GLOBAL EDITION

Interactive Computer Graphics

A Top-Down Approach with WebGL

SEVENTH EDITION

Edward Angel • Dave Shreiner

ALWAYS LEARNING

PEARSON

INTERACTIVE COMPUTER GRAPHICS

A Top-Down Approach with **WebGL** | 7TH EDITION

GLOBAL EDITION

This page is intentionally left blank.

INTERACTIVE COMPUTER GRAPHICS

A Top-Down Approach with **WebGL** | 7TH EDITION

GLOBAL EDITION

EDWARD ANGEL

University of New Mexico

DAVE SHREINER

ARM, Inc.

Global Edition contributions by

ARUP BHATTACHARYA

SOUMEN MUKHERJEE

RCC Institute of
Information Technology,
Kolkata

PEARSON

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Editorial Director, ECS Marcia Horton
Head of Learning Asset Acquisition, Global Edition Laura Dent
Acquisitions Editor Matt Goldstein
Assistant Acquisitions Editor, Global Edition Aditee Agarwal
Associate Project Editor, Global Edition Uttaran Das Gupta
Program Manager Kayla Smith-Tarbox
Director of Marketing Christy Lesko
Marketing Assistant Jon Bryant
Director of Production Erin Gregg
Senior Managing Editor Scott Disanno
Senior Project Manager Marilyn Lloyd
Manufacturing Buyer Linda Sager
Senior Manufacturing Controller, Production, Global Edition Trudy Kimber
Cover Designer Lumina Datamatics
Manager, Text Permissions Tim Nicholls
Text Permission Project Manager William Opaluch
Media Project Manager Renata Butera
Cover Image Abstract Studio/Shutterstock

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at: www.pearsonglobaleditions.com

ISBN 10: 1-292-01934-4
ISBN 13: 978-1-292-01934-5 (Print)
ISBN 13: 978-1-292-01933-8 (PDF)

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

10 9 8 7 6 5 4 3 2 1
14 13 12 11

Typeset by Cypress Graphics in Minion and Avenir 10/12 pt.

Printed and bound by Courier Kendallville in The United States of America.

To Rose Mary —E.A.

To Vicki, Bonnie, Bob, Cookie, and Goatee —D.S.

This page is intentionally left blank.

CONTENTS

PREFACE

This book is an introduction to computer graphics with an emphasis on applications programming. The first edition, which was published in 1997, was somewhat revolutionary in using OpenGL and a top-down approach. Over the succeeding 16 years and 6 editions, this approach has been adopted by most introductory classes in computer graphics and by virtually all the competing textbooks.

The sixth edition reflected the recent major changes in graphics software due to major changes in graphics hardware. In particular, the sixth edition was fully shader-based, enabling readers to create applications that could fully exploit the capabilities of modern GPUs. We noted that these changes are also part of OpenGL ES 2.0, which is being used to develop applications for embedded systems and handheld devices, such as cell phones and tablets, and of WebGL, its JavaScript implementation. At the time, we did not anticipate the extraordinary interest in WebGL that began as soon as web browsers became available that support WebGL through HTML5.

As we continued to write our books, teach our SIGGRAPH courses, and pursue other graphics-related activities, we became aware of the growing excitement about WebGL. WebGL applications were running everywhere, including on some of the latest smart phones, and even though WebGL lacks some of the advanced features of the latest versions of OpenGL, the ability to integrate it with HTML5 opened up a wealth of new application areas. As an added benefit, we found it much better suited than desktop OpenGL for teaching computer graphics. Consequently, we decided to do a seventh edition that uses WebGL exclusively. We believe that this edition is every bit as revolutionary as any of the previous editions.

New to the Seventh Edition

- WebGL is used throughout.
- All code is written in JavaScript.
- All code runs in recent web browsers.
- A new chapter on interaction is included.
- Additional material on render-to-texture has been added.
- Additional material on displaying meshes has been added.
- An efficient matrix–vector package is included.
- An introduction to agent-based modeling has been added.

A Top-Down Approach

Recent advances and the success of the first six editions continue to reinforce our belief in a top-down, programming-oriented approach to introductory computer graphics. Although many computer science and engineering departments now support more than one course in computer graphics, most students will take only a single course. Such a course usually is placed in the curriculum after students have already studied programming, data structures, algorithms, software engineering, and basic mathematics. Consequently, a class in computer graphics allows the instructor to build on these topics in a way that can be both informative and fun. We want these students to be programming three-dimensional applications as soon as possible. Low-level algorithms, such as those that draw lines or fill polygons, can be dealt with later, after students are creating graphics.

When asked "why teach programming," John Kemeny, a pioneer in computer education, used a familiar automobile analogy: You don't have to know what's under the hood to be literate, but unless you know how to program, you'll be sitting in the back seat instead of driving. That same analogy applies to the way we teach computer graphics. One approach—the algorithmic approach—is to teach everything about what makes a car function: the engine, the transmission, the combustion process. A second approach—the survey approach—is to hire a chauffeur, sit back, and see the world as a spectator. The third approach—the programming approach that we have adopted here—is to teach you how to drive and how to take yourself wherever you want to go. As the old auto rental commercial used to say, "Let us put *you* in the driver's seat."

Programming with WebGL and JavaScript

When Ed began teaching computer graphics 30 years ago, the greatest impediment to implementing a programming-oriented course, and to writing a textbook for that course, was the lack of a widely accepted graphics library or application programming interface (API). Difficulties included high cost, limited availability, lack of generality, and high complexity. The development of OpenGL resolved most of the difficulties many of us had experienced with other APIs and with the alternative of using home-brewed software. OpenGL today is supported on all platforms and is widely accepted as a cross-platform standard.

A graphics class teaches far more than the use of a particular API, but a good API makes it easier to teach key graphics topics, including three-dimensional graphics, lighting and shading, client–server graphics, modeling, and implementation algorithms. We believe that OpenGL's extensive capabilities and well-defined architecture lead to a stronger foundation for teaching both theoretical and practical aspects of the field and for teaching advanced concepts, including texture mapping, compositing, and programmable shaders.

Ed switched his classes to OpenGL about 18 years ago and the results astounded him. By the middle of the semester, *every* student was able to write a moderately complex three-dimensional application that required understanding of three-dimensional viewing and event-driven input. In the previous years of teaching

computer graphics, he had never come even close to this result. That class led to the first edition of this book.

This book is a textbook on computer graphics; it is not an OpenGL or WebGL manual. Consequently, it does not cover all aspects of the WebGL API but rather explains only what is necessary for mastering this book's contents. It presents WebGL at a level that should permit users of other APIs to have little difficulty with the material.

Unlike previous editions, this one uses WebGL and JavaScript for all the examples. WebGL is a JavaScript implementation of OpenGL ES 2.0 and runs in most recent browsers. Because it is supported by HTML5, not only does it provide compatibility with other applications but also there are no platform dependences; WebGL runs within the browser and makes use of the local graphics hardware. Although JavaScript is not the usual programming language with which we teach most programming courses, it is the language of the Web. Over the past few years, JavaScript has become increasingly more powerful and our experience is that students who are comfortable with Java, C, or C++ will have little trouble programming in JavaScript.

All the modern versions of OpenGL, including WebGL, require every application to provide two shaders written in the OpenGL Shading Language (GLSL). GLSL is similar to C but adds vectors and matrices as basic types, along with some C++ features such as operator overloading. We have added a JavaScript library **MV.js** that supports both our presentation of graphics functions and the types and operations in GLSL.

Intended Audience

This book is suitable for advanced undergraduates and first-year graduate students in computer science and engineering and for students in other disciplines who have good programming skills. The book also will be useful to many professionals. Between us, we have taught well over 100 short courses for professionals; our experiences with these nontraditional students have had a great influence on what we chose to include in the book.

Prerequisites for the book are good programming skills in JavaScript, C, C++, or Java; an understanding of basic data structures (linked lists, trees); and a rudimentary knowledge of linear algebra and trigonometry. We have found that the mathematical backgrounds of computer science students, whether undergraduates or graduates, vary considerably. Hence, we have chosen to integrate into the text much of the linear algebra and geometry that is required for fundamental computer graphics.

Organization of the Book

The book is organized as follows. Chapter 1 provides an overview of the field and introduces image formation by optical devices; thus, we start with three-dimensional concepts immediately. Chapter 2 introduces programming using WebGL. Although the first example program that we develop (each chapter has one or more complete programming examples) is two-dimensional, it is embedded in a three-dimensional setting and leads to a three-dimensional extension. We introduce interactive graphics

in Chapter 3 and develop event-driven graphics within the browser environment. Chapters 4 and 5 concentrate on three-dimensional concepts. Chapter 4 is concerned with defining and manipulating three-dimensional objects, whereas Chapter 5 is concerned with viewing them. Chapter 6 introduces light–material interactions and shading. Chapter 7 introduces many of the new discrete capabilities that are now supported in graphics hardware and by WebGL. All these techniques involve working with various buffers. These chapters should be covered in order and can be taught in about 10 weeks of a 15-week semester.

The last five chapters can be read in almost any order. All five are somewhat open-ended and can be covered at a survey level, or individual topics can be pursued in depth. Chapter 8 surveys implementation. It gives one or two major algorithms for each of the basic steps, including clipping, line generation, and polygon fill. Chapter 9 includes a number of topics that fit loosely under the heading of hierarchical modeling. The topics range from building models that encapsulate the relationships between the parts of a model, to high-level approaches to graphics over the Internet. Chapter 9 also includes an introduction to scene graphs. Chapter 10 introduces a number of procedural methods, including particle systems, fractals, and procedural noise. Curves and surfaces, including subdivision surfaces, are discussed in Chapter 11. Chapter 12 surveys alternate approaches to rendering. It includes expanded discussions of ray tracing and radiosity, and an introduction to image-based rendering and parallel rendering.

Appendix A presents the details of the WebGL functions needed to read, compile, and link the application and shaders. Appendices B and C contain a review of the background mathematics. Appendix D discusses sampling and aliasing starting with Nyquist's theorem and applying these results to computer graphics.

Changes from the Sixth Edition

The reaction of readers to the first six editions of this book was overwhelmingly positive, especially to the use of OpenGL and the top-down approach. In the sixth edition, we abandoned the fixed-function pipeline and went to full shader-based OpenGL. In this edition, we move to WebGL, which is not only fully shader-based—each application must provide at least a vertex shader and a fragment shader–but also a version that works within the latest web browsers.

Applications are written in JavaScript. Although JavaScript has its own idiosyncrasies, we do not expect that students with experience in a high-level language, such as Java, C, or C++, will experience any serious problems with it.

As we pointed out earlier in this preface, every application must provide its own shaders. Consequently, programmable shaders and GLSL need to be introduced in Chapter 2. Many of the examples produce the same output as in previous editions, but the code is very different.

In the sixth edition, we eliminated a separate chapter on input and interaction, incorporating the material in other chapters. With this edition, we revert to a separate chapter. This decision is based on the ease and flexibility with which we can integrate event-driven input with WebGL through HTML5.

We have added additional material on off-screen rendering and render-to-texture. These techniques have become fundamental to using GPUs for a variety of compute-intensive applications such as image processing and simulation.

Given the positive feedback we've received on the core material from Chapters 1–6 in previous editions, we've tried to keep the changes to those chapters to a minimum. We see Chapters 1–7 as the core of any introductory course in computer graphics. Chapters 8–12 can be used in almost any order, either as a survey in a one-semester course or as the basis of a two-semester sequence.

Support Materials

The support for the book is at www.pearsonglobaleditions.com/Angel. Support material that is available to all readers of this book includes

- Sources of information on WebGL
- Program code
- Solutions to selected exercises
- PowerPoint lectures
- Figures from the book

Additional support materials, including solutions to all the nonprogramming exercises, are available only to instructors adopting this textbook for classroom use. Please contact your school's Pearson Education representative or visit www.pearsonglobaleditions.com/Angel for information on obtaining access to this material.

Acknowledgments

Ed has been fortunate over the past few years to have worked with wonderful students at the University of New Mexico. They were the first to get him interested in OpenGL, and he has learned much from them. They include Ye Cong, Pat Crossno, Tommie Daniel, Chris Davis, Lisa Desjarlais, Kim Edlund, Lee Ann Fisk, Maria Gallegos, Brian Jones, Christopher Jordan, Takeshi Hakamata, Max Hazelrigg, Sheryl Hurley, Thomas Keller, Ge Li, Pat McCormick, Al McPherson, Ken Moreland, Martin Muller, David Munich, Jim Pinkerton, Jim Prewett, Dave Rogers, Hal Smyer, Dave Vick, Hue (Bumgarner-Kirby) Walker, Brian Wylie, and Jin Xiong. Many of the examples in the color plates were created by these students.

The first edition of this book was written during Ed's sabbatical; various parts were written in five different countries. The task would not have been accomplished without the help of a number of people and institutions that made their facilities available to him. He is greatly indebted to Jonas Montilva and Chris Birkbeck of the Universidad de los Andes (Venezuela), to Rodrigo Gallegos and Aristides Novoa of the Universidad Tecnologica Equinoccial (Ecuador), to Long Wen Chang of the National Tsing Hua University (Taiwan), and to Kim Hong Wong and Pheng Ann Heng of the Chinese University of Hong Kong. Ramiro Jordan of ISTEC and the University of New Mexico made possible many of these visits. John Brayer and Jason

Stewart at the University of New Mexico and Helen Goldstein at Addison-Wesley somehow managed to get a variety of items to him wherever he happened to be. His website contains a description of his adventures writing the first edition.

David Kirk and Mark Kilgard at NVIDIA were kind enough to provide graphics cards for testing many of the algorithms. A number of other people provided significant help. Ed thanks Ben Bederson, Gonzalo Cartagenova, Tom Caudell, Kathi Collins, Kathleen Danielson, Roger Ehrich, Robert Geist, Chuck Hansen, Mark Henne, Bernard Moret, Dick Nordhaus, Helena Saona, Dave Shreiner, Vicki Shreiner, Gwen Sylvan, and Mason Woo. Mark Kilgard, Brian Paul, and Nate Robins are owed a great debt by the OpenGL community for creating software that enables OpenGL code to be developed over a variety of platforms.

At the University of New Mexico, the Art, Research, Technology, and Science Laboratory (ARTS Lab) and the Center for High Performance Computing have provided support for many of Ed's projects. The Computer Science Department, the Arts Technology Center in the College of Fine Arts, the National Science Foundation, Sandia National Laboratories, and Los Alamos National Laboratory have supported many of Ed's students and research projects that led to parts of this book. David Beining, formerly with the Lodestar Astronomy Center and now at the ARTS Lab, has provided tremendous support for the Fulldome Project. Sheryl Hurley, Christopher Jordan, Laurel Ladwig, Jon Strawn and Hue (Bumgarner-Kirby) Walker provided some of the images in the color plates through Fulldome projects. Hue Walker has done the wonderful covers for previous editions and some of the examples in the Color Plates.

