GRAPHICS SYSTEMS AND MODELS

Perhaps the dominant characteristic of this new millennium is how computer and communication technologies have become dominant forces in our lives. Activities as wide-ranging as filmmaking, publishing, banking, and education continue to undergo 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 led to new ways of displaying information, seeing virtual worlds, and communicating with people and machines.

Computer graphics is concerned with all aspects of producing pictures or images using a computer. The field began humbly almost 50 years ago, with the display of a few lines on a cathode-ray tube (CRT); now, we can create 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. Massive multiplayer games can involve tens of thousands of concurrent participants.

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 introduce a particular graphics software system, **OpenGL**, which has become a widely accepted standard for developing graphics applications. Fortunately, OpenGL is easy to learn, and it possesses most of the characteristics of other popular 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 will 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 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.

Over the past 150 years, workers in the field of statistics have explored techniques for generating plots that aid the viewer in understanding 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's 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 structural information.

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. The field of information visualization uses computer graphics to aid in the discovery of relationships in data sets in which there is no physical tie between the data and how they are visualized. 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 esthetic 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 iteratively. 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 more than 40 years ago. Today, the use of interactive graphical tools in computer-aided design (CAD) pervades fields including as architecture, mechanical engineering, the design of very-large-scale integrated (VLSI) circuits, and the creation of characters for animations. In all these 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 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. Color Plate 16 shows a physical robot and the corresponding graphical simulation. The simulator can be used for designing the robot, planning its path, and simulating its behavior in complex environments.

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 computers 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. Often images that we see on television, in movies, and in magazines are so realistic that we cannot distinguish computer-generated or computer-altered images from photographs. In Chapters 6 and 9, we discuss many of the lighting effects used to produce computer animations. Color Plate 13 shows a scene from a computer-generated video. The artists and engineers who created this scene used commercially available software. The plate demonstrates our ability to generate realistic environments, in this case a welding robot inside a factory. The sparks demonstrate the use of procedural methods for special effects. We will discuss these techniques in Chapter 11. The images in Color Plate 31 show another example of the use of computer graphics to generate an effect that, although it looks realistic, could not have been created otherwise. The images in Color Plates 23 and 24 also are realistic renderings.

The field of virtual reality (VR) has opened 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, which gives 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.

The graphics to drive interactive video games make heavy use of both standard commodity computers and specialized hardware boxes. To a large degree, games drive the development of graphics hardware. On the commercial side, the revenue from video games has surpassed the revenue for commercial films. The graphics technology for games, both in the form of the graphics processing units that are on graphics cards in personal computers and in game boxes such as the Xbox and the PlayStation, is being used for simulation rather than expensive specialized hardware. Color Plate 16 is a frame from an interactive game, showing a robot warrior, that uses hierarchical methods (Chapter 10), procedural methods for smoke and fire (Chapter 11), and noise textures for landscapes (Chapter 11). The engine that drives the game uses scene graphs that we present in Chapter 10.

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 OS X differ only in details. More recently, millions of people have become Internet users. Their access is through graphical network browsers, such as Firefox and Internet Explorer, that use these same interface tools. We have become so accustomed to this style of interface that we often forget that we are working with computer graphics.

Although we are familiar with the style of graphical user interface used on most workstations,[1] advances in computer graphics have made possible other forms of interfaces. Color Plate 14 shows the interface used with a high-level modeling package. It demonstrates the variety of the tools available in such packages and the interactive devices the user can employ in modeling geometric objects.

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 five major elements in our system:

1. Input devices
2. Processor
3. Memory
4. Frame buffer
5. Output devices

This model is general enough to include workstations and personal computers, interactive game systems, and sophisticated image-generation systems. Although all the components, with the possible exception of the frame buffer, 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.

1.2.1 Pixels and the Frame Buffer

Presently, almost all graphics systems are raster based. A picture is produced as an array—the **raster**—of picture elements, or **pixels**, within the graphics system. As we can see from Figure 1.2, each pixel corresponds to a location, or small area, in

1. Although personal computers and workstations evolved by somewhat different paths, presently, there is virtually no fundamental difference between them. Hence, we use the terms *personal computer* and *workstation* synonymously.

