these three functions are really the (x, y, z) coordinates of $\boldsymbol{p}$, we usually write Equation (2.9) in the form

$$\boldsymbol{p}\,(u, v) = [x\,(u, v), y\,(u, v), z\,(u, v)].$$

For given values of u and v, the expressions for x, y, and z can be evaluated to give the location of $\boldsymbol{p}$. In modeling for graphics applications, including computer-aided design, x, y, and z are usually polynomials in u and v, or the ratio of two polynomials. The parameters are also restricted to a certain range, typically $(u, v) \in [0, 1] \times [0, 1]$. This defines a *parametric surface patch*, which is finite in extent (see the Further Reading section).

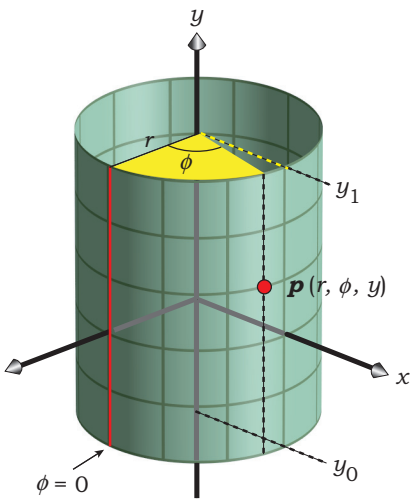


Figure 2.12. A circular cylinder centered on the y-axis with radius r and finite extent in the y-direction.

I'll illustrate parametric surfaces with circular cylinders and spheres. Why not planes? As it turns out, these are more complex to write in parametric form than cylinders and spheres, and since we only need their parametric representation to ray trace triangles, I'll discuss that in Chapter 19.

Circular cylinders. A circular cylinder centered on the y-axis with radius r and finite extent in the y-direction is defined by $\{(x, y, z): x^2 + z^2 = r^2 \text{ and } y \in [y_0, y_1]\}$, as Figure 2.12 illustrates.

The parametric representation of the cylinder is the same as Equations (2.1) for cylindrical coordinates,

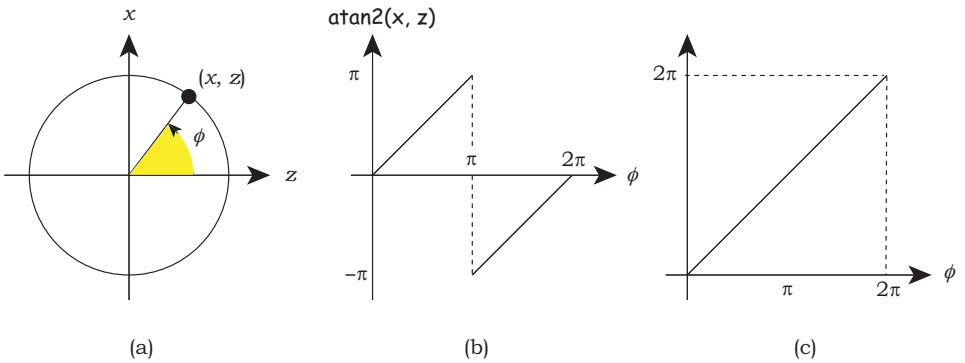


Figure 2.13. (a) Point (x, z) that defines an angle ϕ; (b) plot of `atan2(x, z)`, which is discontinuous at $\phi = \pi$; (c) plot of adjusted function that returns an angle $\phi \in [0, 2\pi]$.

where r is the specified radius of the cylinder and y is confined to the interval $[y_0, y_1]$. The parameters are ϕ and y, as these vary over the surface. We need to calculate the (ϕ, y)-values from the (x, y, z)-coordinates of a point on the cylinder. As there's nothing to calculate for y, that only leaves ϕ to deal with. From Equation (2.1), we have

$$\phi = \tan^{-1}(x / z). \tag{2.10}$$

The Standard C Library function `double atan2(double x, double z)` is the most convenient way to compute ϕ, because it returns the angle whose tangent is x / z, in the full 2π angular range $[-\pi, +\pi]$ radians. However, we need an angle in the range $[0, 2\pi]$, not $[-\pi, +\pi]$. Imagine a point with coordinates (x, z) that moves around a circle centered on the origin, as in Figure 2.13(a). Figure 2.13(b) shows the value of `atan2(x, z)` as a function of the angle ϕ. When $x \geq 0$, `atan2` $= \phi$, but when $x < 0$, `atan2` $= \phi - 2\pi$, to give the angular range $[-\pi, +\pi]$. Since we want an angle in the interval $[0, 2\pi]$, we have to check if `atan2(x, z)` < 0, and when that's true, add 2π to the value. This gives the graph in Figure 2.13(c).

This can be coded as in Listing 2.1.

```
double phi = atan2(x, z);
if (phi < 0.0)
     phi += TWO_PI;
```

Listing 2.1. Code to evaluate ϕ.

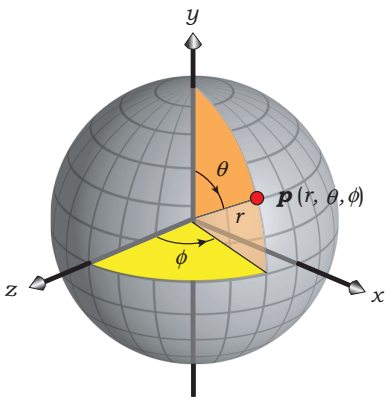


Figure 2.14. A sphere of radius r centered at the world origin.

Spheres. I'll only consider the parametric representation of spheres that are centered on the world origin. Although it's simple to generalize the parametric equations to spheres with arbitrary centers, I'll use *affine transformations* (Chapters 20 and 21) to construct parametric spheres that are not centered at the origin.

The parametric equations of a sphere are the same as Equations (2.2) for spherical coordinates, where r is now the radius of the sphere, and the two spherical-coordinate angles θ and ϕ are the parameters. These are illustrated in Figure 2.14. Note that θ is measured from the top of the sphere.

Given a point $\boldsymbol{p}\ (x, y, z)$ on the surface of the sphere, we need to calculate θ and ϕ. The calculation of ϕ is the same as it is for cylinders, and θ is simple to calculate. It follows from Equation (2.3) that

$$\theta = \cos^{-1}(y/r).$$

We can use the Standard C library function `double acos(double x)` to compute θ, because this returns $\theta \in [0, \pi]$ for $x = y / r \in [-1, +1]$.

2.9.3 Tangent Planes

At every point $\boldsymbol{p}$ on a surface where we can define the normal, we can also define a *tangent plane*. This is a plane that's perpendicular to the normal and just touches the surface at the given point (see Figure 2.15.) Although we won't have to compute the equation of the tangent plane, or use it in the ray-tracer code, I'll often refer to it in the shading chapters.