Ed would also like to acknowledge the informal group that started at the Santa Fe Complex, including Jeff Bowles, Ruth Chabay, Stephen Guerin, Bruce Sherwood, Scott Wittenberg, and especially JavaScript evangelist Owen Densmore, who convinced him to teach a graphics course in Santa Fe in exchange for getting him involved with JavaScript. We've all gained by the experience.

Dave would like first to thank Ed for asking him to participate in this project. We've exchanged ideas on OpenGL and how to teach it for many years, and it's exciting to advance those concepts to new audiences. Dave would also like to thank those who created OpenGL, and who worked at Silicon Graphics Computer Systems, leading the way in their day. He would like to recognize the various Khronos working groups who continue to evolve the API and bring graphics to unexpected places. Finally, as Ed mentioned, SIGGRAPH has featured prominently in the development of these materials, and is definitely owed a debt of gratitude for providing access to enthusiastic test subjects for exploring our ideas.

Reviewers of the manuscript drafts provided a variety of viewpoints on what we should include and what level of presentation we should use. These reviewers for previous editions include Gur Saran Adhar (University of North Carolina at Wilmington), Mario Agrular (Jacksonville State University), Michael Anderson (University of Hartford), Norman I. Badler (University of Pennsylvania), Mike Bailey (Oregon State University), Marty Barrett (East Tennessee State University), C. S. Bauer (University of Central Florida), Bedrich Benes (Purdue University), Kabekode V. Bhat (The Pennsylvania State University), Isabelle Bichindaritz (University of Washington,

Tacoma), Cory D. Boatright (University of Pennsylvania), Eric Brown, Robert P. Burton (Brigham Young University), Sam Buss (University of California, San Diego), Kai H. Chang (Auburn University), James Cremer (University of Iowa), Ron DiNapoli (Cornell University), John David N. Dionisio (Loyola Marymount University), Eric Alan Durant (Milwaukee School of Engineering), David S. Ebert (Purdue University), Richard R. Eckert (Binghamton University), W. Randolph Franklin (Rensselaer Polytechnic Institute), Natacha Gueorguieva (City University of New York/College of Staten Island), Jianchao (Jack) Han (California State University, Dominguez Hills), Chenyi Hu (University of Central Arkansas), George Kamberov (Stevens Institute of Technology), Mark Kilgard (NVIDIA Corporation), Lisa B. Lancor (Southern Connecticut State University), Chung Lee (California State Polytechnic University, Pomona), John L. Lowther (Michigan Technological University), R. Marshall (Boston University and Bridgewater State College), Hugh C. Masterman (University of Massachusetts, Lowell), Bruce A. Maxwell (Swathmore College), Tim McGraw (West Virginia University), James R. Miller (University of Kansas), Rodrigo Obando (Columbus State University), Jon A. Preston (Southern Polytechnic State University), Andrea Salgian (The College of New Jersey), Lori L. Scarlatos (Brooklyn College, CUNY), Han-Wei Shen (The Ohio State University), Oliver Staadt (University of California, Davis), Stephen L. Stepoway (Southern Methodist University), Bill Toll (Taylor University), Michael Wainer (Southern Illinois University, Carbondale), Yang Wang (Southern Methodist State University), Steve Warren (Kansas State University), Mike Way (Florida Southern College), George Wolberg (City College of New York), Xiaoyu Zhang (California State University San Marcos), Ye Zhao (Kent State University). and Ying Zhu (Georgia State University). Although the final decisions may not reflect their views—which often differed considerably from one another—each reviewer forced us to reflect on every page of the manuscript.

The reviewers for this edition were particularly supportive. They include Mike Bailey (Oregon State University), Patrick Cozzi (University of Pennsylvania and Analytic Graphics, Inc) and Jeff Parker (Harvard University). All of them were familiar with previous editions and excited about the potential of moving their classes to WebGL.

We would also like to acknowledge the entire production team at Addison-Wesley. Ed's editors, Peter Gordon, Maite Suarez-Rivas, and Matt Goldstein, have been a pleasure to work with through seven editions of this book and the OpenGL primer. For this edition, Marilyn Lloyd and Kayla Smith-Tarbox at Pearson have provided considerable help. Through seven editions, Paul Anagnostopoulos at Windfall Software has always been more than helpful in assisting with TEX problems. Ed is especially grateful to Lyn Dupré. If the readers could see the original draft of the first edition, they would understand the wonders that Lyn does with a manuscript.

Ed wants to particularly recognize his wife, Rose Mary Molnar, who did the figures for his first graphics book, many of which form the basis for the figures in this book. Probably only other authors can fully appreciate the effort that goes into the book production process and the many contributions and sacrifices our partners make to that effort. The dedication to this book is a sincere but inadequate recognition of all of Rose Mary's contributions to Ed's work.

Dave would like to recognize the support and encouragement of Vicki, his wife, without whom creating works like this would never occur. Not only does she provide warmth and companionship but also provides invaluable feedback on our presentation and materials. She's been a valuable, unrecognized partner in all of Dave's OpenGL endeavors.

Ed Angel
Dave Shreiner

Pearson would like to thank the following persons for reviewing the Global Edition: Chitra Dhawale (PR Pote College of Engineering and Management, Amravati), Mohanesh Bevoor Mahalingappa (National Institute of Engineering, Mysore), and Pankaj K. Sa (National Institute of Technology, Rourkela).

INTERACTIVE COMPUTER GRAPHICS

A Top-Down Approach with **WebGL** | 7TH EDITION

GLOBAL EDITION

This page is intentionally left blank.

CHAPTER 1

GRAPHICS SYSTEMS AND MODELS

It would be difficult to overstate the importance of computer and communication technologies in our lives. Activities as wide-ranging as filmmaking, publishing, banking, and education have undergone revolutionary changes as these technologies alter the ways in which we conduct our daily activities. The combination of computers, networks, and the complex human visual system, through computer graphics, has been instrumental in these advances and has led to new ways of displaying information, seeing virtual worlds, and communicating with both other people and machines.

Computer graphics is concerned with all aspects of producing pictures or images using a computer. The field began humbly 50 years ago, with the display of a few lines on a cathode-ray tube (CRT); now, we can generate images by computer that are indistinguishable from photographs of real objects. We routinely train pilots with simulated airplanes, generating graphical displays of a virtual environment in real time. Feature-length movies made entirely by computer have been successful, both critically and financially.

In this chapter, we start our journey with a short discussion of applications of computer graphics. Then we overview graphics systems and imaging. Throughout this book, our approach stresses the relationships between computer graphics and image formation by familiar methods, such as drawing by hand and photography. We will see that these relationships can help us to design application programs, graphics libraries, and architectures for graphics systems.

In this book, we will use **WebGL,** a graphics software system supported by most modern web browsers. WebGL is a version of OpenGL, which is the widely accepted standard for developing graphics applications. WebGL is easy to learn, and it possesses most of the characteristics of the full (or desktop) OpenGL and of other important graphics systems. Our approach is top-down. We want you to start writing, as quickly as possible, application programs that will generate graphical output. After you begin writing simple programs, we shall discuss how the underlying graphics library and the hardware are implemented. This chapter should give a sufficient overview for you to proceed to writing programs.

1.1 APPLICATIONS OF COMPUTER GRAPHICS

The development of computer graphics has been driven both by the needs of the user community and by advances in hardware and software. The applications of computer graphics are many and varied; we can, however, divide them into four major areas:

1. Display of information
2. Design
3. Simulation and animation
4. User interfaces

Although many applications span two or more of these areas, the development of the field was based largely on separate work in each.

1.1.1 Display of Information

Classical graphics techniques arose as a medium to convey information among people. Although spoken and written languages serve a similar purpose, the human visual system is unrivaled both as a processor of data and as a pattern recognizer. More than 4000 years ago, the Babylonians displayed floor plans of buildings on stones. More than 2000 years ago, the Greeks were able to convey their architectural ideas graphically, even though the related mathematics was not developed until the Renaissance. Today, the same type of information is generated by architects, mechanical designers, and draftspeople using computer-based drafting systems.

For centuries, cartographers have developed maps to display celestial and geographical information. Such maps were crucial to navigators as these people explored the ends of the earth; maps are no less important today in fields such as geographic information systems. Now, maps can be developed and manipulated in real time over the Internet.

During the past 100 years, workers in the field of statistics have explored techniques for generating plots that aid the viewer in determining the information in a set of data. Now, we have computer plotting packages that provide a variety of plotting techniques and color tools that can handle multiple large data sets. Nevertheless, it is still the human ability to recognize visual patterns that ultimately allows us to interpret the information contained in the data. The field of information visualization is becoming increasingly more important as we have to deal with understanding complex phenomena, from problems in bioinformatics to detecting security threats.

Medical imaging poses interesting and important data analysis problems. Modern imaging technologies—such as computed tomography (CT), magnetic resonance imaging (MRI), ultrasound, and positron-emission tomography (PET)—generate three-dimensional data that must be subjected to algorithmic manipulation to provide useful information. Color Plate 20 shows an image of a person's head in which the skin is displayed as transparent and the internal structures are displayed as opaque. Although the data were collected by a medical imaging system, computer graphics produced the image that shows the structures.

Supercomputers now allow researchers in many areas to solve previously intractable problems. The field of scientific visualization provides graphical tools that help these researchers interpret the vast quantity of data that they generate. In fields such as fluid flow, molecular biology, and mathematics, images generated by conversion of data to geometric entities that can be displayed have yielded new insights into complex processes. For example, Color Plate 19 shows fluid dynamics in the mantle of the earth. The system used a mathematical model to generate the data. We present various visualization techniques as examples throughout the rest of the text.

1.1.2 Design

Professions such as engineering and architecture are concerned with design. Starting with a set of specifications, engineers and architects seek a cost-effective and aesthetic solution that satisfies the specifications. Design is an iterative process. Rarely in the real world is a problem specified such that there is a unique optimal solution. Design problems are either *overdetermined*, such that they possess no solution that satisfies all the criteria, much less an optimal solution, or *underdetermined*, such that they have multiple solutions that satisfy the design criteria. Thus, the designer works in an iterative manner. She generates a possible design, tests it, and then uses the results as the basis for exploring other solutions.

The power of the paradigm of humans interacting with images on the screen of a CRT was recognized by Ivan Sutherland over 50 years ago. Today, the use of interactive graphical tools in computer-aided design (CAD) pervades fields such as architecture and the design of mechanical parts and of very-large-scale integrated (VLSI) circuits. In many such applications, the graphics are used in a number of distinct ways. For example, in a VLSI design, the graphics provide an interactive interface between the user and the design package, usually by means of such tools as menus and icons. In addition, after the user produces a possible design, other tools analyze the design and display the analysis graphically. Color Plates 9 and 10 show two views of the same architectural design. Both images were generated with the same CAD system. They demonstrate the importance of having the tools available to generate different images of the same objects at different stages of the design process.

1.1.3 Simulation and Animation

Once graphics systems evolved to be capable of generating sophisticated images in real time, engineers and researchers began to use them as simulators. One of the most important uses has been in the training of pilots. Graphical flight simulators have proved both to increase safety and to reduce training expenses. The use of special VLSI chips has led to a generation of arcade games as sophisticated as flight simulators. Games and educational software for home computers are almost as impressive.

The success of flight simulators led to the use of computer graphics for animation in the television, motion picture, and advertising industries. Entire animated movies can now be made by computer at a cost less than that of movies made with traditional hand-animation techniques. The use of computer graphics with hand animation allows the creation of technical and artistic effects that are not possible with either alone. Whereas computer animations have a distinct look, we can also generate

photorealistic images by computer. Images that we see on television, in movies, and in magazines often are so realistic that we cannot distinguish computer-generated or computer-altered images from photographs. In Chapter 6, we discuss many of the lighting effects used to produce computer animations. Color Plates 15 and 23 show realistic lighting effects that were created by artists and computer scientists using animation software. Although these images were created for commercial animations, interactive software to create such effects is widely available.

The field of virtual reality (VR) has opened up many new horizons. A human viewer can be equipped with a display headset that allows her to see separate images with her right eye and her left eye so that she has the effect of stereoscopic vision. In addition, her body location and position, possibly including her head and finger positions, are tracked by the computer. She may have other interactive devices available, including force-sensing gloves and sound. She can then act as part of a computer-generated scene, limited only by the image generation ability of the computer. For example, a surgical intern might be trained to do an operation in this way, or an astronaut might be trained to work in a weightless environment. Color Plate 22 shows one frame of a VR simulation of a simulated patient used for remote training of medical personnel.

Simulation and virtual reality have come together in many exciting ways in the film industry. Recently, stereo (3D) movies have become both profitable and highly acclaimed by audiences. Special effects created using computer graphics are part of virtually all movies, as are more mundane uses of computer graphics such as removal of artifacts from scenes. Simulations of physics are used to create visual effects ranging from fluid flow to crowd dynamics.

1.1.4 User Interfaces

Our interaction with computers has become dominated by a visual paradigm that includes windows, icons, menus, and a pointing device, such as a mouse. From a user's perspective, windowing systems such as the X Window System, Microsoft Windows, and the Macintosh Operating System differ only in details. More recently, millions of people have become users of the Internet. Their access is through graphical network browsers, such as Firefox, Chrome, Safari, and Internet Explorer, that use these same interface tools. We have become so accustomed to this style of interface that we often forget that what we are doing is working with computer graphics.

Although personal computers and workstations evolved by somewhat different paths, at present they are indistinguishable. When you add in smart phones, tablets, and game consoles, we have an incredible variety of devices with considerable computing power, all of which can access the World Wide Web through a browser. For lack of a better term, we will tend to use *computer* to include all these devices.

Color Plate 13 shows the interface used with a high-level modeling package. It demonstrates the variety of tools available in such packages and the interactive devices the user can employ in modeling geometric objects. Although we are familiar with this style of graphical user interface, devices such as smart phones and tablets have popularized touch-sensitive interfaces that allow the user to interact with every pixel on the display.

1.2 A GRAPHICS SYSTEM

A computer graphics system is a computer system; as such, it must have all the components of a general-purpose computer system. Let us start with the high-level view of a graphics system, as shown in the block diagram in Figure 1.1. There are six major elements in our system:

1. Input devices
2. Central Processing Unit
3. Graphics Processing Unit
4. Memory
5. Framebuffer
6. Output devices

This model is general enough to include workstations and personal computers, interactive game systems, mobile phones, GPS systems, and sophisticated image generation systems. Although most of the components are present in a standard computer, it is the way each element is specialized for computer graphics that characterizes this diagram as a portrait of a graphics system. As more and more functionality can be included in a single chip, many of the components are not physically separate. The CPU and GPU can be on the same chip and their memory can be shared. Nevertheless, the model still describes the software architecture and will be helpful as we study the various parts of computer graphics systems.