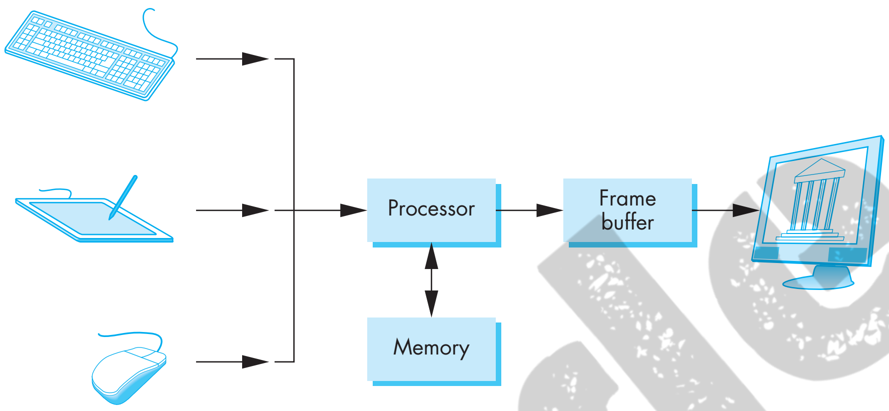

FIGURE 1.1 A graphics system.

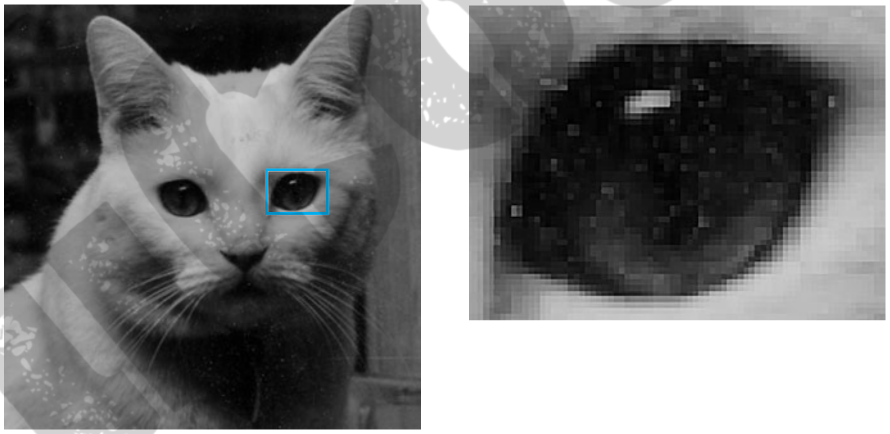

FIGURE 1.2 Pixels. (a) Image of Yeti the cat. (b) Detail of area around one eye showing individual pixels.

the image. Collectively, the pixels are stored in a part of memory called the **frame buffer**. The frame buffer can be viewed as the core element of a graphics system. Its **resolution**—the number of pixels in the frame buffer—determines the detail that you can see in the image. The **depth**, or **precision**, of the frame buffer, 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 frame buffer allows only two colors, whereas an 8-bit-deep frame buffer 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 applications require more than 24-bit fixed point color representations of RGB colors. Some recent frame buffers store RGB values as floating-point numbers in standard IEEE format. Hence, the term *true color* should be interpreted as frame buffers that have sufficient depth to represent colors in terms of RGB values rather than as indices into a limited set of colors. We will return to this topic in Chapter 2.

The frame buffer usually is implemented with special types of memory chips that enable fast redisplay of the contents of the frame buffer. In software-based systems, such as those used for high-resolution rendering or for generating complex visual effects that cannot be produced in real time, the frame buffer is part of system memory.

In a very simple system, the frame buffer holds only the colored pixels that are displayed on the screen. In most systems, the frame buffer holds far more information, such as depth information needed for creating images from three-dimensional data. In these systems, the frame buffer 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 *frame buffer* and *color buffer* synonymously without confusion.

In a simple system, there may be only one processor, the **central processing unit** (**CPU**) of the system, which must do both the normal processing and the graphical processing. 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 frame buffer 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 frame buffer is known as **rasterization**, or **scan conversion**. In early graphics systems, the frame buffer 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 either on the motherboard of the system or on a graphics card. The frame buffer is accessed through the graphics processing unit and may be included in the GPU.