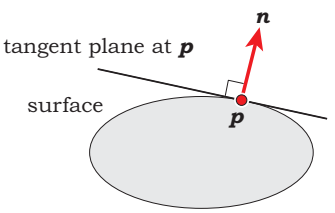


Figure 2.15. Cross section of a tangent plane at a surface point $\boldsymbol{p}$.

2.10 Solid Angles

2.10.1 Definition

Solid angles are the 2D generalization of 1D angles. To see how these are defined, let's consider an object that's visible from a point $\boldsymbol{p}$, as shown in Figure 2.16(a). First, place a sphere around the point and centered on it. Next, draw

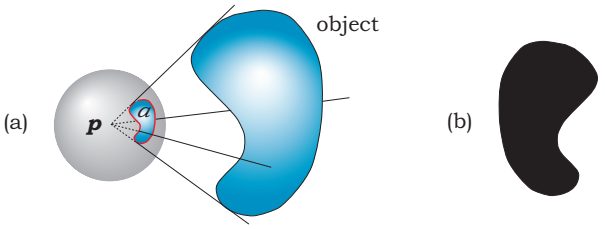


Figure 2.16. (a) Definition of solid angle; (b) silhouette of the object as seen from $\boldsymbol{p}$.

a line from $\boldsymbol{p}$ to some point on the outline of the object as seen from $\boldsymbol{p}$. This outline is the edge of the object's silhouette as seen from $\boldsymbol{p}$ (see Figure 2.16(b)). Finally, run the line completely around the outline and draw the curve (red) traced on the surface of the sphere where the line intersects it. This curve will enclose a certain area a (cyan) on the sphere's surface. The *ratio* of this area to the total surface area of the sphere is the *solid angle* subtended at $\boldsymbol{p}$ by the object. Solid angles are measured in *steradians* and are usually denoted by ω.

If the sphere has radius 1, its surface area is 4π, and the solid angle is $\omega = a / 4\pi$ steradians. This definition does not depend on the radius of the sphere. The maximum value of a solid angle is 4π steradians, which is the solid angle subtended at a point by an object that completely surrounds it.

Figure 2.17 shows a differential surface element dA, oriented so that its normal makes an angle θ with the line that joins it and the center of a unit sphere. If dA is distance d from the center of the sphere, its differential solid angle $d\omega$ is given by

$$d\omega = \frac{\cos\theta \, dA}{d^2}. \tag{2.11}$$

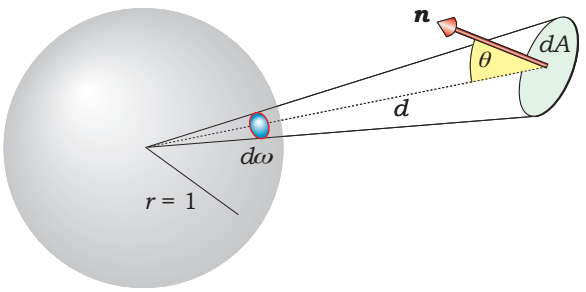


Figure 2.17. A differential surface element dA and its differential solid angle $d\omega$.

2.10.2 Solid Angles in Spherical Coordinates

The radiometric quantities and the definitions of reflectance in Chapter 13 require solid angles to be computed on the surface of a unit hemisphere. For this, we need to start with an expression for the differential solid angle $d\omega$ in spherical coordinates. Figure 2.18 shows how to use the definition of spherical coordinates to compute $d\omega$.

Calculating the second-order differential yields

$$d\omega = \sin\theta \; d\theta \; d\phi.$$

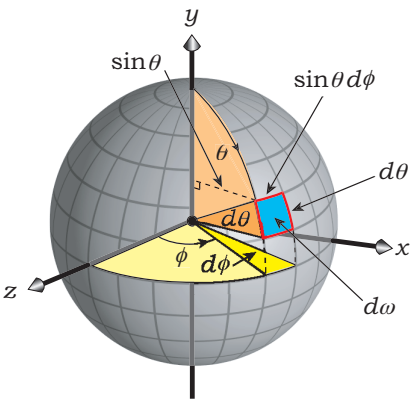


Figure 2.18. Differential solid angle $d\omega$ on the surface of a unit sphere.

A simple application is to compute the surface area of the unit sphere:

$$area = \int_0^{2\pi}\int_0^{\pi} \sin\theta \, d\theta \, d\phi = 4\pi,$$

as expected.

2.10.3 Integrals Over a Hemisphere

In later chapters, we'll need to use integrals of various functions $f(\theta, \phi)$ over the top hemisphere of the unit sphere: $(\theta, \phi) \in [0, \pi/2] \; [0, 2\pi]$. Although most of the integrands will be too complex to evaluate analytically, we can evaluate some of them exactly. The general integral is

$$I = \int_{\omega} f(\theta,\phi)\cos\theta \, d\omega,$$

where there are a few things to notice. First, it's written as a single integral over a solid angle, which is a shorthand notation for the double integral over θ and ϕ. Second, the ω on the integral sign denotes the solid-angle domain over which the integral is evaluated: $\omega \in [0, 2\pi]$ steradians, where $\omega = 2\pi$ is the whole hemisphere. Third, $\cos\theta$ is present in the integrand. This is a geometric factor that's present in all hemisphere integrands, as I'll explain in Chapter 13. The quantity $\cos\theta \, d\omega$ is known as the *projected solid angle*, because it's the projection of the differential solid angle $d\omega$ onto the (x, z) plane.

We will frequently need integrals with $f(\theta, \phi) = \cos^{n-1}\theta$, where n is an integer. In this case,

$$I = \int_{2\pi} \cos^n \theta \, d\omega = \int_0^{2\pi} \int_0^{\pi/2} \cos^n \theta \sin\theta \, d\theta \, d\phi.$$

Fortunately, these integrals can be evaluated exactly. A major factor is that the integrand is separable, allowing I to be written as the product of two 1D integrals:

$$I = \int_0^{2\pi} d\phi \int_0^{\pi/2} \cos^n \theta \sin\theta \, d\theta = 2\pi \int_0^{\pi/2} \cos^n \theta \sin\theta \, d\theta.$$

We can evaluate the θ integral with the change of variable $u = \cos\theta$, so that $du = -\sin\theta \, d\theta$, $u = 1$ when $\theta = 0$, and $u = 0$ when $\theta = \pi/2$. This gives

$$I = 2\pi \int_0^1 u^n du = \left[\frac{u^{n+1}}{n+1} \right]_0^1 = \frac{2\pi}{n+1}. \tag{2.12}$$

2.11 Random Numbers

Ray tracing makes extensive use of random numbers because many ray-tracing effects are based on them. Most of the sampling techniques covered in Chapters 5–7 are based on random numbers, and these are used for antialiasing, depth of field, ambient occlusion, area lights, global illumination, and glossy reflection. The noise-based textures in Chapter 33 are also based on random numbers. It's therefore worthwhile to discuss informally what they are and how we can generate them.