1.2.1 Pixels and the Framebuffer

Virtually all modern graphics systems are raster based. The image we see on the output device is an array—the **raster**—of picture elements, or **pixels**, produced by the graphics system. As we can see from Figure 1.2, each pixel corresponds to a location, or small area, in the image. Collectively, the pixels are stored in a part of

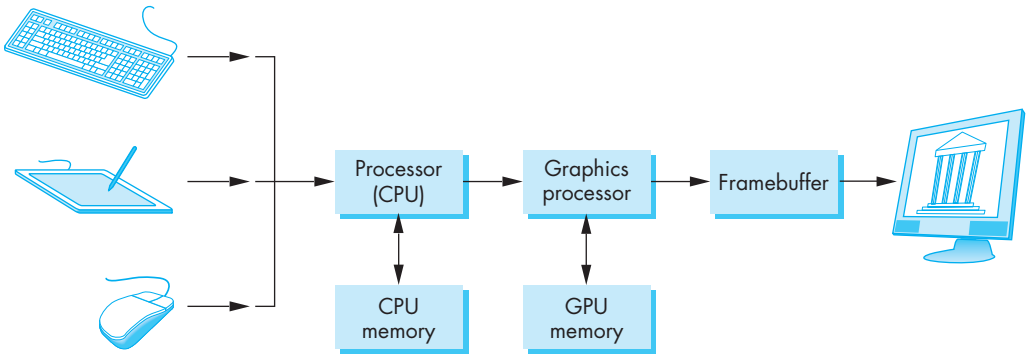
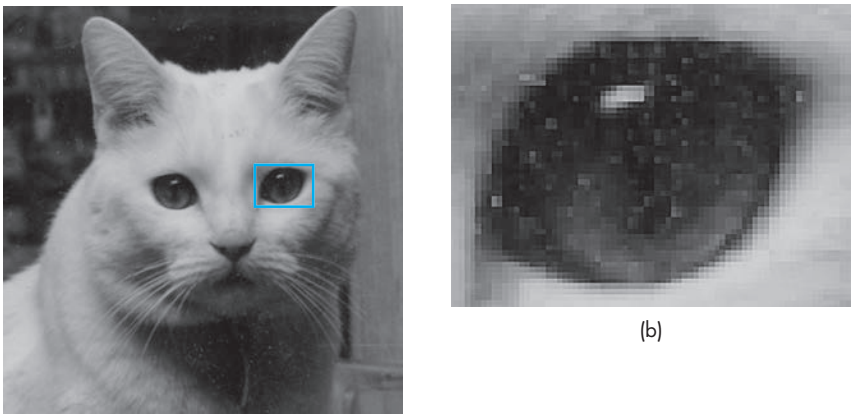


FIGURE 1.1 A graphics system.



(a) (b)

FIGURE 1.2 Pixels. (a) Image of Yeti the cat. (b) Detail of area around one eye showing individual pixels.

memory called the **framebuffer**.[1] The framebuffer can be viewed as the core element of a graphics system. Its **resolution**—the number of pixels in the framebuffer—determines the detail that you can see in the image. The **depth**, or **precision**, of the framebuffer, defined as the number of bits that are used for each pixel, determines properties such as how many colors can be represented on a given system. For example, a 1-bit-deep framebuffer allows only two colors, whereas an 8-bit-deep framebuffer allows 2^8 (256) colors. In **full-color** systems, there are 24 (or more) bits per pixel. Such systems can display sufficient colors to represent most images realistically. They are also called **true-color** systems, or **RGB color** systems, because individual groups of bits in each pixel are assigned to each of the three primary colors—red, green, and blue—used in most displays. **High dynamic range** (HDR) systems use 12 or more bits for each color component. Until recently, framebuffers stored colors in integer formats. Recent framebuffers use floating point and thus support HDR colors more easily.

In a simple system, the framebuffer holds only the colored pixels that are displayed on the screen. In most systems, the framebuffer holds far more information, such as depth information needed for creating images from three-dimensional data. In these systems, the framebuffer comprises multiple buffers, one or more of which are **color buffers** that hold the colored pixels that are displayed. For now, we can use the terms *framebuffer* and *color buffer* synonymously without confusion.

1.2.2 The CPU and the GPU

In a simple system, there may be only one processor, the **central processing unit** (**CPU**), which must perform both the normal processing and the graphical process-

1. Some references use *frame buffer* rather than *framebuffer*.

ing. The main graphical function of the processor is to take specifications of graphical primitives (such as lines, circles, and polygons) generated by application programs and to assign values to the pixels in the framebuffer that best represent these entities. For example, a triangle is specified by its three vertices, but to display its outline by the three line segments connecting the vertices, the graphics system must generate a set of pixels that appear as line segments to the viewer. The conversion of geometric entities to pixel colors and locations in the framebuffer is known as **rasterization** or **scan conversion**. In early graphics systems, the framebuffer was part of the standard memory that could be directly addressed by the CPU. Today, virtually all graphics systems are characterized by special-purpose **graphics processing units** (**GPUs**), custom-tailored to carry out specific graphics functions. The GPU can be located on the motherboard of the system or on a graphics card. The framebuffer is accessed through the graphics processing unit and usually is on the same circuit board as the GPU.

GPUs have evolved to the point where they are as complex or even more complex than CPUs. They are characterized both by special-purpose modules geared toward graphical operations and by a high degree of parallelism—recent GPUs contain over 100 processing units, each of which is user programmable. GPUs are so powerful that they can often be used as mini supercomputers for general-purpose computing. We will discuss GPU architectures in more detail in Section 1.7.

1.2.3 Output Devices

Until recently, the dominant type of display (or **monitor**) was the **cathode-ray tube** (**CRT**). A simplified picture of a CRT is shown in Figure 1.3. When electrons strike the phosphor coating on the tube, light is emitted. The direction of the beam is controlled by two pairs of deflection plates. The output of the computer is converted, by digital-to-analog converters, to voltages across the x and y deflection plates. Light appears on the surface of the CRT when a sufficiently intense beam of electrons is directed at the phosphor.

If the voltages steering the beam change at a constant rate, the beam will trace a straight line, visible to the viewer. Such a device is known as the **random-scan**,

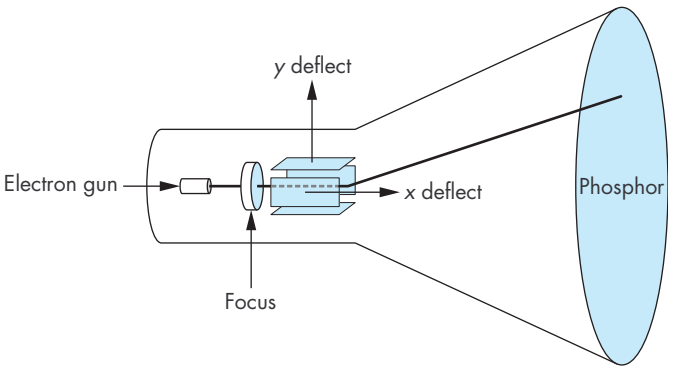


FIGURE 1.3 The cathode-ray tube (CRT).

calligraphic, or **vector** CRT, because the beam can be moved directly from any position to any other position. If intensity of the beam is turned off, the beam can be moved to a new position without changing any visible display. This configuration was the basis of early graphics systems that predated the present raster technology.

A typical CRT will emit light for only a short time—usually, a few milliseconds—after the phosphor is excited by the electron beam. For a human to see a steady, flicker-free image on most CRT displays, the same path must be retraced, or **refreshed**, by the beam at a sufficiently high rate, the **refresh rate**. In older systems, the refresh rate is determined by the frequency of the power system, 60 cycles per second or 60 hertz (Hz) in the United States and 50 Hz in much of the rest of the world. Modern displays are no longer coupled to these low frequencies and operate at rates up to about 85 Hz.

In a raster system, the graphics system takes pixels from the framebuffer and displays them as points on the surface of the display in one of two fundamental ways. In a **noninterlaced** system, the pixels are displayed row by row, or scan line by scan line, at the refresh rate. In an **interlaced** display, odd rows and even rows are refreshed alternately. Interlaced displays are used in commercial television. In an interlaced display operating at 60 Hz, the screen is redrawn in its entirety only 30 times per second, although the visual system is tricked into thinking the refresh rate is 60 Hz rather than 30 Hz. Viewers located near the screen, however, can tell the difference between the interlaced and noninterlaced displays. Noninterlaced displays are becoming more widespread, even though these displays must process pixels at twice the rate of the interlaced display.

Color CRTs have three different-colored phosphors (red, green, and blue), arranged in small groups. One common style arranges the phosphors in triangular groups called **triads**, each triad consisting of three phosphors, one of each primary. Most color CRTs have three electron beams, corresponding to the three types of phosphors. In the shadow-mask CRT (Figure 1.4), a metal screen with small holes—the **shadow mask**—ensures that an electron beam excites only phosphors of the proper color.

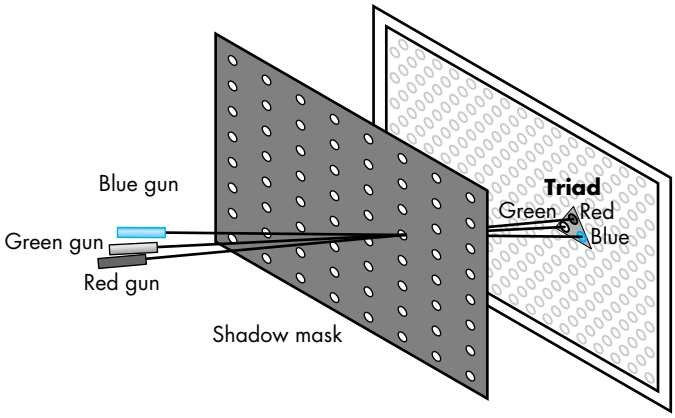


FIGURE 1.4 Shadow-mask CRT.

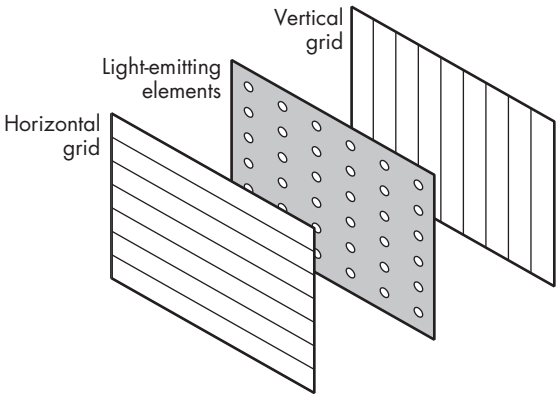


FIGURE 1.5 Generic flat-panel display.

Although CRTs are still common display devices, they are rapidly being replaced by flat-screen technologies. Flat-panel monitors are inherently raster based. Although there are multiple technologies available, including light-emitting diodes (LEDs), liquid-crystal displays (LCDs), and plasma panels, all use a two-dimensional grid to address individual light-emitting elements. Figure 1.5 shows a generic flat-panel monitor. The two outside plates each contain parallel grids of wires that are oriented perpendicular to each other. By sending electrical signals to the proper wire in each grid, the electrical field at a location, determined by the intersection of two wires, can be made strong enough to control the corresponding element in the middle plate. The middle plate in an LED panel contains light-emitting diodes that can be turned on and off by the electrical signals sent to the grid. In an LCD display, the electrical field controls the polarization of the liquid crystals in the middle panel, thus turning on and off the light passing through the panel. A plasma panel uses the voltages on the grids to energize gases embedded between the glass panels holding the grids. The energized gas becomes a glowing plasma.

Most projection systems are also raster devices. These systems use a variety of technologies, including CRTs and digital light projection (DLP). From a user perspective, they act as standard monitors with similar resolutions and precisions. Hard-copy devices, such as printers and plotters, are also raster based but cannot be refreshed.

Stereo (3D) television displays use alternate refresh cycles to switch the display between an image for the left eye and an image for the right eye. The viewer wears special glasses that are coupled to the refresh cycle. 3D movie projectors produce two images with different polarizations. The viewer wears polarized glasses so that each eye sees only one of the two projected images. As we shall see in later chapters, producing stereo images is basically a matter of changing the location of the viewer for each frame to obtain the left- and right-eye views.

1.2.4 Input Devices

Most graphics systems provide a keyboard and at least one other input device. The most common input devices are the mouse, the joystick, and the data tablet. Each

provides positional information to the system, and each is usually equipped with one or more buttons to provide signals to the processor. Often called **pointing devices**, these devices allow a user to indicate a particular location on the display.

Modern systems, such as game consoles, provide a much richer set of input devices, with new devices appearing almost weekly. In addition, there are devices that provide three- (and more) dimensional input. Consequently, we want to provide a flexible model for incorporating the input from such devices into our graphics programs. We will discuss input devices and how to use them in Chapter 3.

1.3 IMAGES: PHYSICAL AND SYNTHETIC

For many years, the pedagogical approach to teaching computer graphics started with how to construct raster images of simple two-dimensional geometric entities (for example, points, line segments, and polygons) in the framebuffer. Next, most textbooks discussed how to define two- and three-dimensional mathematical objects in the computer and image them with the set of two-dimensional rasterized primitives.

This approach worked well for creating simple images of simple objects. In modern systems, however, we want to exploit the capabilities of the software and hardware to create realistic images of computer-generated three-dimensional objects—a task that involves many aspects of image formation, such as lighting, shading, and properties of materials. Because such functionality is supported directly by most present computer graphics systems, we prefer to set the stage for creating these images now, rather than to expand a limited model later.

Computer-generated images are synthetic or artificial, in the sense that the objects being imaged do not exist physically. In this chapter, we argue that the preferred method to form computer-generated images is similar to traditional imaging methods, such as cameras and the human visual system. Hence, before we discuss the mechanics of writing programs to generate images, we discuss the way images are formed by optical systems. We construct a model of the image formation process that we can then use to understand and develop computer-generated imaging systems.

In this chapter, we make minimal use of mathematics. We want to establish a paradigm for creating images and to present a computer architecture for implementing that paradigm. Details are presented in subsequent chapters, where we shall derive the relevant equations.

1.3.1 Objects and Viewers

We live in a world of three-dimensional objects. The development of many branches of mathematics, including geometry and trigonometry, was in response to the desire to systematize conceptually simple ideas, such as the measurement of the size of objects and the distance between objects. Often, we seek to represent our understanding of such spatial relationships with pictures or images, such as maps, paintings, and photographs. Likewise, the development of many physical devices—including cameras, microscopes, and telescopes—was tied to the desire to visualize spatial relationships among objects. Hence, there always has been a fundamental link between the physics and the mathematics of image formation—one that we can exploit in our development of computer image formation.

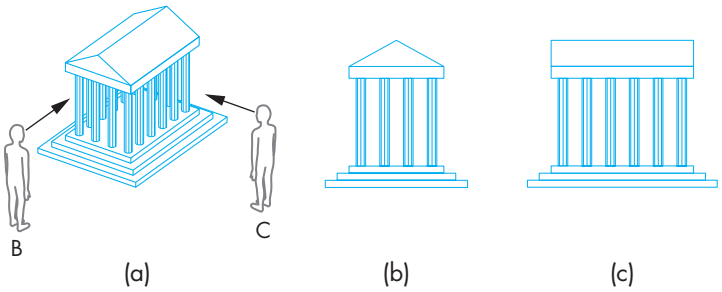


FIGURE 1.6 Image seen by three different viewers. (a) A's view. (b) B's view. (c) C's view.