1.2.2 Output Devices

For many years, the dominant type of display (or **monitor**) has been the **cathode-ray tube** (**CRT**). Although various flat-panel technologies are now more popular, the basic functioning of the CRT has much in common with these newer displays. 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

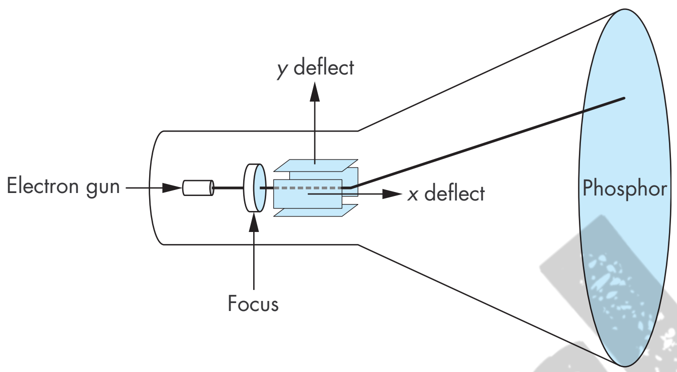

FIGURE 1.3 The cathode-ray tube (CRT).

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 a viewer. Such a device is known as the **random-scan**, **calligraphic**, or **vector** CRT, because the beam can be moved directly from any position to any other position. If the 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 frame buffer and displays them as points on the surface of the display in one of two fundamental ways. In a **noninterlaced** or **progressive** display, 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 process pixels at twice the rate of interlaced displays.

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.

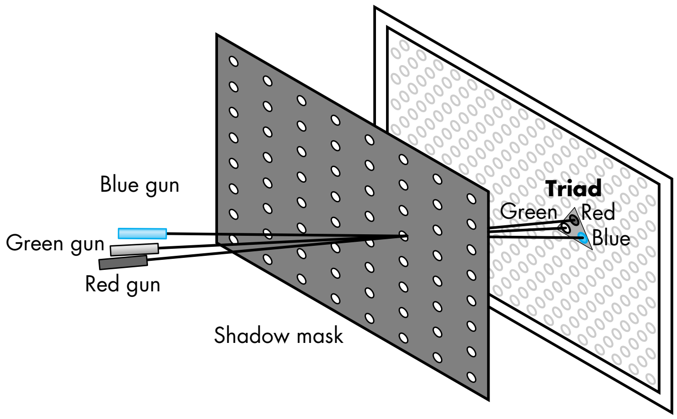

FIGURE 1.4 Shadow-mask CRT.

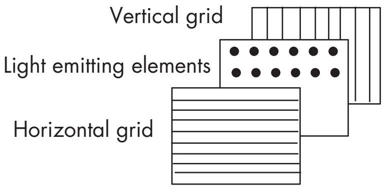

FIGURE 1.5 Generic flat-panel display.

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.

Although CRTs are still the most common display device, they are rapidly being replaced by flat-screen technologies. Flat-panel monitors are inherently raster. 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 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.

Until recently, most displays had a 4:3 width to height ratio (or **aspect ratio**) that correspondeds to commercial television. In discrete terms, displays started with VGA resolution of 640 × 480 pixels, which was consistent with the number of lines displayed in Standard NTSC video.[2] Computer displays moved up to the popular resolutions of 1024 × 768 (XGA) and 1280 × 1024 (SXGA). The newer High Definition Television (HDTV) standard uses a 16:9 aspect ratio, which is between the older television aspect ratio and that of movies. HDTV monitors display 780 or 1080 lines in either progressive (1080p, 780p) or interlaced (1080i, 780i) modes. Hence, the most popular computer display resolutions are now 1920 × 1080 and 1280 × 720, which have the HDTV aspect ratio, and 1920 × 1024 and 1280 × 768, which have the vertical resolution of XGA and SXGA displays. At the high end, there are now "4K" (4096 × 2160) digital projectors that are suitable for commercial digital movies.

1.2.3 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 usually is 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. We study these devices in Chapter 3.

Game consoles lack keyboards but include a greater variety of input devices than a standard workstation. A typical console might have multiple buttons, a joystick, and dials. Devices such as the Nintendo Wii are wireless and can sense accelerations in three dimensions.

Games, CAD, and virtual reality applications have all generated the need for input devices that provide more than two-dimensional data. Three-dimensional locations on a real-world object can be obtained by a variety of devices, including laser range finders and acoustic sensors. Higher-dimensional data can be obtained by devices such as data gloves, which include many sensors, and computer vision systems.