If we had a function that returned a true random number every time it was called, it would always return a different number, and there would be no way we could predict what the number was going to be. Because computers are deterministic devices, the common random-number-generation algorithms don't return true random numbers; instead, they return *pseudorandom numbers*, which we call PRNs. To see what this means, suppose we have a program that generates n PRNs. Each time we run the program, we'll get the same numbers, but that's actually what we want. Let's say you are modeling a scene that contains a noise-based texture such as marble. If the numbers were truly random, you would get a slightly different marble texture each time you ran the program. This could make it difficult to debug your program and design the exact scene you want. It would be even worse for animation, where each frame would show a different texture.

We use PRNs that are in the range [0, 1] and are *uniformly distributed* in that range. To see what this means, we need to look at the numbers from a

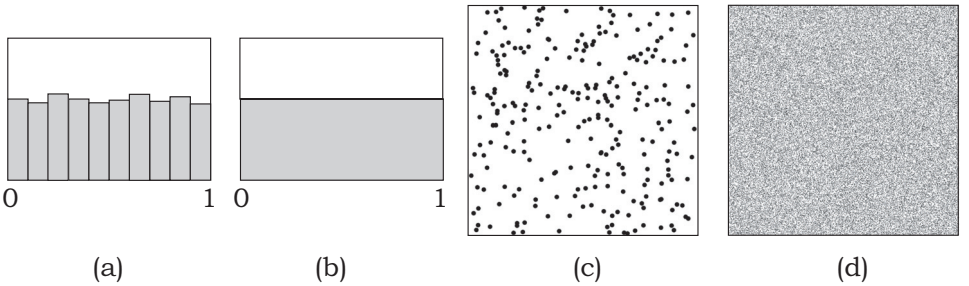


Figure 2.19. (a) Random numbers in 10 bins; (b) uniform distribution; (c) 256 random points; (d) 100,000 random points.

statistical point of view. Therefore, let's divide the interval [0, 1] into 10 subintervals [0.0, 0.1] ... [0.9, 1.0], called *bins*, and generate n =1000 PRNs. If these are uniformly distributed in [0, 1], there will be approximately 100 of them in each bin. Because the numbers won't be exactly the same, we will get a *frequency histogram*, something like that shown in Figure 2.19(a). Here, the height of each bar represents the number of PRNs in the bin. If we now increase n, the *fractional* difference between the numbers should decrease, and in the limit $n \to \infty$, this difference should approach zero. In this limit we could also have an infinite number of infinitely small bins, and the frequency histogram will look like Figure 2.19(b), a horizontal line. An exact uniform distribution only exists in the limit $n \to \infty$. Back in the real world, where we can only generate a finite number of PRNs, they will only be approximately uniformly distributed, as in Figure 2.19(a). This can create problems if we don't use enough of them, but that's best illustrated in two dimensions.

We'll often need to generate pairs of PRNs r_1 and r_2, where $(r_1, r_2) \in [0, 1] \times [0, 1]$. Figure 2.19(c) shows the locations of $256 = 16 \times 16$ random pairs in a unit square and illustrates a problem with using low numbers. Here, the distribution of points is not particularly uniform because we have only generated 16 PRNs in each direction. As a result, there are clumps and gaps in the distribution. Because we'll often use even lower numbers, for example, 4×4 or 5×5, we need a way of making the pairs more uniformly distributed but still random. In Chapter 5, I'll discuss a number of ways to achieve this. We could also solve the problem by using very large numbers, as in Figure 2.19(d), where there are 100,000 random pairs. This is a nice uniform distribution but, of course, completely impractical.

We can generate pseudorandom numbers with the Standard C library function `rand`, which returns an integer uniformly distributed in the interval [0, `RAND_MAX`]. Here, `RAND_MAX` is system-dependent. Depending on your sys-

```
inline int
rand_int(void) {
     return (rand());
}
```

Listing 2.2. The function `rand_int`.

```
inline float
rand_float(void) {
     return ((float)rand() / (float)RAND_MAX);
}
```

Listing 2.3. The function `rand_float`.

```
inline void
set_rand_seed(const int seed) {
     srand(seed);
}
```

Listing 2.4. The function `set_rand_seed`.

tem, there may be other random-number generators that you can use. To make it as simple as possible for you to use another random-number generator, I've wrapped `rand` inside the function `rand_int` as in Listing 2.2.

It's simple to convert the output of `rand` to floating-point numbers in the interval [0, 1] by dividing it by `RAND_MAX`. In fact, it's also convenient to define the wrapper function `rand_float`, as Listing 2.3 indicates.

Another convenient wrapper function is `set_rand_seed`, in Listing 2.4, which just calls `srand`. This has two essential uses, the first of which is for scene construction. For example, one of the scenes used in Chapter 11 has random boxes. If the build function didn't call `set_rand_seed` before constructing the boxes, their sizes, shapes, and colors would depend on how many samples were used for antialiasing. The reason is that the antialiasing samples also use `rand`. The second use is for setting up noise values for texture synthesis in Chapter 31, to guarantee that the textures are the same for every time that we render them.

As `rand_int`, `rand_float`, and `set_rand_seed` are the only functions that call `rand` and `srand`, it's easy to use another random-number generator. For example, if you want to use `random` in `rand_float`, it can return `(float)random() / (float) 0x80000000U`.

2.12 Orthonormal Bases and Frames

2.12.1 Definition

Definition 2.6. Three vectors $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$ form an *orthonormal basis* (ONB) if they are mutually perpendicular, they are all unit vectors, and they form a right-handed system where $\boldsymbol{w} = \boldsymbol{u} \times \boldsymbol{v}$.

The *ortho* in the name is short for orthogonal, and *normal* is there because the vectors are unit vectors. The word *basis* is there because when we have three non-parallel vectors that are not in the same plane, we can express any other 3D vector as a linear combination of them. Because it's most convenient to work with mutually orthogonal unit vectors, most of the common coordinate systems such as Cartesian and spherical coordinates have mutually orthogonal axes, and mutually orthogonal unit basis vectors.[3] In the case of Cartesian coordinates, the unit vectors ($\boldsymbol{i}$, $\boldsymbol{j}$, $\boldsymbol{k}$) form an orthonormal basis.

Orthonormal bases are an important construct in ray tracing because we use them whenever we need to set up a local coordinate system, a task we often have to do. Here's a list of the situations where we need an ONB:

- cameras (Chapters 9–12)
- ambient occlusion (Chapter 17)
- area-light shading (Chapter 18)
- rotation about an arbitrary line (Chapter 20)
- glossy reflection (Chapter 25)
- global illumination (Chapter 26)

2.12.2 Construction

We can construct an orthonormal basis from two arbitrary vectors, for example, the vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ in Figure 2.20(a). These don't have to be unit vectors or be orthogonal. If $\boldsymbol{a}$ and $\boldsymbol{b}$ are defined in world coordinates, $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$ will also be defined in world coordinates when we construct them. That's what we always need.