Two basic entities must be part of any image formation process, be it mathematical or physical: *object* and *viewer*. The object exists in space independent of any image formation process and of any viewer. In computer graphics, where we deal with synthetic objects, we form objects by specifying the positions in space of various geometric primitives, such as points, lines, and polygons. In most graphics systems, a set of locations in space, or of **vertices**, is sufficient to define, or approximate, most objects. For example, a line can be specified by two vertices; a polygon can be specified by an ordered list of vertices; and a sphere can be specified by two vertices that specify its center and any point on its circumference. One of the main functions of a CAD system is to provide an interface that makes it easy for a user to build a synthetic model of the world. In Chapter 2, we show how WebGL allows us to build simple objects; in Chapter 9, we learn to define objects in a manner that incorporates relationships among objects.

Every imaging system must provide a means of forming images from objects. To form an image, we must have someone or something that is viewing our objects, be it a human, a camera, or a digitizer. It is the **viewer** that forms the image of our objects. In the human visual system, the image is formed on the back of the eye. In a camera, the image is formed in the film plane. It is easy to confuse images and objects. We usually see an object from our single perspective and forget that other viewers, located in other places, will see the same object differently. Figure 1.6(a) shows two viewers observing the same building. This image is what is seen by an observer A who is far enough away from the building to see both the building and the two other viewers B and C. From A's perspective, B and C appear as objects, just as the building does. Figure 1.6(b) and (c) shows the images seen by B and C, respectively. All three images contain the same building, but the image of the building is different in all three.

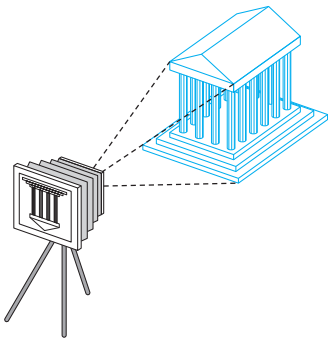


FIGURE 1.7 Camera system.

Figure 1.7 shows a camera system viewing a building. Here we can observe that both the object and the viewer exist in a three-dimensional world. However, the image that they define—what we find on the projection plane—is two-dimensional. The process by which the specification of the object is combined with the specification of the viewer to produce a two-dimensional image is the essence of image formation, and we will study it in detail.

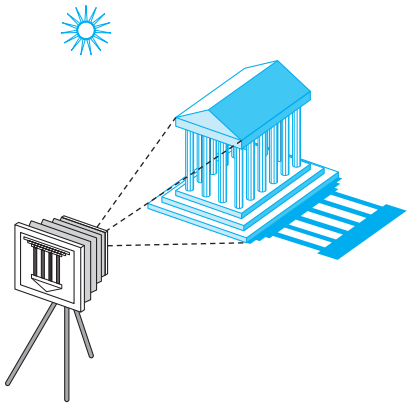


FIGURE 1.8 A camera system with an object and a light source.

1.3.2 Light and Images

The preceding description of image formation is far from complete. For example, we have yet to mention light. If there were no light sources, the objects would be dark, and there would be nothing visible in our image. Nor have we indicated how color enters the picture or what the effects of the surface properties of the objects are.

Taking a more physical approach, we can start with the arrangement in Figure 1.8, which shows a simple physical imaging system. Again, we see a physical object and a viewer (the camera); now, however, there is a light source in the scene. Light from the source strikes various surfaces of the object, and a portion of the reflected light enters the camera through the lens. The details of the interaction between light and the surfaces of the object determine how much light enters the camera.

Light is a form of electromagnetic radiation. Taking the classical view, we look at electromagnetic energy as traveling as waves[2] that can be characterized either by their wavelengths or by their frequencies.[3] The electromagnetic spectrum (Figure 1.9) includes radio waves, infrared (heat), and a portion that causes a response in our visual systems. This **visible spectrum**, which has wavelengths in the range of 350 to 780 nanometers (nm), is called (visible) **light**. A given light source has a color determined by the energy that it emits at various wavelengths. Wavelengths in the middle of the range, around 520 nm, are seen as green; those near 450 nm are seen as blue; and those near 650 nm are seen as red. Just as with a rainbow, light at wavelengths between red and green we see as yellow, and at wavelengths shorter than blue we see as violet.

Light sources can emit light either as a set of discrete frequencies or over a continuous range. A laser, for example, emits light at a single frequency, whereas an incandescent lamp emits energy over a range of frequencies. Fortunately, in computer

2. In Chapter 12, we will introduce photon mapping that is based on light being emitted in discrete packets.

3. The relationship between frequency (f) and wavelength (λ) is $f\lambda = c$, where c is the speed of light.

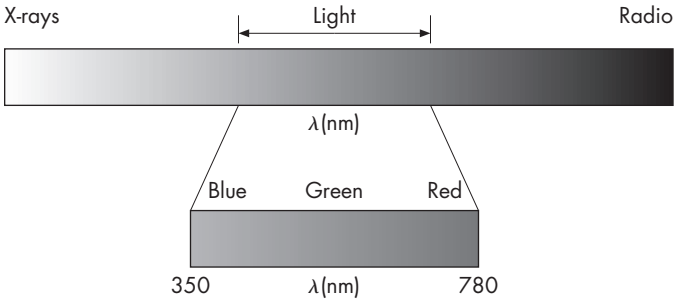


FIGURE 1.9 The electromagnetic spectrum.

graphics, except for recognizing that distinct frequencies are visible as distinct colors, we rarely need to deal with the physical properties of light.

Instead, we can follow a more traditional path that is correct when we are operating with sufficiently high light levels and at a scale where the wave nature of light is not a significant factor. **Geometric optics** models light sources as emitters of light energy, each of which have a fixed intensity. Modeled geometrically, light travels in straight lines, from the sources to those objects with which it interacts. An ideal **point source** emits energy from a single location at one or more frequencies equally in all directions. More complex sources, such as a lightbulb, can be characterized as emitting light over an area and by emitting more light in one direction than another. A particular source is characterized by the intensity of light that it emits at each frequency and by that light's directionality. We consider only point sources for now. More complex sources often can be approximated by a number of carefully placed point sources. Modeling of light sources is discussed in Chapter 6.

1.3.3 Imaging Models

There are multiple approaches to forming images from a set of objects, the light-reflecting properties of these objects, and the properties of the light sources in the scene. In this section, we introduce two physical approaches. Although these approaches are not suitable for the real-time graphics that we ultimately want, they will give us some insight into how we can build a useful imaging architecture. We return to these approaches in Chapter 12.

We can start building an imaging model by following light from a source. Consider the scene in Figure 1.10; it is illuminated by a single point source. We include the viewer in the figure because we are interested in the light that reaches her eye. The viewer can also be a camera, as shown in Figure 1.18. A **ray** is a semi-infinite line that emanates from a point and travels to infinity in a particular direction. Because light travels in straight lines, we can think in terms of rays of light emanating in all directions from our point source. A portion of these infinite rays contributes to the image on the film plane of our camera. For example, if the source is visible from the camera, some of the rays go directly from the source through the lens of the camera and strike the film plane. Most rays, however, go off to infinity, neither entering the

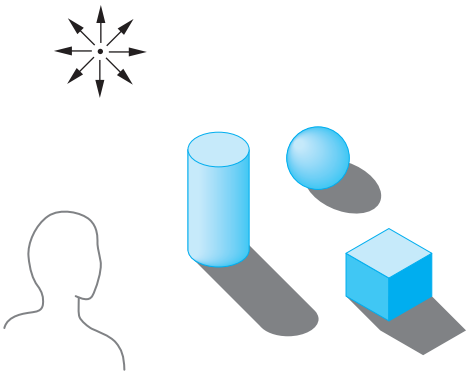


FIGURE 1.10 Scene with a single point light source.

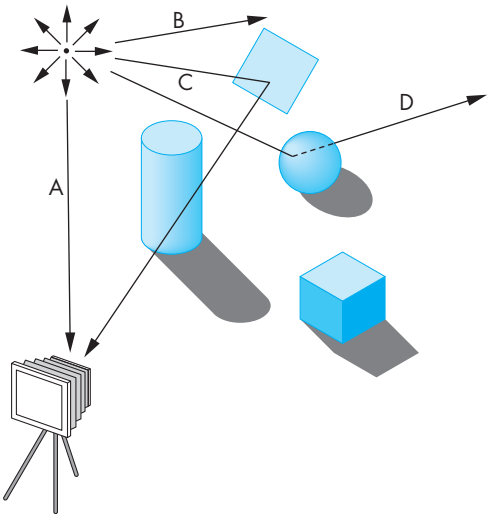


FIGURE 1.11 Ray interactions. Ray A enters camera directly. Ray B goes off to infinity. Ray C is reflected by a mirror. Ray D goes through a transparent sphere.

camera directly nor striking any of the objects. These rays contribute nothing to the image, although they may be seen by some other viewer. The remaining rays strike and illuminate objects. These rays can interact with the objects' surfaces in a variety of ways. For example, if the surface is a mirror, a reflected ray might—depending on the orientation of the surface—enter the lens of the camera and contribute to the image. Other surfaces scatter light in all directions. If the surface is transparent, the light ray from the source can pass through it and may interact with other objects, enter the camera, or travel to infinity without striking another surface. Figure 1.11 shows some of the possibilities.

Ray tracing and **photon mapping** are image formation techniques that are based on these ideas and that can form the basis for producing computer-generated images. We can use the ray-tracing idea to simulate physical effects as complex as we wish, as long as we are willing to carry out the requisite computing. Although tracing rays can provide a close approximation to the physical world, it is usually not well suited for real-time computation.

Other physical approaches to image formation are based on conservation of energy. The most important in computer graphics is **radiosity**. This method works best for surfaces that scatter the incoming light equally in all directions. Even in this case, radiosity requires more computation than can be done in real time. We defer discussion of these techniques until Chapter 12.

1.4 IMAGING SYSTEMS

We now introduce two imaging systems: the pinhole camera and the human visual system. The pinhole camera is a simple example of an imaging system that will enable us to understand the functioning of cameras and of other optical imagers. We emulate it to build a model of image formation. The human visual system is extremely complex but still obeys the physical principles of other optical imaging systems. We introduce it not only as an example of an imaging system but also because understanding its properties will help us to exploit the capabilities of computer graphics systems.

1.4.1 The Pinhole Camera

The pinhole camera in Figure 1.12 provides an example of image formation that we can understand with a simple geometric model. A **pinhole camera** is a box with a small hole in the center of one side; the film is placed inside the box on the side opposite the pinhole. Suppose that we orient our camera along the z-axis, with the pinhole at the origin of our coordinate system. We assume that the hole is so small that only a single ray of light, emanating from a point, can enter it. The film plane is

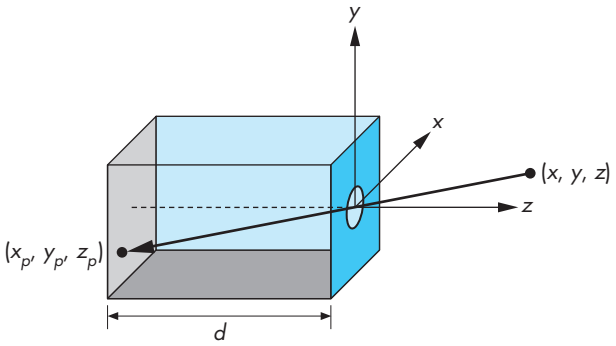


FIGURE 1.12 Pinhole camera.

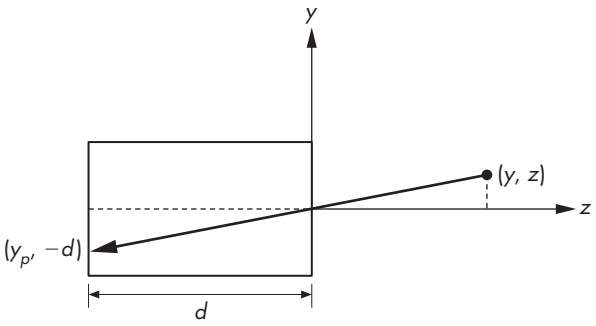


FIGURE 1.13 Side view of pinhole camera.

located a distance d from the pinhole. A side view (Figure 1.13) allows us to calculate where the image of the point (x, y, z) is on the film plane $z = -d$. Using the fact that the two triangles in Figure 1.13 are similar, we find that the y coordinate of the image is at y_p, where

$$y_p = -\frac{y}{z/d}.$$

A similar calculation, using a top view, yields

$$x_p = -\frac{x}{z/d}.$$

The point $(x_p, y_p, -d)$ is called the **projection** of the point (x, y, z). In our idealized model, the color on the film plane at this point will be the color of the point (x, y, z). The **field**, or **angle, of view** of our camera is the angle made by the largest object that our camera can image on its film plane. We can calculate the field of view with the aid of Figure 1.14.[4] If h is the height of the camera, the field of view (or angle of view) θ is

$$\theta = 2 \tan^{-1} \frac{h}{2d}.$$

The ideal pinhole camera has an infinite **depth of field**: Every point within its field of view is in focus. Every point in its field of view projects to a point on the back of the camera. The pinhole camera has two disadvantages. First, because the pinhole is so small—it admits only a single ray from a point source—almost no light enters the camera. Second, the camera cannot be adjusted to have a different field of view.

The jump to more sophisticated cameras and to other imaging systems that have lenses is a small one. By replacing the pinhole with a lens, we solve the two problems of the pinhole camera. First, the lens gathers more light than can pass through the

4. If we consider the problem in three, rather than two, dimensions, then the diagonal length of the film will substitute for h.

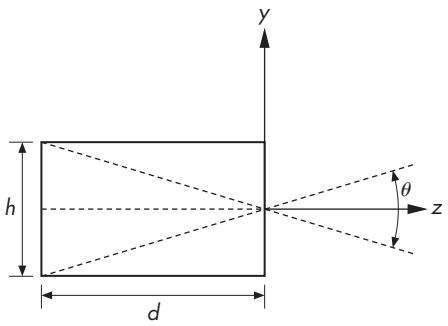


FIGURE 1.14 Field of view.

pinhole. The larger the aperture of the lens, the more light the lens can collect. Second, by picking a lens with the proper focal length—a selection equivalent to choosing d for the pinhole camera—we can achieve any desired field of view (up to 180 degrees). Lenses, however, do not have an infinite depth of field: Not all distances from the lens are in focus.

For our purposes, in this chapter we can work with a pinhole camera whose focal length is the distance d from the front of the camera to the film plane. Like the pinhole camera, computer graphics produces images in which all objects are in focus.

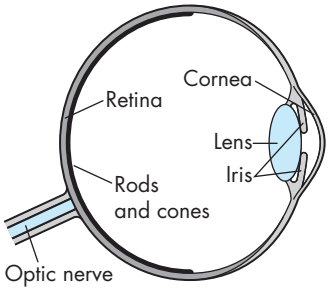


FIGURE 1.15 The human visual system.