1.3 IMAGES: PHYSICAL AND SYNTHETIC

The traditional pedagogical approach to teaching computer graphics has been focused on how to construct raster images of simple two-dimensional geometric entities (for example, points, line segments, and polygons) in the frame buffer. Next, most textbooks discussed how to define two- and three-dimensional mathematical

2. Outside the United States, the PAL and SECAM systems display more lines but use a lower frame rate.

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 here, rather than to expand a limited model later.

Computer-generated images are synthetic or artificial, in the sense that the objects being imaged may 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 will 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.

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 give 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 OpenGL allows us to build simple objects; in Chapter 11, we learn to define objects in a manner that incorporates relationships among objects.

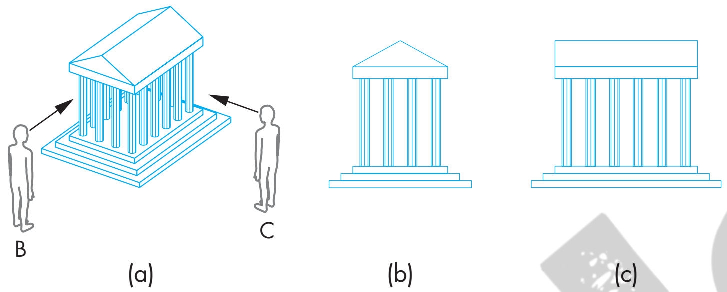

FIGURE 1.6 Image seen by three different viewers. (a) A's view. (b) B's view. (c) C's view.

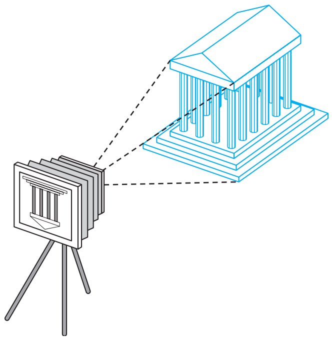

FIGURE 1.7 Camera system.

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 person, 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. Figures 1.6(b) and (c) show 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 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 film 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.

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 shown 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.

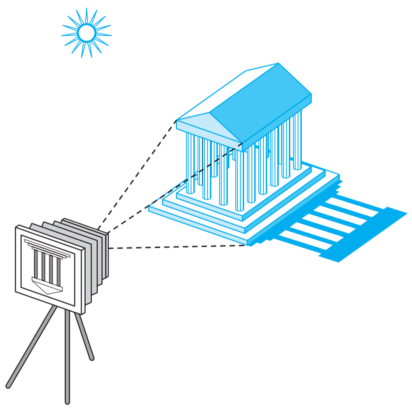
FIGURE 1.8 A camera system with an object and a light source.

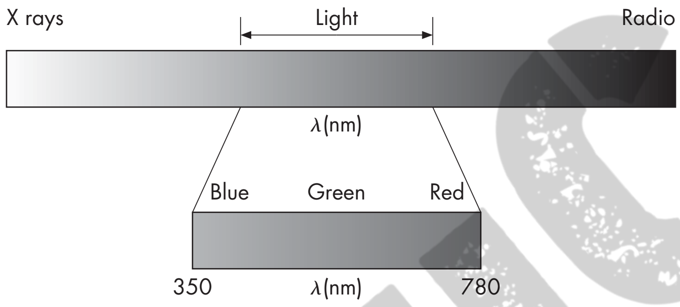

FIGURE 1.9 The electromagnetic spectrum.

Light is a form of electromagnetic radiation. Electromagnetic energy travels as waves that can be characterized by either their wavelengths or 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 wavelengths shorter than blue generate violet light.

Light sources can emit light either as a set of discrete frequencies or continuously. A laser, for example, emits light at a single frequency, whereas an incandescent lamp

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

emits energy over a range of frequencies. Fortunately, in computer graphics, except for recognizing that distinct frequencies are visible as distinct colors, we rarely need to deal with the physical properties of light, such as its wave nature.

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 light bulb, 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 Image Formation Models

There are multiple approaches to how we can form 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 insight into how we can build a useful imaging architecture.

We can start building an imaging model by following light from a source. Consider the scene illustrated 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 1.11. 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

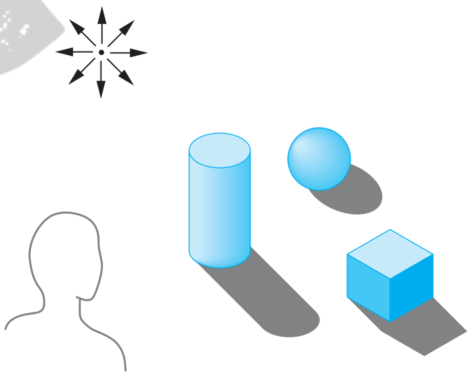

FIGURE 1.10 Scene with a single point light source.