We can construct the orthonormal basis vectors in the order $\boldsymbol{w}$, $\boldsymbol{u}$, $\boldsymbol{v}$ by taking $\boldsymbol{w}$ to be parallel to $\boldsymbol{a}$ and then using cross products. First, make $\boldsymbol{w}$ a unit vector in the direction of $\boldsymbol{a}$:

$$\boldsymbol{w} = \boldsymbol{a} / \|\boldsymbol{a}\|,$$

3. Barycentric coordinates, which we use to ray trace triangles, are an exception: these are non-orthogonal.

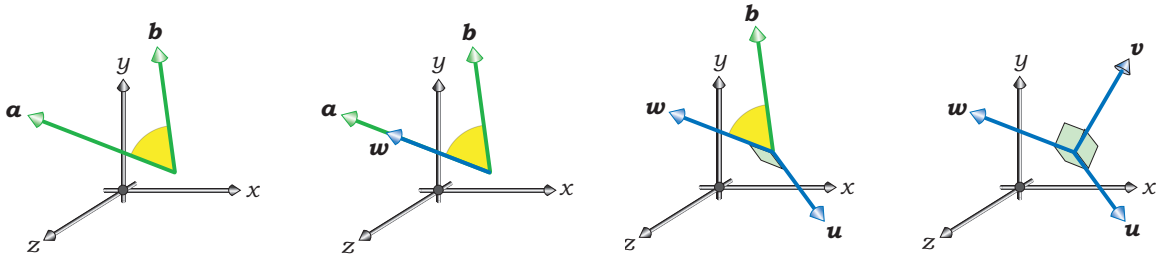


Figure 2.20. Construction of an orthonormal basis from two given vectors.

as in Figure 2.20(b). Next, construct $\boldsymbol{u}$ as the cross product of $\boldsymbol{b}$ and w, normalized to a unit vector:

$$\boldsymbol{u} = (\boldsymbol{b} \times w) / \|\boldsymbol{b} \times w\|,$$

as in Figure 2.02(c). Finally, construct v as the cross product of w and $\boldsymbol{u}$ to form a right-handed system:

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

as in Figure 2.20(d). This is a unit vector by construction.

Where do $\boldsymbol{a}$ and $\boldsymbol{b}$ come from? In practice, we only have a single predefined direction, for example, the view direction for a camera or the surface normal at a point being shaded. The normal, of course, isn't even a vector. Given $\boldsymbol{a}$, I arbitrarily use $\boldsymbol{b} = (0, 1, 0)$, which is vertically up in world coordinates and results in $\boldsymbol{u}$ being horizontal. There is, however, a potential problem when $\boldsymbol{a}$ is also vertical and therefore parallel to $\boldsymbol{b}$. This will be the situation if $\boldsymbol{a}$ is the normal to a horizontal surface. In this case, the construction above for $\boldsymbol{u}$ and v will fail. There are a couple of solutions, the simplest being to set $\boldsymbol{b}$ to a vector like

$$\boldsymbol{b} = (0.00424, 1, 0.00764),$$

as this is slightly offset from vertically up. This situation also arises when the camera is looking vertically up or down in world coordinates, but there, I'll use a different technique (see Chapter 9).

2.12.3 Orthonormal Frames

An orthonormal basis and a point $\boldsymbol{o}$ where the unit vectors meet is an *orthonormal frame*. This defines a coordinate system with $\boldsymbol{o}$ as the origin. A common

example consists of the ($\boldsymbol{i}$, $\boldsymbol{j}$, $\boldsymbol{k}$) basis vectors and the origin in 3D Cartesian coordinates. In shading applications, the origin of the ($\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$) frame is a ray-object hit point, and the orthonormal frame defines a local coordinate system centered on the hit point.

2.12.4 Using an Orthonormal Frame

Typically, we need to calculate a ray direction in a local coordinate system *and* specify the direction in world coordinates. This is because rays are always defined in world coordinates. Suppose we have calculated a ray direction $\boldsymbol{d} = (d_u, d_v, d_w)$ with respect to an orthonormal frame. We can express $\boldsymbol{d}$ as

$$\boldsymbol{d} = d_u\boldsymbol{u} + d_v\boldsymbol{v} + d_w\boldsymbol{w}. \tag{2.13}$$

In case you are puzzled by this formula, it's helpful to remember that Equation (2.13) is no different from writing an arbitrary vector $\boldsymbol{e}$ in Cartesian coordinates as

$$\boldsymbol{e} = e_x\boldsymbol{i} + e_y\boldsymbol{j} + e_z\boldsymbol{k}.$$

The critical thing to note about Equation (2.13) is that since ($\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$) are defined in world coordinates, it also expresses $\boldsymbol{d}$ in world coordinates.

Setting up an orthonormal frame only takes a few lines of code, and using it typically consists of calculating (d_u, d_v, d_w). I've included several examples of how to carry out this process, with sample code. The first is in Chapter 9 for a pinhole camera.

2.13 Geometric Series

A *geometric series* is a sum of n terms of the form

$$s_n = a + ar + ar^2 + \ldots + ar^{n-1}, \tag{2.14}$$

$$= \sum_{j=0}^{n-1} ar^n, \tag{2.15}$$

where, for our purposes, a and r are floating-point numbers. The number a is the *scale factor*, and r is the *ratio*, because the ratio of successive terms is always the same: $ar^n/ar^{n-1} = r$. If $r = 1$, the sum (2.15) is

$$s_n = an. \tag{2.16}$$

If $r \neq 1$, we can write the sum (2.15) as

$$s_n = \frac{a(1-r^n)}{1-r}. \tag{2.17}$$

An *infinite geometric series* is of the form (2.14), but where the number of terms is infinite. Provided $r \in (-1, +1)$, that is, $|r| < 1$, $r^n \to 0$ as $n \to \infty$, and Equation (2.17) becomes

$$s = \frac{a}{1-r}. \tag{2.18}$$

I'll use the expressions (2.15)–(2.18) in Chapters 28 and 31.

2.14 The Dirac Delta Function

The Dirac delta function $\delta(x)$ is defined as the function with the following properties:

$$\delta(x) = 0 \text{ if } x \neq 0,$$

$$\int_{-\infty}^{\infty} \delta(x)\, dx = 1,$$

$$\int_a^b \delta(x-c)\, f(x)\, dx = f(c), \text{ provided } c \in [a, b]. \tag{2.19}$$

This is not an ordinary function; it's zero everywhere except the origin, where it's infinite. Notice from Equation (2.19) that when an integrand contains a delta function, the integral collapses to $f(c)$. This happens with multi-dimensional integrals as well as one-dimensional integrals, a property that will make it simple for us to evaluate certain radiometric integrals in the shading chapters. When the distribution of incoming radiance at a surface point is confined to a single direction, it can be represented by a delta function (see Chapters 14, 24, and 27). I'll define radiometry and radiance in Chapter 13.