1.4.2 The Human Visual System

Our extremely complex visual system has all the components of a physical imaging system such as a camera or a microscope. The major components of the visual system are shown in Figure 1.15. Light enters the eye through the cornea, a transparent structure that protects the eye, and the lens. The iris opens and closes to adjust the amount of light entering the eye. The lens forms an image on a two-dimensional structure called the **retina** at the back of the eye. The rods and cones (so named because of their appearance when magnified) are light sensors and are located on the retina. They are excited by electromagnetic energy in the range of 350 to 780 nm.

The rods are low-level-light sensors that account for our night vision and are not color sensitive; the cones are responsible for our color vision. The sizes of the rods and cones, coupled with the optical properties of the lens and cornea, determine the **resolution** of our visual systems, or our **visual acuity**. Resolution is a measure of what size objects we can see. More technically, it is a measure of how close we can place two points and still recognize that there are two distinct points.

The sensors in the human eye do not react uniformly to light energy at different wavelengths. There are three types of cones and a single type of rod. Whereas intensity is a physical measure of light energy, **brightness** is a measure of how intense we perceive the light emitted from an object to be. The human visual system does not have the same response to a monochromatic (single-frequency) red light as to a monochromatic green light. If these two lights were to emit the same energy, they would appear to us to have different brightness, because of the unequal response

of the cones to red and green light. We are most sensitive to green light, and least sensitive to red and blue.

Brightness is an overall measure of how we react to the intensity of light. Human color-vision capabilities are due to the different sensitivities of the three types of cones. The major consequence of having three types of cones is that, instead of having to work with all visible wavelengths individually, we can use three standard primaries to approximate any color that we can perceive. Consequently, most image production systems, including film and video, work with just three basic, or **primary**, colors. We discuss color in greater depth in Chapters 2 and 8.

The initial processing of light in the human visual system is based on the same principles used by most optical systems. However, the human visual system has a back end much more complex than that of a camera or telescope. The optic nerve is connected to the rods and cones in an extremely complex arrangement that has many of the characteristics of a sophisticated signal processor. The final processing is done in a part of the brain called the visual cortex, where high-level functions, such as object recognition, are carried out. We shall omit any discussion of high-level processing; instead, we can think simply in terms of an image that is conveyed from the rods and cones to the brain.

1.5 THE SYNTHETIC-CAMERA MODEL

Our models of optical imaging systems lead directly to the conceptual foundation for modern three-dimensional computer graphics. We look at creating a computer-generated image as being similar to forming an image using an optical system. This paradigm has become known as the **synthetic-camera model**. Consider the imaging system shown in Figure 1.16. We again see objects and a viewer. In this case, the viewer is a bellows camera.[5] The image is formed on the film plane at the back of the camera. So that we can emulate this process to create artificial images, we need to identify a few basic principles.

First, the specification of the objects is independent of the specification of the viewer. Hence, we should expect that, within a graphics library, there will be separate functions for specifying the objects and the viewer.

Second, we can compute the image using simple geometric calculations, just as we did with the pinhole camera. Consider the side view of the camera and a simple object in Figure 1.17. The view in part (a) of the figure is similar to that of the pinhole camera. Note that the image of the object is flipped relative to the object. Whereas with a real camera we would simply flip the film to regain the original orientation of the object, with our synthetic camera we can avoid the flipping by a simple trick. We draw another plane in front of the lens (Figure 1.17(b)) and work in three dimensions, as shown in Figure 1.18. We find the image of a point on the object

5. In a bellows camera, the front plane of the camera, where the lens is located, and the back of the camera, the film plane, are connected by flexible sides. Thus, we can move the back of the camera independently of the front of the camera, introducing additional flexibility in the image formation process. We use this flexibility in Chapter 5.

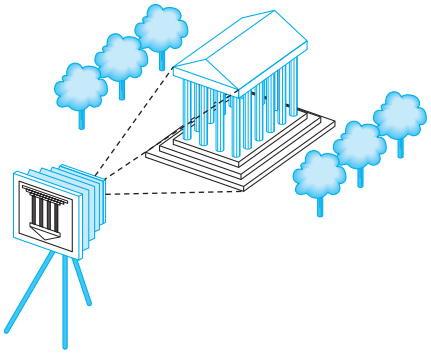


FIGURE 1.16 Imaging system.

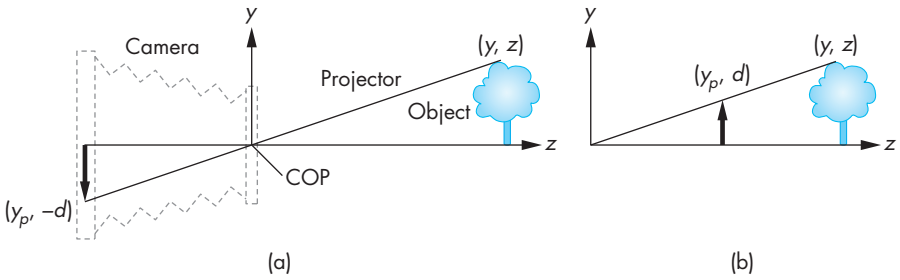


FIGURE 1.17 Equivalent views of image formation. (a) Image formed on the back of the camera. (b) Image plane moved in front of the camera.

on the virtual image plane by drawing a line, called a **projector**, from the point to the center of the lens, or the **center of projection** (**COP**). Note that all projectors are rays emanating from the center of projection. In our synthetic camera, the virtual image plane that we have moved in front of the lens is called the **projection plane**. The image of the point is located where the projector passes through the projection plane. In Chapter 5, we discuss this process in detail and derive the relevant mathematical formulas.

We must also consider the limited size of the image. As we saw, not all objects can be imaged onto the pinhole camera's film plane. The field of view expresses this limitation. In the synthetic camera, we can move this limitation to the front by placing a **clipping rectangle**, or **clipping window**, in the projection plane (Figure 1.19). This rectangle acts as a window through which a viewer, located at the center of projection, sees the world. Given the location of the center of projection, the location and orientation of the projection plane, and the size of the clipping rectangle, we can determine which objects will appear in the image.

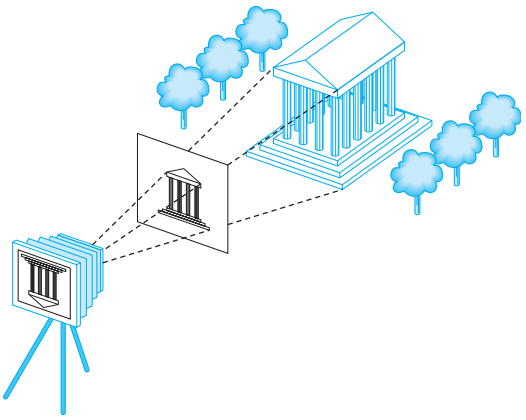
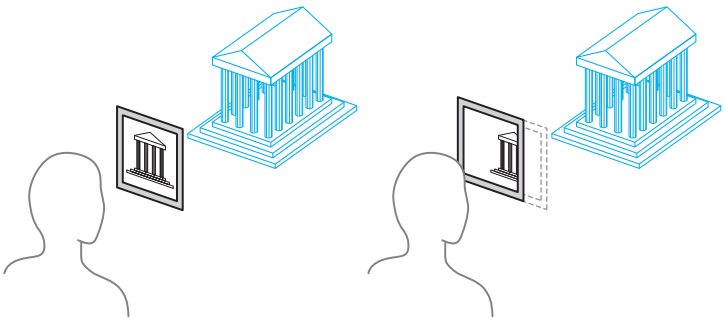


FIGURE 1.18 Imaging with the synthetic camera.



(a) (b)

FIGURE 1.19 Clipping. (a) Window in initial position. (b) Window shifted.

1.6 THE PROGRAMMER'S INTERFACE

There are numerous ways that a user can interact with a graphics system. With completely self-contained packages such as those used in the CAD community, a user develops images through interactions with the display using input devices such as a mouse and a keyboard. In a typical application, such as the painting program in Figure 1.20, the user sees menus and icons that represent possible actions. By clicking on these items, the user guides the software and produces images without having to write programs.

Of course, someone has to develop the code for these applications, and many of us, despite the sophistication of commercial products, still have to write our own graphics application programs (and even enjoy doing so).

The interface between an application program and a graphics system can be specified through a set of functions that resides in a graphics library. These speci-

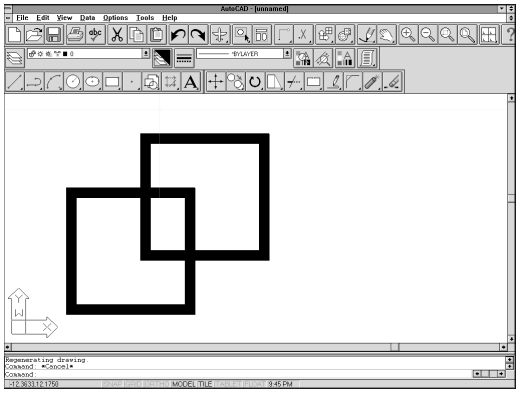


FIGURE 1.20 Interface for a painting program.

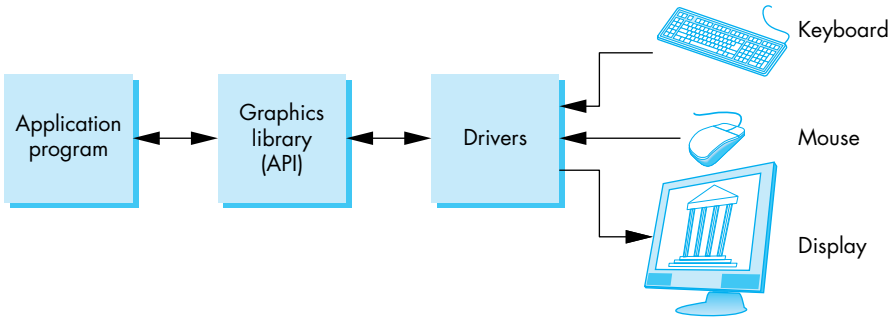


FIGURE 1.21 Application programmer's model of graphics system.

fications are called the **application programming interface** (**API**). The application programmer's model of the system is shown in Figure 1.21. The application programmer sees only the API and is thus shielded from the details of both the hardware and the software implementation of the graphics library. The software **drivers** are responsible for interpreting the output of the API and converting these data to a form that is understood by the particular hardware. From the perspective of the writer of an application program, the functions available through the API should match the conceptual model that the user wishes to employ to specify images. By developing code that uses the API, the application programmer is able to develop applications that can be used with different hardware and software platforms.

1.6.1 The Pen-Plotter Model

Historically, most early graphics systems were two-dimensional systems. The conceptual model that they used is now referred to as the **pen-plotter model**, referring to the output device that was available on these systems. A **pen plotter** (Figure 1.22) produces images by moving a pen held by a gantry, a structure that can move the pen

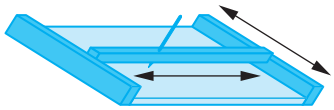


FIGURE 1.22 Pen plotter.

in two orthogonal directions across the paper. The plotter can raise and lower the pen as required to create the desired image. Pen plotters are still in use; they are well suited for drawing large diagrams, such as blueprints. Various APIs—such as LOGO and PostScript—have their origins in this model. The HTML5 canvas upon which we will display the output from WebGL also has its origins in the pen-plotter model. Although they differ from one another, they have a common view of the process of creating an image as being similar to the process of drawing on a pad of paper. The user works on a two-dimensional surface of some size. She moves a pen around on this surface, leaving an image on the paper.

We can describe such a graphics system with two drawing functions:

```
moveto(x, y);
lineto(x, y);
```

Execution of the `moveto` function moves the pen to the location (x, y) on the paper without leaving a mark. The `lineto` function moves the pen to (x, y) and draws a line from the old to the new location of the pen. Once we add a few initialization and termination procedures, as well as the ability to change pens to alter the drawing color or line thickness, we have a simple—but complete—graphics system. Here is a fragment of a simple program in such a system:

```
moveto(0, 0);
lineto(1, 0);
lineto(1, 1);
lineto(0, 1);
lineto(0, 0);
```

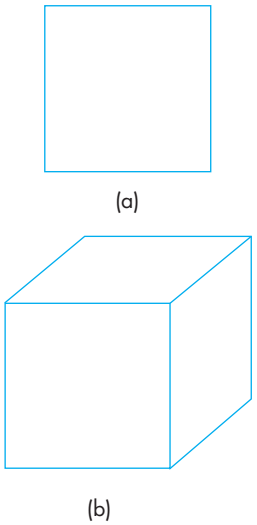


FIGURE 1.23 Output of pen-plotter program for (a) a square, and (b) a projection of a cube.

This fragment would generate the output in Figure 1.23(a). If we added the code

```
moveto(0, 1);
lineto(0.5, 1.866);
lineto(1.5, 1.866);
lineto(1.5, 0.866);
lineto(1, 0);
moveto(1, 1);
lineto(1.5, 1.866);
```

we would have the image of a cube formed by an oblique projection, as is shown in Figure 1.23(b).

For certain applications, such as page layout in the printing industry, systems built on this model work well. For example, the PostScript page description language, a sophisticated extension of these ideas, is a standard for controlling typesetters and printers.

An alternate raster-based (but still limited) two-dimensional model relies on writing pixels directly into a framebuffer. Such a system could be based on a single function of the form

```
writePixel(x, y, color);
```

where `x,y` is the location of the pixel in the framebuffer and `color` gives the color to be written there. Such models are well suited to writing the algorithms for rasterization and processing of digital images.

We are much more interested, however, in the three-dimensional world. The pen-plotter model does not extend well to three-dimensional graphics systems. For example, if we wish to use the pen-plotter model to produce the image of a three-dimensional object on our two-dimensional pad, either by hand or by computer, then we have to figure out where on the page to place two-dimensional points corresponding to points on our three-dimensional object. These two-dimensional points are, as we saw in Section 1.5, the projections of points in three-dimensional space. The mathematical process of determining projections is an application of trigonometry. We develop the mathematics of projection in Chapter 5; understanding projection is crucial to understanding three-dimensional graphics. We prefer, however, to use an API that allows users to work directly in the domain of their problems and to use computers to carry out the details of the projection process automatically, without the users having to make any trigonometric calculations within the application program. That approach should be a boon to users who have difficulty learning to draw various projections on a drafting board or sketching objects in perspective. More important, users can rely on hardware and software implementations of projections within the implementation of the API that are far more efficient than any possible implementation of projections within their programs would be.

Three-dimensional printers are revolutionizing design and manufacturing, allowing the fabrication of items as varied as mechanical parts, art, and biological items constructed from living cells. They illustrate the importance of separating the low-level production of the final piece from the high-level software used for design. At the physical level, they function much like our description of a pen plotter except that rather than depositing ink, they can deposit almost any material. The three-dimensional piece is built up in layers, each of which can be described using our pen-plotter model. However, the design is done in three dimensions with a high-level API that can output a file that is converted into a stack of layers for the printer.