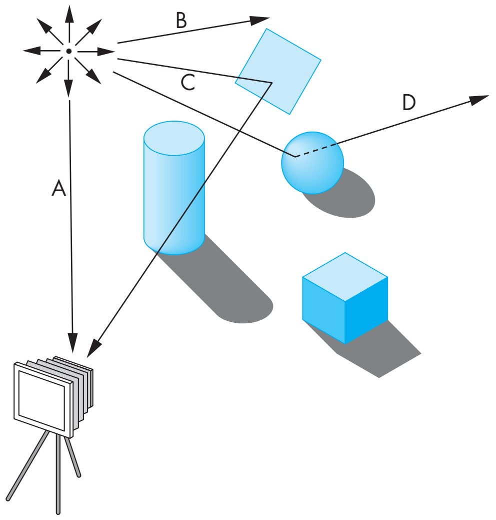

FIGURE 1.11 Ray interactions. Ray A enters camera directly. Ray B goes off to infinity. Ray C is reflected by a mirror. Ray D goes through a transparent sphere.

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 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.[4] 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 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

4. This model of image formation was used by Leonardo da Vinci 500 years ago.

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.

Given the time constraints in image formation by real-time graphics, we usually are satisfied with images that look reasonable rather than images that are physically correct. Such is not the case with images that are non–real-time, such as those generated for a feature film, which can use hours of computer time for each frame. With the increased speed of present hardware, we can get closer to physically correct images in real-time systems. Consequently, we will return to these approaches in Chapter 13.

1.4 IMAGING SYSTEMS

We now introduce two physical 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 shown 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 of the box; the film is placed inside the box on the side opposite the pinhole. Initially, the pinhole is covered. It is uncovered for a short time to expose the film. 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 located a distance *d* from the pinhole. A side view (Figure 1.13) allows us to

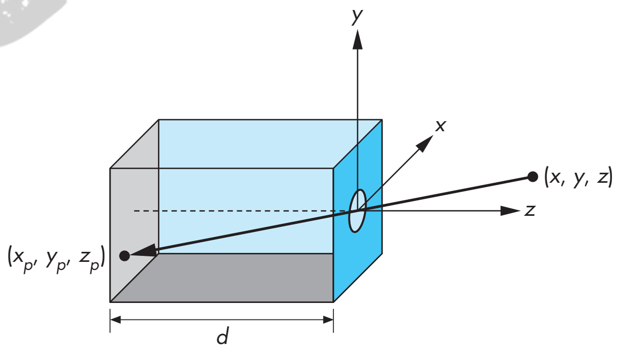

FIGURE 1.12 Pinhole camera.

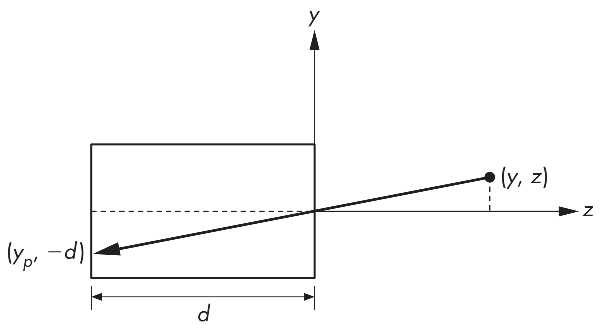

FIGURE 1.13 Side view of pinhole camera.

calculate where the image of the point (x, y, z) is on the film plane $z = -d$. Using the fact that the two triangles shown 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). Note that all points along the line between (x, y, z) and $(x_p, y_p, -d)$ project to $(x_p, y_p, -d)$ so that we cannot go backward from a point in the image plane to the point that produced it. 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.[5] If h is the height of the camera, then the 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, regardless of how far it is from the camera. The image of a point is a point. 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 angle of view.