Notes and Discussion

Because vectors, normals, and points all store an ordered triple of floating-point numbers, you can represent all three in a single class as vectors, instead of using separate classes for each. Opinion in the ray-tracing community is divided as to which is the best approach. Certainly, using a single class is simpler and saves code, but it's incorrect modeling. With separate classes, the

application code shows explicitly the types of all of the relevant variables and is therefore more readable. My experience in the classroom is that it's also easier to teach affine transformations when there are separate classes.

Where should you put the functions `rand_int` and `rand_float` introduced in Section 2.9? As you develop your ray tracer, you will write or use numerous mathematical utility functions. It's most convenient to keep these in a separate compilation unit called, say, **`Maths.cpp`** or **`Utilities.cpp`**. You should put their prototypes in a corresponding header file that you can `#include` as required. These functions don't have to be class members, and the shorter ones can be inlined.

Further Reading

There are many excellent books on the mathematics discussed in this chapter, and the mathematics not discussed, such as calculus and matrices. Here are some examples. Vince and Morris (1990) discusses sets. The calculus text Thomas and Finney (1996) covers intervals, coordinate systems, vectors, geometric series, and of course, calculus. Anton (2004) covers systems of linear equations, matrices, determinants, and vectors. Mortenson (1999) is on mathematics for computer graphics and discusses vectors and points. Rogers (2001) is an introduction to parametric curves and surfaces with code. Hill and Kelley (2006) also discusses curves and surfaces. Bloomenthal (1997) is an introduction to implicit surfaces.

Questions

2.1. In Figure 2.16, why doesn't the solid angle subtended at $\boldsymbol{p}$ by the object depend on the radius of the sphere?

2.2. When we construct an orthonormal basis with the procedure described in Section 2.12.2, why is $\boldsymbol{v}$ a unit vector by construction?

2.3. Why does the construction of $\boldsymbol{u}$ and $\boldsymbol{v}$ in an orthonormal basis fail when $\boldsymbol{a}$ and $\boldsymbol{b}$ are parallel?

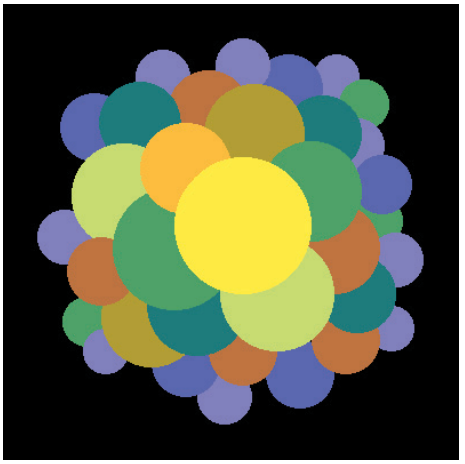
Exercises

2.1. Use Equations (2.4) and (2.6) to derive the plane equation (2.7).

2.2. Prove that when $\boldsymbol{b}$ is vertical, the orthonormal basis vector $\boldsymbol{u}$ is horizontal.

3 Bare-Bones Ray Tracing

3.1 How Ray Tracing Works
3.2 The World
3.3 Rays
3.4 Ray-Object Intersections
3.5 Representing Colors
3.6 A Bare-Bones Ray Tracer
3.7 Tracers
3.8 Color Display
3.9 Ray Tracing Multiple Objects



Objectives

By the end of this chapter, you should:

- understand how ray casting works;
- know how rays are defined;
- know how to intersect a ray with a plane and a sphere;
- understand the structure of a simple ray tracer;
- have implemented a ray tracer that can render orthographic views of an arbitrary number of planes and spheres.

The purpose of this chapter is to explain how a number of ray-tracing processes work in a simplified context. I'll discuss here how ray tracing generates images, how rays are defined, how ray-object intersections work, the classes required for a simple ray tracer, and how to ray trace an arbitrary number of spheres and planes. To make things as simple as possible, I've left out a lot of important processes such as antialiasing, perspective viewing with a pinhole camera, and shading. By doing this, the resulting ray tracer is as simple as possible, although, as you'll see, it still has a degree of complexity. This is also a long chapter because it covers a lot of material.

This chapter differs from the following chapters in that the skeleton ray tracer on the book's website does everything in the chapter. Hopefully, this will have you quickly ray tracing multiple planes and spheres.

3.1 How Ray Tracing Works

A simple ray tracer works by performing the following operations:

```
define some objects
specify a material for each object
define some light sources
define a window whose surface is covered with pixels

for each pixel
     shoot a ray towards the objects from the center
        of the pixel
     compute the nearest hit point of the ray with the
        objects (if any)

     if the ray hits an object
          use the object's material and the lights to
          compute the pixel color
     else
          set the pixel color to black
```

This is known as *ray casting*. Figure 3.1 illustrates some of the above processes for a sphere, a triangle, and a box, illuminated by a single light. The gray rectangles on the left are the pixels, and the white arrows are the rays, which start at the center of each pixel. Although there is one ray for each pixel, Figure 3.1 only shows a few rays. The red dots show where the rays hit the objects. The rays used in ray tracing differ from real light rays (or photons) in two ways. First, they travel in the opposite direction of real rays. This is best appreciated when we use a pinhole camera (Chapter 9), because there the rays start at the pinhole, which is infinitely small. If the rays started at the lights, none of them would pass through the pinhole, and we wouldn't have any images. Starting the rays from the camera (or pixels, in this chapter) is the only practical way to render images with ray tracing. The second difference is that we let the rays pass through the objects, even if they are opaque. We have to do this because the ray tracer needs to intersect each ray with each object to find the hit point that's closest to the start of the ray.

In Figure 3.1, one ray doesn't hit any objects, two rays hit one object, and one ray hits two objects. The hit points are always on the surfaces of the objects, which we treat as empty shells. As a result, rays hit the sphere and the box in two places and the triangle in one place.

The pixels are on a plane called the *view plane*, which is perpendicular to the rays. I'll sometimes refer to these as view-plane pixels. The rays are paral-