1.6.2 Three-Dimensional APIs

The synthetic-camera model is the basis for a number of popular APIs, including OpenGL and Direct3D. If we are to follow the synthetic-camera model, we need functions in the API to specify the following:

- Objects
- A viewer
- Light sources
- Material properties

Objects are usually defined by sets of vertices. For simple geometric objects—such as line segments, rectangles, and polygons—there is a simple relationship between a list of **vertices**, or positions in space, and the object. For more complex

objects, there may be multiple ways of defining the object from a set of vertices. A circle, for example, can be defined by three points on its circumference, or by its center and one point on the circumference.

Most APIs provide similar sets of primitive objects for the user. These primitives are usually those that can be displayed rapidly on the hardware. The usual sets include points, line segments, and triangles. WebGL programs specify primitives through lists of vertices. The following code fragment shows one way to specify three vertices in JavaScript for use with WebGL:

```
var vertices = [ ];

vertices[0] = [0.0, 0.0, 0.0]; // Vertex A
vertices[1] = [0.0, 1.0, 0.0]; // Vertex B
vertices[2] = [0.0, 0.0, 1.0]; // Vertex C
```

Or we could use

```
var vertices = [ ];

vertices.push([0.0, 0.0, 0.0]); // Vertex A
vertices.push([0.0, 1.0, 0.0]); // Vertex B
vertices.push([0.0, 0.0, 1.0]); // Vertex C
```

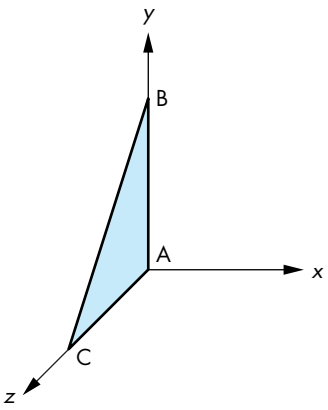


FIGURE 1.24 A triangle.

We could either send this array to the GPU each time that we want it to be displayed or store it on the GPU for later display. Note that these three vertices only give three locations in a three-dimensional space and do not specify the geometric entity that they define. The locations could describe a triangle, as in Figure 1.24, or we could use them to specify two line segments, using the first two locations to specify the first segment and the second and third locations to specify the second segment. We could also use the three points to display three pixels at locations in the framebuffer corresponding to the three vertices. We make this choice in our application by setting a parameter corresponding to the geometric entity we would like these locations to specify. For example, in WebGL we would use `gl.TRIANGLES`, `gl.LINE_STRIP`, or `gl.POINTS` for the three possibilities we just described. Although we are not yet ready to describe all the details of how we accomplish this task, we can note that, regardless of which geometric entity we wish our vertices to specify, we are specifying the geometry and leaving it to the graphics system to determine which pixels to color in the framebuffer.

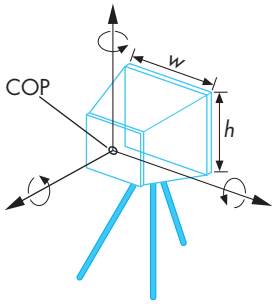


FIGURE 1.25 Camera specification.

Some APIs let the user work directly in the framebuffer by providing functions that read and write pixels. Additionally, some APIs provide curves and surfaces as primitives; often, however, these types are approximated by a series of simpler primitives within the application program. WebGL provides access to the framebuffer through texture maps.

We can define a viewer or camera in a variety of ways. Available APIs differ both in how much flexibility they provide in camera selection and in how many different methods they allow. If we look at the camera in Figure 1.25, we can identify four types of necessary specifications:

1. **Position** The camera location usually is given by the position of the center of the lens, which is the center of projection (COP).

2. **Orientation** Once we have positioned the camera, we can place a camera coordinate system with its origin at the center of projection. We can then rotate the camera independently around the three axes of this system.
3. **Focal length** The focal length of the lens determines the size of the image on the film plane or, equivalently, the portion of the world the camera sees.
4. **Film plane** The back of the camera has a height and a width. On the bellows camera, and in some APIs, the orientation of the back of the camera can be adjusted independently of the orientation of the lens.

These specifications can be satisfied in various ways. One way to develop the specifications for the camera location and orientation is through a series of coordinate-system transformations. These transformations convert object positions represented in a coordinate system that specifies object vertices to object positions in a coordinate system centered at the COP. This approach is useful, both for doing implementation and for getting the full set of views that a flexible camera can provide. We use this approach extensively, starting in Chapter 5.

Having many parameters to adjust, however, can also make it difficult to get a desired image. Part of the problem lies with the synthetic-camera model. Classical viewing techniques, such as are used in architecture, stress the *relationship* between the object and the viewer, rather than the *independence* that the synthetic-camera model emphasizes. Thus, the classical two-point perspective of a cube in Figure 1.26 is a *two-point* perspective because of a particular relationship between the viewer and the planes of the cube (see Exercise 1.7). Although the WebGL API allows us to set transformations with complete freedom, we will provide additional helpful functions. For example, consider the two function calls

```
lookAt(cop, at, up);
perspective(fieldOfView, aspectRatio, near, far);
```

The first function call points the camera from the center of projection toward a desired point (the *at* point), with a specified *up* direction for the camera. The second selects a lens for a perspective view (the *field of view*) and how much of the world the camera should image (the *aspect ratio* and the *near* and *far* distances). These functions are built using the WebGL API but are so useful that we will add them to our libraries.

However, none of the APIs built on the synthetic-camera model provide functions for directly specifying a desired relationship between the camera and an object.

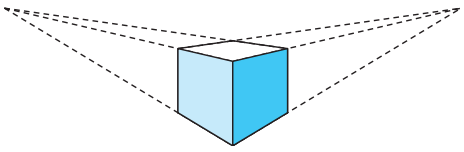


FIGURE 1.26 Two-point perspective of a cube.

Light sources are defined by their location, strength, color, and directionality. With WebGL, we can specify these parameters for each source. Material properties are characteristics, or attributes, of the objects, and such properties can also be specified through WebGL at the time that each object is defined. As we shall see in Chapter 6, we will be able to implement different models of light–material interactions with our shaders.

1.6.3 A Sequence of Images

In Chapter 2, we begin our detailed discussion of the WebGL API that we will use throughout this book. The images defined by your WebGL programs will be formed automatically by the hardware and software implementation of the image formation process.

Here we look at a sequence of images that shows what we can create using the WebGL API. We present these images as an increasingly more complex series of renderings of the same objects. The sequence not only loosely follows the order in which we present related topics but also reflects how graphics systems have developed over the past 40 years.

Color Plate 1 shows an image of an artist's creation of a sunlike object. Color Plate 2 shows the object rendered using only line segments. Although the object consists of many parts, and although the programmer may have used sophisticated data structures to model each part and the relationships among the parts, the rendered object shows only the outlines of the parts. This type of image is known as a **wireframe** image because we can see only the edges of surfaces: Such an image would be produced if the objects were constructed with stiff wires that formed a frame with no solid material between the edges. Before raster graphics systems became available, wireframe images were the only type of computer-generated images that we could produce.

In Color Plate 3, the same object has been rendered with flat polygons. Certain surfaces are not visible because there is a solid surface between them and the viewer; these surfaces have been removed by a hidden-surface-removal (HSR) algorithm. Most raster systems can fill the interior of polygons with a solid color in not much more time than they can render a wireframe image. Although the objects are three-dimensional, each surface is displayed in a single color, and the image fails to show the three-dimensional shapes of the objects. Early raster systems could produce images of this form.

In Chapters 2 and 4, we show you how to generate images composed of simple geometric objects—points, line segments, and triangles. In Chapters 4 and 5, you will learn how to transform objects in three dimensions and how to obtain a desired three-dimensional view of a model, with hidden surfaces removed.

Color Plate 4 illustrates smooth shading of the triangles that approximate the object; it shows that the object is three-dimensional and gives the appearance of a smooth surface. We develop shading models that are supported by WebGL in Chapter 6. These shading models are also supported in the hardware of most recent workstations; generating the shaded image on one of these systems takes approximately the same amount of time as does generating a wireframe image.

Color Plate 5 shows a more sophisticated wireframe model constructed using NURBS surfaces, which we introduce in Chapter 11. Such surfaces give the application programmer great flexibility in the design process but are ultimately rendered using line segments and polygons.

In Color Plates 6 and 7, we add surface texture to our object; texture is one of the effects that we discuss in Chapter 7. All recent graphics processors support texture mapping in hardware, so rendering of a texture-mapped image requires little additional time. In Color Plate 6, we use a technique called *bump mapping* that gives the appearance of a rough surface even though we render the same flat polygons as in the other examples. Color Plate 7 shows an *environment map* applied to the surface of the object, which gives the surface the appearance of a mirror. These techniques will be discussed in detail in Chapter 7.

Color Plate 8 shows a small area of the rendering of the object using an environment map. The image on the left shows the jagged artifacts known as aliasing errors that are due to the discrete nature of the framebuffer. The image on the right has been rendered using a smoothing or antialiasing method that we shall study in Chapters 7 and 8.

Not only do these images show what is possible with available hardware and a good API, but they are also simple to generate, as we shall see in subsequent chapters. In addition, just as the images show incremental changes in the renderings, the programs are incrementally different from one another.

1.6.4 The Modeling–Rendering Paradigm

Both conceptually and in practice it is often helpful to separate the modeling of the scene from the production of the image, or the **rendering** of the scene. Hence, we can look at image formation as the two-step process shown in Figure 1.27. Although the tasks are the same as those we have been discussing, this block diagram suggests that we might implement the modeler and the renderer with different software and hardware.

This paradigm first became important in CAD and animation where, due to the limitations of the hardware, the design or modeling of objects needed to be separated from the production, or the rendering, of the scene.

For example, consider the production of a single frame in an animation. We first want to design and position our objects. This step is highly interactive, and usually does not require all the detail in the objects nor to render the scene in great detail incorporating effects such as reflections and shadows. Consequently, we can carry out this step interactively with standard graphics hardware. Once we have designed

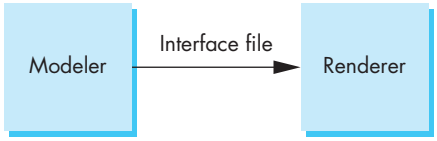


FIGURE 1.27 The modeling–rendering paradigm.

the scene, we want to render it, adding light sources, material properties, and a variety of other detailed effects, to form a production-quality image. This step can require a tremendous amount of computation, so we might prefer to use a render farm: a cluster of computers configured for numerical computing.

The interface between the modeler and renderer can be as simple as a file produced by the modeler that describes the objects and that contains additional information important only to the renderer, such as light sources, viewer location, and material properties. Pixar's RenderMan Interface is based on this approach and uses a file format that allows modelers to pass models to the renderer. One of the other advantages of this approach is that it allows us to develop modelers that, although they use the same renderer, are custom-tailored to particular applications. Likewise, different renderers can take as input the same interface file.

Although it is still used in the movie industry where we need to produce high-resolution images with many effects, with modern GPUs most applications do not need to separate modeling and rendering in quite the way we just described. However, there are a few places where will see this paradigm, albeit in a slightly different form. This paradigm has become popular as a method for generating images for multiplayer computer games. Models, including the geometric objects, lights, cameras, and material properties, are placed in a data structure called a **scene graph** that is passed to a renderer or game engine. We shall examine scene graphs in Chapter 9. As we shall see in Chapter 2, the standard way in which we use the GPU is to first form our geometry of objects in the CPU, and then send these data to the GPU for rendering. Hence, in a limited sense, the CPU is the modeler and the GPU is the renderer.

1.7 GRAPHICS ARCHITECTURES

On one side of the API is the application program. On the other is some combination of hardware and software that implements the functionality of the API. Researchers have taken various approaches to developing architectures to support graphics APIs.

Early graphics systems used general-purpose computers with the standard von Neumann architecture. Such computers are characterized by a single processing unit that processes a single instruction at a time. A simple model of these early graphics systems is shown in Figure 1.28. The display in these systems was based on a calligraphic CRT display that included the necessary circuitry to generate a line segment connecting two points. The job of the host computer was to run the application program and to compute the endpoints of the line segments in the image (in units of the

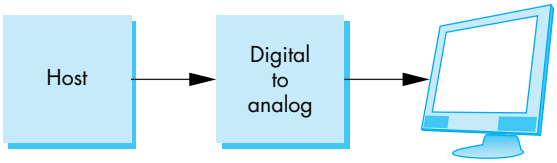


FIGURE 1.28 Early graphics system.

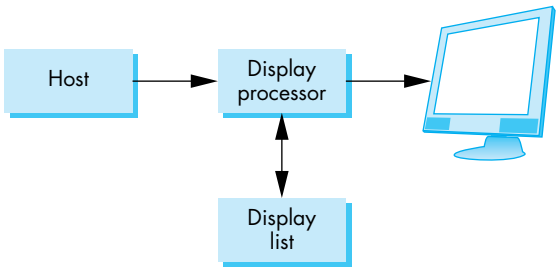


FIGURE 1.29 Display-processor architecture.

display). This information had to be sent to the display at a rate high enough to avoid flicker. In the early days of computer graphics, computers were so slow that refreshing even simple images, containing a few hundred line segments, would burden an expensive computer.

1.7.1 Display Processors

The earliest attempts to build special-purpose graphics systems were concerned primarily with relieving the general-purpose computer from the task of refreshing the display continuously. These **display processors** had conventional architectures (Figure 1.29) but included instructions to display primitives on the CRT. The main advantage of the display processor was that the instructions to generate the image could be assembled once in the host and sent to the display processor, where they were stored in the display processor's own memory as a **display list**, or **display file**. The display processor would then repetitively execute the program in the display list, at a rate sufficient to avoid flicker, independently of the host, thus freeing the host for other tasks. This architecture has become closely associated with the client–server architectures that are used in most systems.

1.7.2 Pipeline Architectures

The major advances in graphics architectures closely parallel the advances in workstations. In both cases, the ability to create special-purpose VLSI chips was the key enabling technological development. In addition, the availability of inexpensive solid-state memory led to the universality of raster displays. For computer graphics applications, the most important use of custom VLSI circuits has been in creating **pipeline** architectures.

The concept of pipelining is illustrated in Figure 1.30 for a simple arithmetic calculation. In our pipeline, there is an adder and a multiplier. If we use this configuration to compute $a + (b * c)$, the calculation takes one multiplication and one addition—the same amount of work required if we use a single processor to carry out both operations. However, suppose that we have to carry out the same computation with many values of a, b, and c. Now, the multiplier can pass on the results of its calculation to the adder and can start its next multiplication while the adder carries out the second step of the calculation on the first set of data. Hence, whereas it

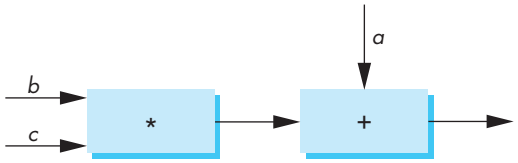


FIGURE 1.30 Arithmetic pipeline.

takes the same amount of time to calculate the results for any one set of data, when we are working on two sets of data at one time, our total time for calculation is shortened markedly. Here the rate at which data flow through the system, the **throughput** of the system, has been doubled. Note that as we add more boxes to a pipeline, it takes more time for a single datum to pass through the system. This time is called the **latency** of the system; we must balance it against increased throughput in evaluating the performance of a pipeline.