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

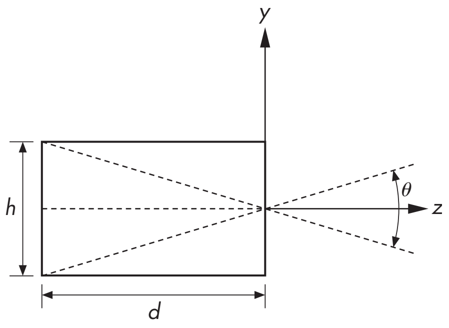

FIGURE 1.14 Angle of view.

The jump to more sophisticated cameras and to other imaging systems that have lenses is direct. 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 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 angle of view (up to 180 degrees). Physical lenses, however, do not have an infinite depth of field: Not all objects in front of 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.

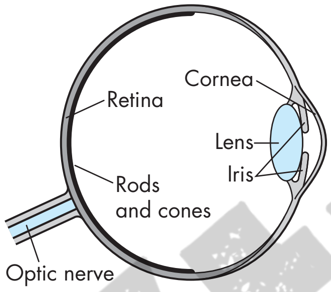

FIGURE 1.15 The human visual system.

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 lens and cornea, a transparent structure that protects the eye. 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 depth in Chapter 2.

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 will 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. Again we see objects and a viewer. In this case, the viewer is a bellows camera.[6] 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 a side view of a camera and a simple object, as shown in Figure 1.17. The view in part (a) is similar to that of the pinhole camera.

6. In a bellows camera, the front 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.

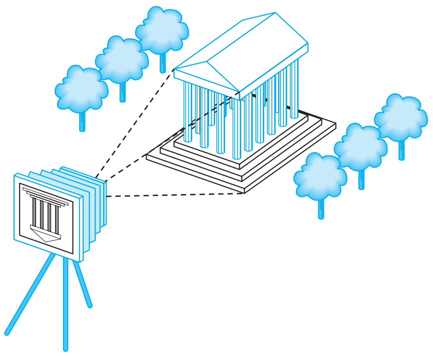
FIGURE 1.16 Imaging system.

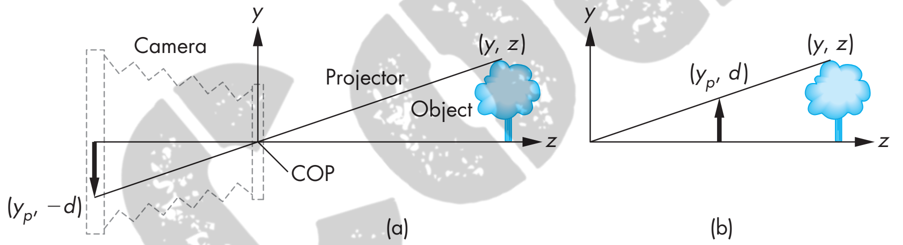

FIGURE 1.17 Equivalent views of image formation. (a) Image formed on the back of the camera. (b) Image plane moved in front of the camera.

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 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 angle 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

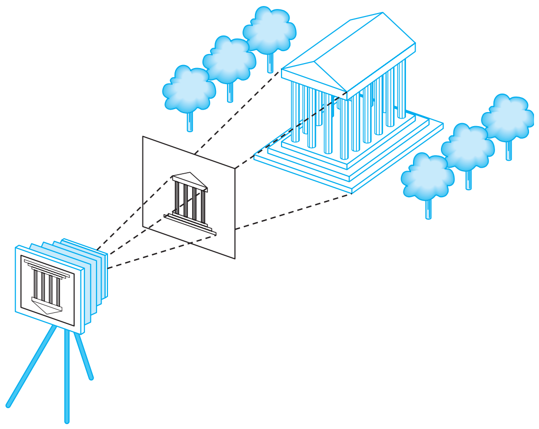
FIGURE 1.18 Imaging with the synthetic camera.

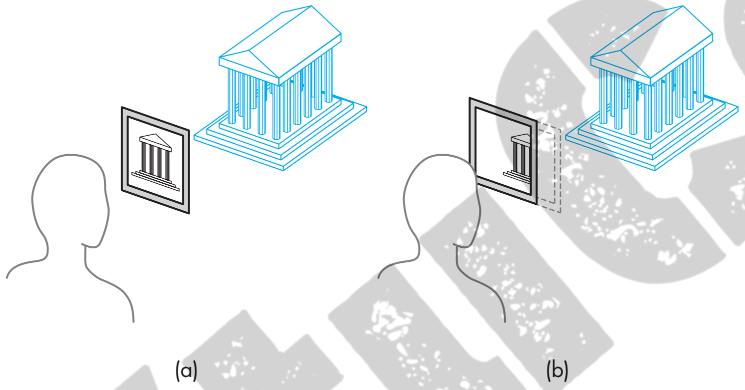

FIGURE 1.19 Clipping. (a) Window in initial position. (b) Window shifted.

and orientation of the projection plane, and the size of the clipping rectangle, we can determine which objects will appear in the image.