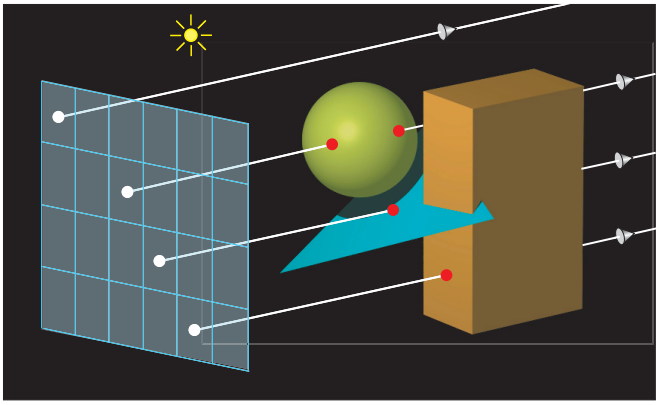


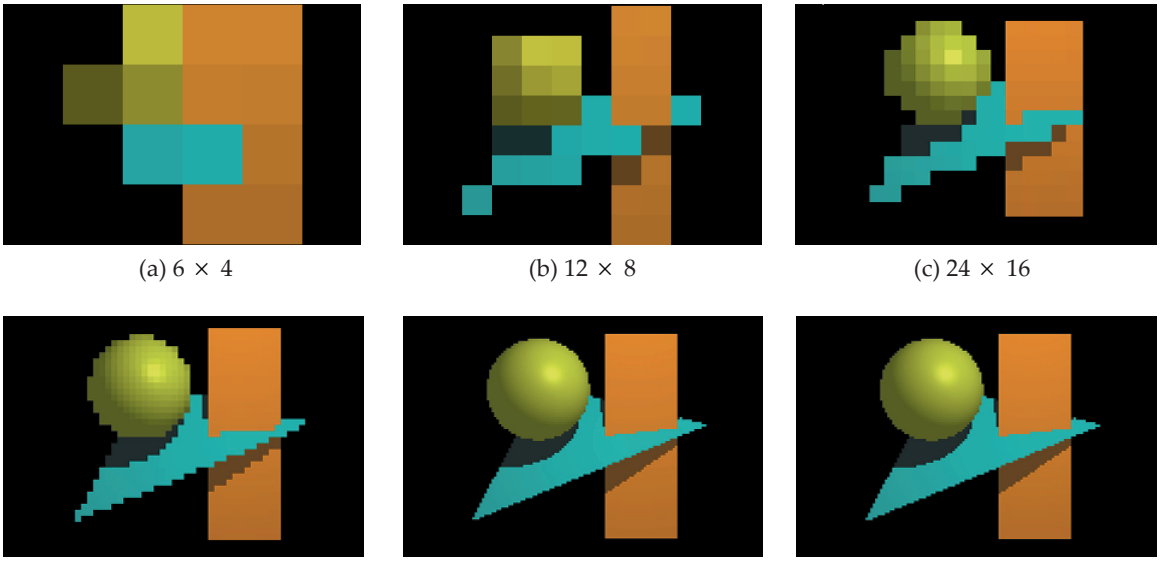
Figure 3.1. Rays shot from pixels into a scene that consists of three objects and a single light source.

lel to each other and produce an *orthographic projection* of the objects. When a ray hits an object, the color of its pixel is computed from the way the object's material reflects light, a process that's known as *shading*. Although the pixels on the view plane are just mathematical abstractions, like everything else in the ray tracer, each one is associated with a real pixel in a window on a computer screen. This is how you view the ray-traced image.

The process of working out where a ray hits an object is known as the *ray-object intersection calculation*. This is a fundamental process in ray tracing and usually takes most of the time. The intersection calculation is different for each type of object; some objects are easy to intersect, while others are difficult. All intersection calculations require some mathematics.

In Figure 3.1, there are 24 pixels arranged in four rows of six pixels, and in this case we say the image has a *pixel resolution* of 6 × 4. What would these objects look like if we ray traced them at this resolution? The result is in Figure 3.2(a), which gives no indication of what we are looking at. So, how can we get a meaningful image of these objects? The answer is simple: just increase the number of pixels. The other parts of Figure 3.2 show the objects ray traced at increasing pixel resolutions. With 24 × 16 pixels, we have some idea of what the objects are, and with 150 × 100 pixels, the image is quite good.

The above example was just to illustrate how ray tracing works with multiple objects, a light source, and shading. We'll start with something much simpler in Section 3.6: a single sphere with no lights, no material, and no shading, but first you need to learn how a basic ray tracer is organized and works.



(a) 6 × 4 (b) 12 × 8 (c) 24 × 16

(d) 60 × 40 (e) 120 × 80 (f) 150 × 100

Figure 3.2. The objects in Figure 3.1 ray traced at different pixel resolutions.

3.2 The World

Ray tracers render *scenes* that contain the geometric objects, lights, a camera, a view plane, a tracer, and a background color. In the ray tracer described in this book, these objects are all stored in a *world* object. For now, the world will only store the objects and view plane. The locations and orientations of all scene elements are specified in *world coordinates*, which is a 3D Cartesian coordinate system, as described in Chapter 2. I'll denote world coordinates by (x_w, y_w, z_w), or just (x, y, z) when the context is clear.

World coordinates are known as *absolute coordinates* because their origin and orientation are not defined, but that's not a problem. The only task of the ray tracer is to compute the color of each pixel, and the pixels are also defined in world coordinates. I'll discuss the `World` class in Section 3.6.

3.3 Rays

A ray is an infinite straight line that's defined by a point $\boldsymbol{o}$, called the *origin*, and a unit vector $\boldsymbol{d}$, called the *direction*. A ray is parametrized with the *ray param-*

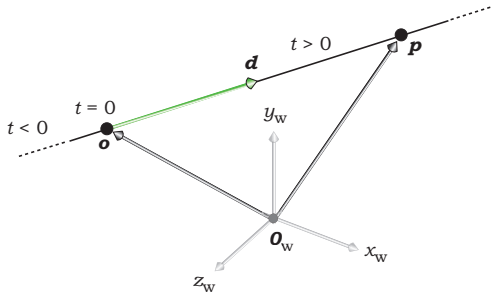


Figure 3.3. Ray definition in world coordinates.

eter t, where $t = 0$ at the ray origin, so that an arbitrary point $\boldsymbol{p}$ on a ray can be expressed as

$$\boldsymbol{p} = \boldsymbol{o} + t\,\boldsymbol{d}. \tag{3.1}$$

Figure 3.3 is a schematic diagram of a ray. The direction $\boldsymbol{d}$ defines an intrinsic direction for the ray along the line, where the value of the parameter t increases in the direction $\boldsymbol{d}$. Since $\boldsymbol{d}$ is a unit vector, t measures distance along the ray from the origin. Although we regard a ray as starting at its origin, we allow t to lie in the infinite interval $t \in (-\infty, +\infty)$ so that Equation (3.1) generates an infinite straight line. As we'll see later, it's essential to consider values of $t \in (-\infty, +\infty)$ in ray-object intersections. The origin and direction are always expressed in world coordinates before the ray is intersected with the objects.

Ray tracing uses the following types of rays:

- primary rays;
- secondary rays;
- shadow rays;
- light rays.

Primary rays start at the centers of the pixels for parallel viewing, and at the camera location for perspective viewing. Secondary rays are reflected and transmitted rays that start on object surfaces. Shadow rays are used for shading and start at object surfaces. Light rays start at the lights and are used to simulate certain aspects of global illumination, such as caustics. I'll only discuss primary rays in this chapter.