We can construct pipelines for more complex arithmetic calculations that will afford even greater increases in throughput. Of course, there is no point in building a pipeline unless we will do the same operation on many data sets. But that is just what we do in computer graphics, where large sets of vertices and pixels must be processed in the same manner.

1.7.3 The Graphics Pipeline

We start with a set of objects. Each object comprises a set of graphical primitives. Each primitive comprises a set of vertices. We can think of the collection of primitive types and vertices as defining the **geometry** of the scene. In a complex scene, there may be thousands—even millions—of vertices that define the objects. We must process all these vertices in a similar manner to form an image in the framebuffer. If we think in terms of processing the geometry of our objects to obtain an image, we can employ the block diagram in Figure 1.31, which shows the four major steps in the imaging process:

1. Vertex processing
2. Clipping and primitive assembly
3. Rasterization
4. Fragment processing

In subsequent chapters, we discuss the details of these steps. Here we are content to preview these steps and show that they can be pipelined.

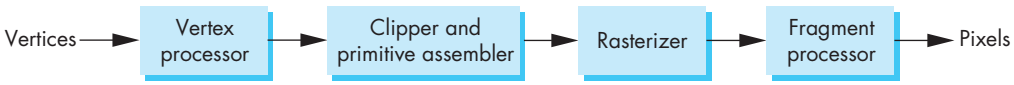


FIGURE 1.31 Geometric pipeline.

1.7.4 Vertex Processing

In the first block of our pipeline, each vertex is processed independently. The major function of this block is to carry out coordinate transformations. It can also compute a color for each vertex and change any other attributes of the vertex.

Many of the steps in the imaging process can be viewed as transformations between representations of objects in different coordinate systems. For example, in our discussion of the synthetic camera, we observed that a major part of viewing is to convert to a representation of objects from the system in which they were defined to a representation in terms of the coordinate system of the camera. A further example of a transformation arises when we finally put our images onto the output device. The internal representation of objects—whether in the camera coordinate system or perhaps in a system used by the graphics software—eventually must be represented in terms of the coordinate system of the display. We can represent each change of coordinate systems by a matrix. We can represent successive changes in coordinate systems by multiplying, or **concatenating**, the individual matrices into a single matrix. In Chapter 4, we examine these operations in detail. Because multiplying one matrix by another matrix yields a third matrix, a sequence of transformations is an obvious candidate for a pipeline architecture. In addition, because the matrices that we use in computer graphics will always be small (4×4), we have the opportunity to use parallelism within the transformation blocks in the pipeline.

Eventually, after multiple stages of transformation, the geometry is transformed by a projection transformation. We shall see in Chapter 5 that we can implement this step using 4×4 matrices, and thus projection fits in the pipeline. In general, we want to keep three-dimensional information as long as possible, as objects pass through the pipeline. Consequently, the projection transformation is somewhat more general than the projections in Section 1.5. In addition to retaining three-dimensional information, there is a variety of projections that we can implement. We shall see these projections in Chapter 5.

The assignment of vertex colors can be as simple as the program specifying a color or as complex as the computation of a color from a physically realistic lighting model that incorporates the surface properties of the object and the characteristic light sources in the scene. We shall discuss lighting models in Chapter 6.

1.7.5 Clipping and Primitive Assembly

The second fundamental block in the implementation of the standard graphics pipeline is for clipping and primitive assembly. We must do clipping because of the limitation that no imaging system can see the whole world at once. The human retina has a limited size corresponding to an approximately 90-degree field of view. Cameras have film of limited size, and we can adjust their fields of view by selecting different lenses.

We obtain the equivalent property in the synthetic camera by considering a **clipping volume**, such as the pyramid in front of the lens in Figure 1.18. The projections of objects within this volume appear in the image. Those that are outside do not and

are said to be clipped out. Objects that straddle the edges of the clipping volume are partly visible in the image. Efficient clipping algorithms are developed in Chapter 8.

Clipping must be done on a primitive-by-primitive basis rather than on a vertex-by-vertex basis. Thus, within this stage of the pipeline, we must assemble sets of vertices into primitives, such as line segments and polygons, before clipping can take place. Consequently, the output of this stage is a set of primitives whose projections can appear in the image.

1.7.6 Rasterization

The primitives that emerge from the clipper are still represented in terms of their vertices and must be converted to pixels in the framebuffer. For example, if three vertices specify a triangle with a solid color, the rasterizer must determine which pixels in the framebuffer are inside the polygon. We discuss this rasterization (or scan conversion) process for line segments and polygons in Chapter 8. The output of the rasterizer is a set of **fragments** for each primitive. A fragment can be thought of as a potential pixel that carries with it information, including its color and location, that is used to update the corresponding pixel in the framebuffer. Fragments can also carry along depth information that allows later stages to determine whether a particular fragment lies behind other previously rasterized fragments for a given pixel.

1.7.7 Fragment Processing

The final block in our pipeline takes in the fragments generated by the rasterizer and updates the pixels in the framebuffer. If the application generated three-dimensional data, some fragments may not be visible because the surfaces that they define are behind other surfaces. The color of a fragment may be altered by texture mapping or bump mapping, as in Color Plates 6 and 7. The color of the pixel that corresponds to a fragment can also be read from the framebuffer and blended with the fragment's color to create translucent effects. These effects will be covered in Chapter 7.

1.8 PROGRAMMABLE PIPELINES

Graphics architectures have gone through multiple design cycles in which the importance of special-purpose hardware relative to standard CPUs has gone back and forth. However, the importance of the pipeline architecture has remained regardless of this cycle. None of the other approaches—ray tracing, radiosity, photon mapping—can achieve real-time behavior, that is, the ability to render complex dynamic scenes so that the viewer sees the display without defects. However, the term *real-time* is becoming increasingly difficult to define as graphics hardware improves. Although some approaches such as ray tracing can come close to real time, none can achieve the performance of pipeline architectures with simple application programs and simple GPU programs. Hence, the commodity graphics market is dominated by graphics cards that have pipelines built into the graphics processing unit. All of these com-

modity cards implement the pipeline that we have just described, albeit with more options, many of which we shall discuss in later chapters.

For many years, these pipeline architectures had a fixed functionality. Although the application program could set many parameters, the basic operations available within the pipeline were fixed. Recently, there has been a major advance in pipeline architectures. Both the vertex processor and the fragment processor are now programmable by the application programmer. One of the most exciting aspects of this advance is that many of the techniques that formerly could not be done in real time because they were not part of the fixed-function pipeline can now be done in real time. Bump mapping, which we illustrated in Color Plate 6, is but one example of an algorithm that is now programmable but formerly could only be done off-line.

Vertex shaders can alter the location or color of each vertex as it flows through the pipeline. Thus, we can implement a variety of light–material models or create new kinds of projections. Fragment shaders allow us to use textures in new ways and to implement other parts of the pipeline, such as lighting, on a per-fragment basis rather than per-vertex.

Programmability is now available at every level, including in handheld devices such as smart phones and tablets that include a color screen and a touch interface. Additionally, the speed and parallelism in programmable GPUs make them suitable for carrying out high-performance computing that does not involve graphics.

The latest versions of OpenGL have responded to these advances first by adding more and more programmability and options to the standard. Additionally, variants of OpenGL, like WebGL, have been created from the standard to better match the capabilities of the devices on which the API needs to run. For example, in mobile (i.e., handheld or embedded) devices, a version of OpenGL called OpenGL ES (where "ES" is an abbreviation for *embedded system*) is available on most smart phones and tablets. It has a reduced number of function calls but many of the same capabilities. Strictly speaking, WebGL is a version of OpenGL ES that works in web browsers, and may use OpenGL ES internally as its rendering engine. For those who started with the original fixed-function pipeline, it may take a little more time for our first programs, but the rewards will be significant.

1.9 PERFORMANCE CHARACTERISTICS

There are two fundamentally different types of processing in our architecture. At the front end, there is geometric processing, based on processing vertices through the various transformations: vertex shading, clipping, and primitive assembly. This processing is ideally suited for pipelining, and it usually involves floating-point calculations. The geometry engine developed by Silicon Graphics, Inc. (SGI) was a VLSI implementation for many of these operations in a special-purpose chip that became the basis for a series of fast graphics workstations. Later, floating-point accelerator chips put 4×4 matrix transformation units on the chip. Nowadays, graphics workstations and commodity graphics cards use graphics processing units (GPUs) that perform most of the graphics operations at the chip level. Pipeline architectures are the dominant type of high-performance system.

Beginning with rasterization and including many features that we discuss later, processing involves a direct manipulation of bits in the framebuffer. This back-end processing is fundamentally different from front-end processing, and, until recently, was implemented most effectively using architectures that had the ability to move blocks of bits quickly. The overall performance of a system was (and still is) characterized by how fast we can move geometric entities through the pipeline and by how many pixels per second we can alter in the framebuffer. Consequently, the fastest graphics workstations were characterized by geometric pipelines at the front ends and parallel bit processors at the back ends.

Until about 10 years ago, there was a clear distinction between front- and back-end processing and there were different components and boards dedicated to each. Now commodity graphics cards use GPUs that contain the entire pipeline within a single chip. The latest cards implement the entire pipeline using floating-point arithmetic and have floating-point framebuffers. These GPUs are so powerful that even the highest-level systems—systems that incorporate multiple pipelines—use these processors.

Since the operations performed when processing vertices and fragments are very similar, modern GPUs are **unified shading engines** that carry out both vertex and fragment shading concurrently. This processing might be limited to a single core in a small GPU (like in a mobile phone), or hundreds of processors in high-performance GPUs and gaming systems. This flexibility in processing allows GPUs to use their resources optimally regardless of an application's requirements.

Pipeline architectures dominate the graphics field, especially where real-time performance is of importance. Our presentation has made a case for using such an architecture to implement the hardware of a graphics system. Commodity graphics cards incorporate the pipeline within their GPUs. Today, even mobile phones can render millions of shaded texture-mapped triangles per second. However, we can also make as strong a case for pipelining being the basis of a complete software implementation of an API.

1.10 OPENGL VERSIONS AND WEBGL

So far we have not carefully distinguished between OpenGL and WebGL, other than to say WebGL is a version of OpenGL that runs in most modern browsers. To get a better understanding of the differences between various versions of OpenGL, it will help if we take a brief look at the history of OpenGL.

OpenGL is derived from IRIS GL, which had been developed as an API for SGI's revolutionary VLSI pipelined graphics workstations. IRIS GL was close enough to the hardware to allow applications to run efficiently and at the same time provided application developers with a high-level interface to graphics capabilities of the workstations. Because it was designed for a particular architecture and operating system, it was able to include both input and windowing functions in the API.

When OpenGL was developed as a cross-platform rendering API, the input and windowing functionality were eliminated. Consequently, although an application has

to be recompiled for each architecture, the rendering part of the application code should be identical.

The first version of OpenGL (Version 1.0) was released in 1992. The API was based on the graphics pipeline architecture, but there was almost no ability to access the hardware directly. Early versions focused on **immediate mode graphics** in which graphic primitives were specified in the application and then immediately passed down the pipeline for display. Consequently, there was no memory of the primitives and their redisplay required them to be resent through the pipeline. New features were added incrementally in Versions 1.1–1.5. Other new features became available as extensions and, although not part of the standard, were supported on particular hardware. Version 2.0 was released in 2004 and Version 2.1 in 2006. Looking back, Version 2.0 was the key advance because it introduced the OpenGL Shading Language, which allowed application programmers to write their own shaders and exploit the power of GPUs. In particular, with better GPUs and more memory, geometry information could be stored or retained and **retained mode graphics** became increasingly more important.

Through Version 3.0 (2008), all versions of OpenGL were backward compatible so code developed on earlier versions was guaranteed to run on later versions. Hence, as new more complex versions were released, developers were forced to support older and less useful features. Version 3.0 announced that starting with Version 3.1, backward compatibility would no longer be provided by all implementations and, in particular, immediate mode was to be deprecated. Successive versions introduced more and more features. We are presently at Version 4.4. Although OpenGL has been ported to some other languages, the vast majority of applications are written in C and C++.

At the same time OpenGL was providing more and more capabilities to exploit the advances in GPU technology, there was an increasing demand for a simpler version that would run in embedded systems, smart phones, and other handheld devices. OpenGL ES 1.1 was released in 2008 and was based on OpenGL Version 1.5. OpenGL ES 2.0 is based on OpenGL Version 2.0 and supports only shader-based applications and not immediate mode. OpenGL ES Version 3.0 was released in 2013. We can now develop three-dimensional interactive applications for devices like smart phones whose hardware now contains GPUs.

Both the full OpenGL and the simpler OpenGL ES are designed to run locally and cannot take advantage of the Web. Thus, if we want to run an OpenGL application that we find on the Internet, we must download the binary file if it runs on the same architecture or download the source file and recompile. Even if we can run an application remotely and see its output on a local window, the application cannot use the GPU on the local system. WebGL is a JavaScript interface to OpenGL ES 2.0 that runs in modern web browsers. Thus, a user can visit the URL of a WebGL application on a remote system and, like other web applications, the program will run on the local system and make use of the local graphics hardware. WebGL and OpenGL ES 2.0 are fully shader based. Although WebGL does not have all the features of the latest versions of OpenGL, all the basic properties and capabilities of OpenGL are in WebGL. Consequently, we can almost always refer to OpenGL or WebGL without concern for the differences.

SUMMARY AND NOTES

In this chapter, we have set the stage for our top-down development of computer graphics. We presented the overall picture so that you can proceed to writing graphics application programs in the next chapter without feeling that you are working in a vacuum.

We have stressed that computer graphics is a method of image formation that should be related to classical methods of image formation—in particular, to image formation in optical systems such as cameras. In addition to explaining the pinhole camera, we have introduced the human visual system; both are examples of imaging systems.

We described multiple image formation paradigms, each of which has applicability in computer graphics. The synthetic-camera model has two important consequences for computer graphics. First, it stresses the independence of the objects and the viewer—a distinction that leads to a good way of organizing the functions that will be in a graphics library. Second, it leads to the notion of a pipeline architecture, in which each of the various stages in the pipeline performs distinct operations on geometric entities and then passes on the transformed objects to the next stage.

We also introduced the idea of tracing rays of light to obtain an image. This paradigm is especially useful in understanding the interaction between light and materials that is essential to physical image formation. Because ray tracing and other physically based strategies cannot render scenes in real time, we defer further discussion of them until Chapter 12.