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 the ones 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 shown in Figure 1.20, the user sees menus and icons that represent possible actions.

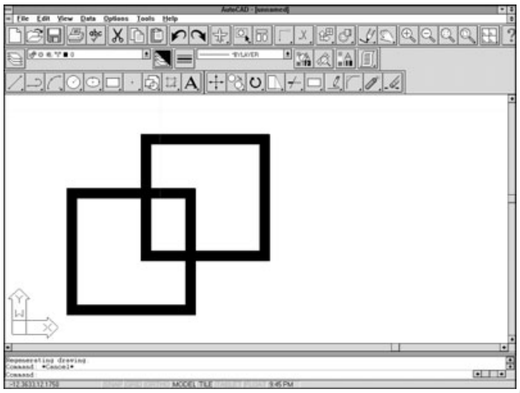

FIGURE 1.20 Interface for a painting program.

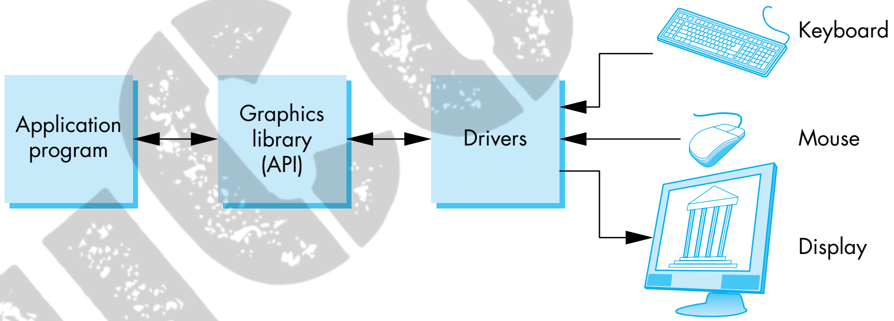

FIGURE 1.21 Application programmer's model of graphics system.

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 specifications are called the **application programmer's 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.[7]

1.6.1 The Pen-Plotter Model

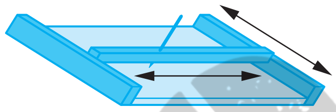
FIGURE 1.22 Pen plotter.

Historically, most early graphics systems were two-dimensional systems. The conceptual model that they used is now referred to as the *pen-plotter model*, referencing 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 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. 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 the following 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);
```

This fragment would generate the output shown 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);
```

7. There may be one or more additional layers between the API and the driver, such as a virtual machine or hardware abstraction layer. However, because the application programmer sees only the API, she does not need to know this level of detail.

```
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 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 limiting, two-dimensional model relies on writing pixels directly into a frame buffer. Such a system could be based on a single function of the form

```
write_pixel(x, y, color)
```

where `x,y` is the location of the pixel in the frame buffer 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.

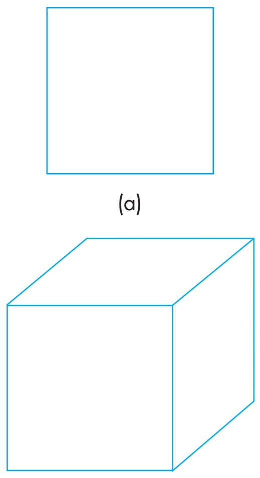

FIGURE 1.23 Output of pen-plotter program for (a) a square, and (b) a projection of a cube.

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.

1.6.2 Three-Dimensional APIs

The synthetic-camera model is the basis for all the popular APIs, including OpenGL, Direct3D, and Open Scene Graph. 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, polygons, and sometimes text. OpenGL programs define primitives through lists of vertices. The following OpenGL code fragment specifies the triangular polygon shown in Figure 1.24 through five function calls:

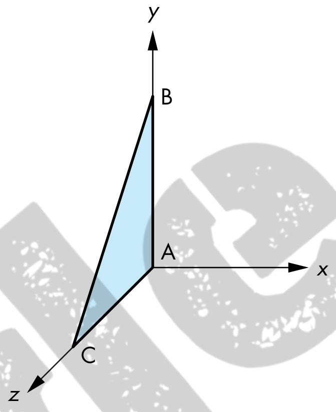

FIGURE 1.24 A triangle.