You should have a `Ray` class that stores the origin and direction, as in Listing 3.1. Because of their frequent use, all data members are public. In my shading architecture, there's no need to store the ray parameter in the ray. The code in Listing 3.1 will go in the header file **`Ray.h`**. Note the pre-compiler

```
#ifndef __RAY__
#define __RAY__

#include "Point3D.h"
#include "Vector3D.h"

class Ray {
     public:

          Point3D    o;                  // origin
          Vector3D   d;                  // direction

          Ray(void);                     // default constructor

          Ray(const Point3D& origin, const Vector3D& dir); // constructor

          Ray(const Ray& ray);           // copy constructor

          Ray&                           // assignment operator
          operator= (const Ray& rhs);

          ~Ray(void);                    // destructor
};

#endif
```

Listing 3.1. The **`Ray.h`** file.

directives `#ifndef __RAY__`, etc., to prevent multiple inclusion. These should go in every header file, but to save space, I won't quote them with other class declarations. You should also `#inlcude` header files for any classes that the current class requires: `Point.h` and `Vector3D` in this case. Again, to save space, I'll leave these out except for the `World` class in Section 3.6.1.

3.4 Ray-Object Intersections

3.4.1 General Points

The basic operation we perform with a ray is to intersect it with all geometric objects in the scene. This finds the nearest hit point, if any, along the ray from $\boldsymbol{o}$ in the direction $\boldsymbol{d}$. We look for the hit point with the smallest value of t in the interval $t \in [\varepsilon, +\infty)$ where ε is a small positive number, say $\varepsilon = 10^{-6}$. Why don't

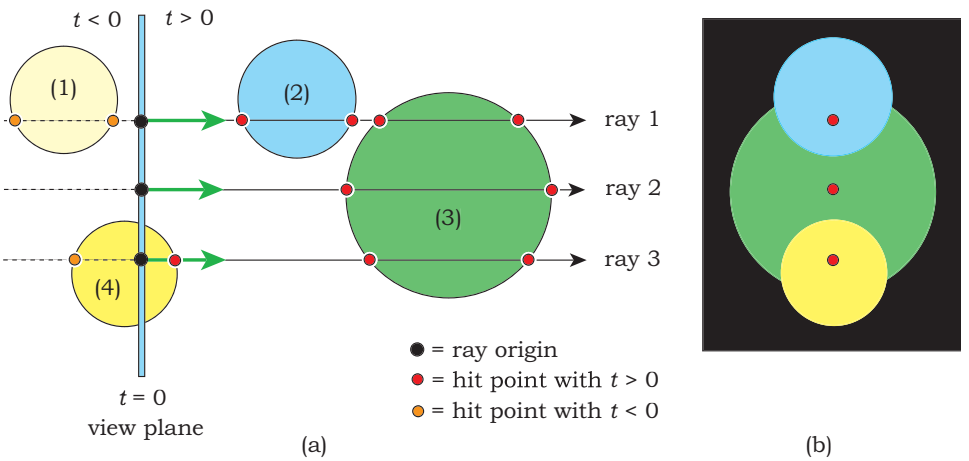


Figure 3.4. (a) Rays and their intersections with spheres; (b) ray-traced image of the spheres.

we use $\varepsilon = 0$? We could get away with this here, but it would create problems when we use shadows (Chapter 16), reflections (Chapters 24–26), and transparency (Chapters 27 and 28). I'll discuss what the problem is in Chapter 16. By using $\varepsilon > 0$ in this chapter, we won't have to change the plane and sphere hit functions in later chapters.

Because a ray origin can be anywhere in the scene, including inside objects and on their surfaces, ray-object hit points can occur for positive, negative, and zero values of t. This is why we need to treat rays as infinite straight lines instead of semi-infinite lines that start at $\boldsymbol{o}$. Figure 3.4(a) shows a number of spheres that are behind, straddling, and in front of the view plane, with three rays.

The spheres in Figure 3.4(a) will be rendered (or not) in the following ways:

- Sphere (1) is behind the origin of all rays that intersect it ($t < 0$) and will not appear in the image.
- Sphere (2) will be rendered with ray 1 and with all rays that hit it.
- Sphere (3) will only be rendered with rays like ray 2 that don't hit any other spheres.
- Sphere (4) will only be rendered with rays like ray 3 that start inside it.

Figure 3.4(b) shows how the spheres would look if we ray traced them with no shading and if their centers were all in the vertical plane that contains the three rays. The nearest hit points of these rays are indicated in the figure.

3.4.2 Rays and Implicit Surfaces

As I discussed in the previous chapter, the objects we ray trace are defined by implicit surfaces. A ray can hit an implicit surface any number of times, depending on how complex the surface is (see Figure 3.5). If the surface is *closed*, rays that start outside the surface will have an even number of hit points for $t > \varepsilon$ (rays 1 and 2). In principal, a ray that starts outside can have an odd number of hit points, including a single hit point, if it hits the surface tangentially (rays 3 and 5). In practice, this rarely happens (see Question 3.1). When the ray starts inside the surface, it will have an odd number of hit points for $t > \varepsilon$ (ray 4).

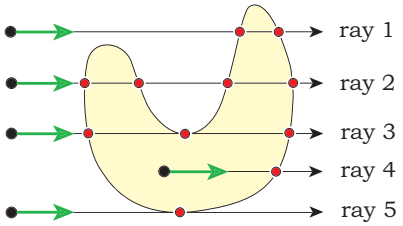


Figure 3.5. (a) Ray intersections with a closed implicit surface.

To intersect a ray with an implicit surface, we can re-write Equation (2.5), $f(x, y, z) = 0$, as

$$f(\boldsymbol{p}) = 0, \tag{3.2}$$

because the x, y, and z variables in $f(x, y, z)$ define a 3D point $\boldsymbol{p} = (x, y, z)$.

To find the hit points, we have to find the values of the ray parameter t that correspond to them. How do we do this? Here's the key point: *Hit points satisfy both the ray equation (3.1) and the implicit surface equation (3.2).* We can therefore substitute (3.1) into (3.2) to get

$$f(\boldsymbol{o} + t\,\boldsymbol{d}) = 0 \tag{3.3}$$

as the equation to solve for t. Since Equation (3.3) is symbolic, we can't do anything with it unless we specify $f(x, y, z)$. For a given ray and a given implicit surface, the only unknown in Equation (3.3) is t. After we have found the values of t, we substitute the smallest $t > \varepsilon$ value into Equation (3.1) to find the coordinates of the nearest hit point. If all this seems confusing, don't worry, because I'm going to illustrate it for planes and spheres. For these objects, we can solve Equation (3.3) exactly.

3.4.3 Geometric Objects

All geometric objects belong to an inheritance structure with class `GeometricObject` as the base class. This is by far the largest inheritance structure in the ray tracer, with approximately 40 objects, but in this chapter we'll use a simplified structure consisting of `GeometricObject`, `Plane`, and `Sphere`, as shown on the left of Figure 1.1.

```
class GeometricObject {
     public:
          ...

          virtual bool
          hit(const Ray& ray, double& tmin, ShadeRec& sr)
          const = 0;

     protected:

          RGBColor color;   // only used in this chapter
};
```

Listing 3.2. Partial declaration of the `GeometricObject` class.