The modeling–rendering paradigm is becoming increasingly important. A standard graphics workstation can generate millions of line segments or polygons per second at a resolution exceeding 2048×1546 pixels. Such a workstation can shade the polygons using a simple shading model and can display only visible surfaces at the same rate. However, realistic images may require a resolution of up to 6000×4000 pixels to match the resolution of film and may use light and material effects that cannot be implemented in real time. Even as the power of available hardware and software continues to grow, modeling and rendering have such different goals that we can expect the distinction between modeling and rendering to survive.

Our next step will be to explore the application side of graphics programming. We use the WebGL API, which is powerful, is supported on modern browsers, and has a distinct architecture that will allow us to use it to understand how computer graphics works, from an application program to a final image on a display.

SUGGESTED READINGS

There are many excellent graphics textbooks. The book by Newman and Sproull [New73] was the first to take the modern viewpoint of using the synthetic-camera model. The various editions of Foley et al. [Fol90, Fol94] and Hughes et al. [Hug13] have been the standard references. Other good texts include Hearn and Baker [Hea11], Hill [Hil07], and Shirley [Shi09].

Good general references include *Computer Graphics*, the quarterly journal of SIGGRAPH (the Association for Computing Machinery's Special Interest Group on Graphics), *IEEE Computer Graphics and Applications*, and *Visual Computer*. The proceedings of the annual SIGGRAPH conference include the latest techniques. These proceedings were formerly published as the summer issue of *Computer Graphics*. Now they are published as an issue of the *ACM Transactions on Graphics* and are available on DVD. Of particular interest to newcomers to the field are the state-of-the-art animations available from SIGGRAPH and the notes from tutorial courses taught at that conference, both of which are now available on DVD or in ACM's digital library.

Sutherland's doctoral dissertation, published as *Sketchpad: A Man–Machine Graphical Communication System* [Sut63], was probably the seminal paper in the development of interactive computer graphics. Sutherland was the first person to realize the power of the new paradigm in which humans interacted with images on a CRT display. Videotape copies of film of his original work are still available.

Tufte's books [Tuf83, Tuf90, Tuf97] show the importance of good visual design and contain considerable historical information on the development of graphics. The article by Carlbom and Paciorek [Car78] provides a good discussion of some of the relationships between classical viewing, as used in fields such as architecture, and viewing by computer.

Many books describe the human visual system. Pratt [Pra78] gives a good short discussion for working with raster displays. Also see Glassner [Gla95], Wyszecki and Stiles [Wys82], and Hall [Hal89].

EXERCISES

1.1 In representing a picture, one-bit- and eight-bit-deep frame buffers allow only two and 256 colors, respectively. How many bits does a full-color system allow?

1.2 In computer graphics, objects such as spheres are usually approximated by simpler objects constructed from flat polygons (polyhedra). Using lines of longitude and latitude, define a set of simple polygons that approximate a sphere centered at the origin. Can you use only quadrilaterals or only triangles?

1.3 A different method of approximating a sphere starts with a regular tetrahedron, which is constructed from four triangles. Find its vertices, assuming that it is centered at the origin and has one vertex on the y-axis. Derive an algorithm for obtaining increasingly closer approximations to a unit sphere, based on subdividing the faces of the tetrahedron.

1.4 Consider the clipping of a point in two dimensions against a circular clipping window. Show that you require only the coordinate of the point and the centre and radius of the circle to determine whether the point is not clipped or is clipped out completely.

1.5 For a line segment, show that clipping against the top of the clipping rectangle can be done independently of the clipping against the other sides. Use this result to show that a clipper can be implemented as a pipeline of four simpler clippers.

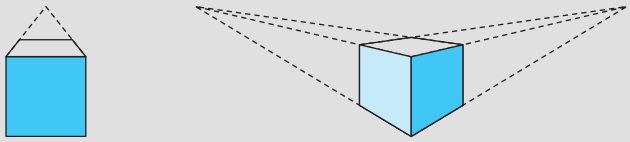


FIGURE 1.32 Perspective views of a cube.

1.6 Extend Exercise 1.4 to clipping against a three-dimensional sphere.

1.7 Consider the perspective views of the cube shown in Figure 1.32. The one on the left is called a *one-point perspective* because parallel lines in one direction of the cube—along the sides of the top—converge to a *vanishing point* in the image. In contrast, the image on the right is a *two-point perspective*. Characterize the particular relationship between the viewer, or a simple camera, and the cube that determines why one is a two-point perspective and the other a one-point perspective.

1.8 The memory in a frame buffer must be fast enough to allow the display to be refreshed at a rate sufficiently high to avoid flicker. A typical workstation display can have a resolution of 1024×1024 pixels. If it is refreshed 60 times per second, how fast must the memory be? That is, how much time can we take to read one pixel from memory? What is this number for a 600×800 display that operates at 50 Hz but is interlaced?

1.9 Movies are generally produced on 35-mm film that has a resolution of approximately 2000×3000 pixels. What implication does this resolution have for producing animated images for computers with resolution 600×800 pixels as compared with film?

1.10 Consider the design of a two-dimensional graphical API for a specific application, such as for VLSI design. List all the primitives and attributes that you would include in your system.

1.11 In a typical shadow-mask CRT, if we want to have a smooth display, the width of a pixel must be about three times the width of a triad. Assume that a monitor displays 600×800 pixels, has a CRT diameter of 50 cm, and has a CRT depth of 30 cm. Estimate the spacing between holes in the shadow mask.

1.12 An interesting exercise that should help you understand how rapidly graphics performance has improved is to go to the websites of some of the GPU manufacturers, such as NVIDIA, AMD, and Intel, and look at the specifications for their products. Often the specs for older cards and GPUs are still there. How rapidly has geometric performance improved? What about pixel processing? How has the cost per rendered triangle decreased?

CHAPTER 2

GRAPHICS PROGRAMMING

Our approach to computer graphics is programming oriented. Consequently, we want you to get started programming graphics as soon as possible. To this end, we will introduce a minimal application programming interface (API). This API will be sufficient to allow you to program many interesting two- and three-dimensional problems and to familiarize you with the basic graphics concepts.

We regard two-dimensional graphics as a special case of three-dimensional graphics. This perspective allows us to get started, even though we will touch on three-dimensional concepts lightly in this chapter. Our two-dimensional code will execute without modification on a three-dimensional system.

Our development will use a simple but informative problem: the Sierpinski gasket. It shows how we can generate an interesting and, to many people, unexpectedly sophisticated image using only a handful of graphics functions. We use WebGL as our API, but our discussion of the underlying concepts is broad enough to encompass most modern systems. In particular, our JavaScript code, which is required for WebGL, can easily be converted to C or C++ code for use with desktop OpenGL. The functionality that we introduce in this chapter is sufficient to allow you to write basic two- and three-dimensional programs that do not require user interaction.

2.1 THE SIERPINSKI GASKET

We will use as a sample problem the drawing of the Sierpinski gasket: an interesting shape that has a long history and is of interest in areas such as fractal geometry. The Sierpinski gasket is an object that can be defined recursively and randomly; in the limit, however, it has properties that are not at all random. We start with a two-dimensional version, but as we will see in Section 2.10, the three-dimensional version is almost identical.

Suppose that we start with three points in space. As long as the points are not collinear, they are the vertices of a unique triangle and also define a unique plane. We assume that this plane is the plane $z = 0$ and that these points, as specified in

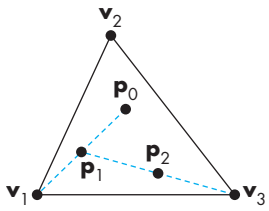


FIGURE 2.1 Generation of the Sierpinski gasket.

some convenient coordinate system,[1] are $(x_1, y_1, 0)$, $(x_2, y_2, 0)$, and $(x_3, y_3, 0)$. The construction proceeds as follows:

1. Pick an initial point $\mathbf{p} = (x, y, 0)$ at random inside the triangle.
2. Select one of the three vertices at random.
3. Find the point $\mathbf{q}$ halfway between $\mathbf{p}$ and the randomly selected vertex.
4. Display $\mathbf{q}$ by putting some sort of marker, such as a small circle, at the corresponding location on the display.
5. Replace $\mathbf{p}$ with $\mathbf{q}$.
6. Return to step 2.

Thus, each time that we generate a new point, we display it on the output device. This process is illustrated in Figure 2.1, where $\mathbf{p}_0$ is the initial point, and $\mathbf{p}_1$ and $\mathbf{p}_2$ are the first two points generated by our algorithm.

Before we develop the program, you might try to determine what the resulting image will be. Try to construct it on paper; you might be surprised by your results.

A possible form for our graphics program might be this:

```
function sierpinski()
{
  initialize_the_system();
  p = find_initial_point();

  for (some_number_of_points) {
    q = generate_a_point(p);
    display_the_point(q);
    p = q;
  }

  cleanup();
}
```

This form can be converted into a real program fairly easily. However, even at this level of abstraction, we can see two other alternatives. Consider the pseudocode

```
function sierpinski()
{
  initialize_the_system();
  p = find_initial_point();

  for (some_number_of_points) {
    q = generate_a_point(p);
    store_the_point(q);
    p = q;
  }
```

1. In Chapter 4, we expand the concept of a coordinate system to the more general formulation of a *frame*.

```
  display_all_points();
  cleanup();
}
```

In this algorithm, we compute all the points first and put them into an array or some other data structure. We then display all the points through a single function call. This approach avoids the overhead of sending small amounts of data to the graphics processor for each point we generate at the cost having to store all the data. The strategy used in the first algorithm is known as **immediate mode graphics** and, until recently, was the standard method for displaying graphics, especially where interactive performance was needed. One consequence of immediate mode is that there is no memory of the geometric data. With our first example, if we want to display the points again, we would have to go through the entire creation and display process a second time.

In our second algorithm, because the data are stored in a data structure, we can redisplay the data, perhaps with some changes such as altering the color or changing the size of a displayed point, by resending the array without regenerating the points. The method of operation is known as **retained mode graphics** and goes back to some of the earliest special-purpose graphics display hardware. The architecture of modern graphics systems that employ a GPU leads to a third version of our program.

Our second approach has one major flaw. Suppose that we wish to redisplay the same objects in a different manner, as we might in an animation. The geometry of the objects is unchanged but the objects may be moving. Displaying all the points as we just outlined involves sending the data from the CPU to the GPU each time that we wish to display the objects in a new position. For large amounts of data, this data transfer is the major bottleneck in the display process. Consider the following alternative scheme:

```
function sierpinski()
{
  initialize_the_system();
  p = find_initial_point();

  for (some_number_of_points) {
    q = generate_a_point(p);
    store_the_point(q);
    p = q;
  }

  send_all_points_to_GPU();
  display_data_on_GPU();
  cleanup();
}
```

As before, we place data in an array, but now we have broken the display process into two parts: storing the data on the GPU and displaying the data that have been stored. If we only have to display our data once, there is no advantage over our previous method; but if we want to animate the display, our data are already on the GPU and

redisplay does not require any additional data transfer, only a simple function call that alters the location of some spatial data describing the objects that have moved.

Although our final WebGL program will have a slightly different organization, we will follow this third strategy. We develop the full program in stages. First, we concentrate on the core: generating and displaying points. We must answer two questions:

- How do we represent points in space?
- Should we use two-dimensional, three-dimensional, or some other representation?

Once we answer these questions, we will be able to place our geometry on the GPU in a form that can be rendered. Then, we will be able to address how we view our objects using the power of programmable shaders.

2.2 PROGRAMMING TWO-DIMENSIONAL APPLICATIONS

For two-dimensional applications, such as the Sierpinski gasket, we could use a pen-plotter API, but such an approach would limit us. Instead, we choose to start with a three-dimensional world; we regard two-dimensional systems, such as the one on which we will produce our image, as special cases. Mathematically, we view the two-dimensional plane, or a simple two-dimensional curved surface, as a subspace of a three-dimensional space. Hence, statements—both practical and abstract—about the larger three-dimensional world hold for the simpler two-dimensional world.

We can represent a point in the plane $z = 0$ as $\mathbf{p} = (x, y, 0)$ in the three-dimensional world, or as $\mathbf{p} = (x, y)$ in the two-dimensional plane. WebGL, like most three-dimensional graphics systems, allows us to use either representation, with the underlying internal representation being the same, regardless of which form the programmer chooses. We can implement representations of points in a number of ways, but the simplest is to think of a three-dimensional point as being represented by a triplet $\mathbf{p} = (x, y, z)$ or a column matrix

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix},$$

whose components give the location of the point. For the moment, we can leave aside the question of the coordinate system in which $\mathbf{p}$ is represented.

We use the terms *vertex* and *point* in a somewhat different manner in WebGL. A **vertex** is a position in space; we use two-, three-, and four-dimensional spaces in computer graphics. We use vertices to specify the atomic geometric primitives that are recognized by our graphics system. The simplest geometric primitive is a point in space, which is usually specified by a single vertex. Two vertices can specify a line segment, a second primitive object; three vertices can specify either a triangle or a circle; four vertices can specify a quadrilateral; and so on. Two vertices can also specify either a circle or a rectangle. Likewise, three vertices can also specify three points

or two connected line segments, and four vertices can specify a variety of objects including two triangles.

The heart of our Sierpinski gasket program is generating the points. In order to go from our third algorithm to a working WebGL program, we need to introduce a little more detail on WebGL. We want to start with as simple a program as possible. One simplification is to delay a discussion of coordinate systems and transformations among them by putting all the data we want to display inside a cube centered at the origin whose diagonal goes from $(-1, -1, -1)$ to $(1, 1, 1)$. This system, known as **clip coordinates**, is the one that our vertex shader uses to send information to the rasterizer. Objects outside this cube will be eliminated or **clipped** and cannot appear on the display. Later, we will learn to specify geometry in our application program in coordinates better suited for our application—**object coordinates**—and use transformations to convert the data to a representation in clip coordinates.

We could write the program using a simple array of two elements to hold the x and y values of each point. In JavaScript, we would construct such an array as follows:

```
var p = [x, y];
```

However, a JavaScript array is not just a collection of values as in C; it is an object with methods and properties, such as `length`. Thus, the code

```
var n = p.length;
```

sets `n` to 2. This difference is important when we send data to the GPU because the GPU expects just an array of numbers. Alternately, we can use a JavaScript typed array, such as

```
var p = new Float32Array([x, y]);
```

which is just a contiguous array of standard 32-bit floating-point numbers and thus can be sent to a GPU with WebGL. In either case, we can initialize the array component-wise as

```
p[0] = x;
p[1] = y;
```

We can generate far clearer code if we first define a two-dimensional point object and operations for this object. We have created such objects and methods and put them in a package `MV.js`. Within these objects, numerical data are stored using JavaScript arrays.

The functions and the three- and four-dimensional objects that we define in `MV.js` match the types in the OpenGL ES Shading Language (GLSL) that we use to write our shaders. Consequently, the use of `MV.js` should make all our coding examples clearer than if we had used ordinary JavaScript arrays. Although these functions have been defined to match GLSL, because JavaScript does not allow overloading of operators as do languages such as C++ and GLSL, we created functions for arithmetic operations using points, vectors, and other types. Nevertheless, code fragments using `MV.js`, such as