Listing 3.2 shows part of the `GeometricObject` class declaration that stores an `RBGColor` for use in Section 3.6. I'll replace this with a material pointer when I discuss shading in Chapter 14. This listing also doesn't show other functions that

```
class ShadeRec {
     public:

          bool       hit_an_object;      // did the ray hit an object?
          Point3D    local_hit_point;    // world coordinates of hit point
          Normal     normal;             // normal at hit point
          RGBColor   color;              // used in Chapter 3 only
          World&     w;                  // world reference for shading

          ShadeRec(World& wr);           // constructor
          ShadeRec(const ShadeRec& sr);  // copy constructor
          ~ShadeRec(void);               // destructor

          ShadeRec&                      // assignment operator
          operator= (const ShadeRec& rhs);
};

ShadeRec::ShadeRec(World& wr)            // constructor
     :     hit_an_object(false),
          local_hit_point(),
          normal(),
          color(black),
          w(wr)
{}
```

Listing 3.3. Declaration of the `ShadeRec` class.

this class must have in order to operate as the base class of the geometric objects hierarchy, but it does show the declaration of the pure virtual function `hit`.

The `ShadeRec` object in the parameter list of the hit function is a utility class that stores all of the information that the ray tracer needs to *shade* a ray-object hit point. Briefly, shading is the process of computing the color that's reflected back along the ray, a process that most of this book is about. The `ShadeRec` object plays a critical role in the ray tracer's shading procedures, as this chapter starts to illustrate in a simplified context. Listing 3.3 shows a declaration of the `ShadeRec` class with the data members that we need here. Note that one data member is a world reference. Although this is only used for shading, I've included it here, as it prevents the `ShadeRec` class from having a default constructor; the reference must always be initialized when a `ShadeRec` object is constructed (Listings 3.14 and 3.16) or copy constructed (Listing 3.17). Listing 3.3 includes the `ShadeRec` constructor code (see also the Notes and Discussion section). I haven't included an assignment operator, as the ray tracer is written in such a way that it's not required. For example, no class has a `ShadeRec` object as a data member.

3.4.4 Planes

Planes are the best geometric objects to discuss first because they are the easiest to intersect. To do this, we first substitute Equation (3.1) into the plane equation (2.6),

$$(\boldsymbol{p} - \boldsymbol{a}) \bullet \boldsymbol{n} = 0,$$

to get

$$(\boldsymbol{o} + t\,\boldsymbol{d} - \boldsymbol{a}) \bullet \boldsymbol{n} = 0.$$

This is a *linear equation* in the ray parameter t whose solution is

$$t = (\boldsymbol{a} - \boldsymbol{o})\ \ \boldsymbol{n} \,/\, (\boldsymbol{d} \bullet \boldsymbol{n}). \tag{3.4}$$

A linear equation has the form

$$at + b = 0,$$

where a and b are constants, and t is an unknown variable. The solution is

$$t = -b \,/\, a.$$

See Exercise 3.10.

Because linear equations have a single solution, Equation (3.4) tells us that a ray can only hit a plane once. We could now substitute the expression (3.4) for t into Equation (3.1) to get a symbolic expression for the hit-point coordinates, but we don't do this for two reasons. First, we don't need the hit-

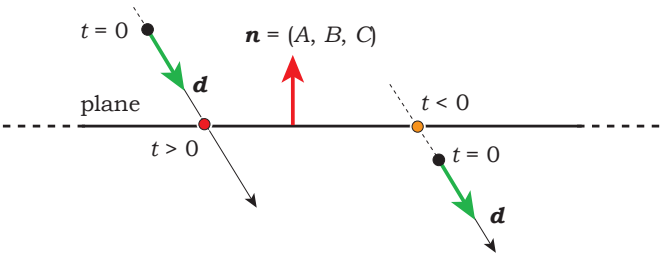


Figure 3.6. Ray-plane intersections.

point coordinates until we shade the point, and we only do that for the closest point to the ray origin. We won't know what that point will be until we've intersected the ray with all of the objects. Second, it's more efficient to calculate the *numerical* value of t from Equation (3.4) and substitute that into (3.1) to get *numerical* values for the coordinates. This applies to all geometric objects. The ray tracer must have numerical values for shading.

Figure 3.6 shows an edge-on view of a plane with two rays. The ray on the left hits the plane with $t > \varepsilon$, but the ray on the right hits it with $t < 0$. We must check that the value of t in Equation (3.4) satisfies $t > \varepsilon$ before recording that an intersection has occurred. In this context, it doesn't matter whether the normal to the plane points up or down in Figure 3.6, as this will only affect the shading.

```
class Plane: public GeometricObject {
     public:

          Plane(void);

          Plane(const Point3D p, const Normal& n);
          ...

          virtual bool
          hit(const Ray& ray, double& t, ShadeRec& s) const;

     private:

          Point3D                  point;       // point through which
                                                //    plane passes
          Normal                   normal;      // normal to the plane
          static const double      kEpsilon;    // see Chapter 16
};
```

Listing 3.4. The `Plane::hit` function.

```
bool
Plane::hit(const Ray& ray, double& tmin, ShadeRec& sr) const {
     double t = (point - ray.o) * normal / (ray.d * normal);

     if (t > kEpsilon) {
          tmin = t;
          sr.normal                = normal;
          sr.local_hit_point       = ray.o + t * ray.d;

          return (true);
     }
     else
          return (false);
}
```

Listing 3.5. The `Plane::hit` function.

What happens if the ray is parallel to the plane? In this case, $\boldsymbol{d} \bullet \boldsymbol{n} = 0$, and the value of the expression (3.4) is infinity. Is this a problem? Not if you are programming in C++, because floating-point calculations in this language satisfy the IEEE floating-point standard, where division by zero returns the legal number `INF` (infinity). As a result, there's no need to check for $\boldsymbol{d} \bullet \boldsymbol{n} = 0$ as a special case; your ray tracer will not crash if it divides by zero.

The class `Plane` stores the point and the normal. Its declaration appears in Listing 3.4 with two constructors; each class should have a default constructor, and other constructors as required.

Listing 3.5 shows the ray-plane hit function. Ray-object hit functions don't come any simpler than this.

All object hit functions compute and return information in three ways: their return type is a `bool` that indicates if the ray hits the object; they return the ray parameter for the nearest hit point (if any) through the parameter `tmin`; they return information required for shading with the `ShadeRec` parameter. We won't need the normal until shading in Chapter 14, and we won't need the hit point `local_hit_point` until texturing in Chapter 29. By including them now, we won't have to change this hit function later on.

3.4.5 Spheres

Equation (2.8) for a sphere can be written in vector form as

$$(\boldsymbol{p} - \boldsymbol{c}) \bullet (\boldsymbol{p} - \boldsymbol{c}) - r^2 = 0 \tag{3.5}$$