```
glBegin(GL_POLYGON);
    glVertex3f(0.0, 0.0, 0.0); /* vertex A */
    glVertex3f(0.0, 1.0, 0.0); /* vertex B */
    glVertex3f(0.0, 0.0, 1.0); /* vertex C */
glEnd( );
```

The function `glBegin` specifies the type of primitive that the vertices define. Each subsequent execution of `glVertex3f` specifies the x, y, z coordinates of a location in space. The function `glEnd` ends the list of vertices. Note that by adding additional vertices, we can define an arbitrary polygon. If we change the type parameter, `GL_POLYGON`, we can use the same vertices to define a different geometric primitive. For example, the type `GL_LINE_STRIP` uses the vertices to define two connected line segments, whereas the type `GL_POINTS` uses the same vertices to define three points.

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

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

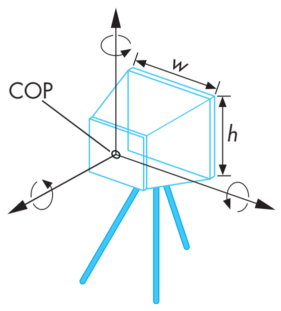

FIGURE 1.25 Camera specification.

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 uses 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 implementing 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 the ones 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 shown 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 OpenGL API allows us to set transformations with complete freedom, it also provides helpful extra functions. For example, consider the following function calls:

```
gluLookAt(cop_x, cop_y, cop_z, at_x, at_y, at_z, up_x, up_y, up_z);
glPerspective(field_of_view, aspect_ratio, near, far);
```

The first function call points the camera from a 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 that the camera should image (the *aspect ratio* and the *near* and *far* distances). 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.

Light sources are defined by their location, strength, color, and directionality. APIs provide a set of functions to specify these parameters for each source. Material properties are characteristics, or attributes, of the objects, and such properties are specified through a series of function calls at the time that each object is defined. Both light sources and material properties depend on the models of light–material interactions supported by the API. We discuss such models in Chapter 6.

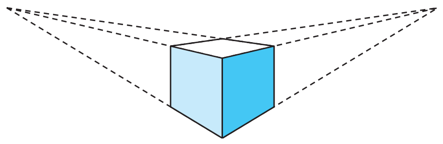
FIGURE 1.26 Two-point perspective of a cube.

1.6.3 A Sequence of Images

In Chapter 2, we begin our detailed discussion of the OpenGL API that we will use throughout this book. The images defined by your OpenGL 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 OpenGL 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 30 years.

Color Plate 1 shows an image of an artist's creation of a sun-like 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 approximately the same time that 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 3, we will show you how to generate images composed of simple geometric objects—points, line segments, and polygons. 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 polygons 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 OpenGL 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 12. 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 8. 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 Chapters 8 and 9.

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 frame buffer. The image on the right has been rendered using a smoothing or antialiasing method that we will 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 will 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

In many situations—especially in CAD applications and in the development of complex images, such as for movies—we can 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. 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 we do not need to work with detailed images of the objects. Consequently, we prefer to carry out this step on an interactive workstation with good graphics hardware. Once we have designed 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 requires a tremendous amount of computation, so we prefer to use a high-performance cluster or a render farm. Not only is the optimal hardware different in the modeling and rendering steps, but the software that we use also may be different.

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 follows this approach and uses a file

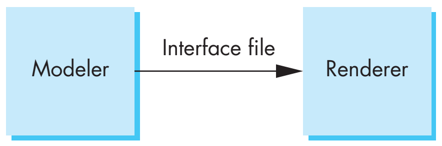

FIGURE 1.27 The modeling–rendering pipeline.

format that allows modelers to pass models to the renderer in text format. One of the other advantages of this approach is that it allows us to develop modelers that, although they use the same renderer, are tailored to particular applications. Likewise, different renderers can take as input the same interface file. It is even possible, at least in principle, to dispense with the modeler completely and to use a standard text editor to generate an interface file. For any but the simplest scenes, however, users cannot edit lists of information for a renderer. Rather, they use interactive modeling software. Because we must have at least a simple image of our objects to interact with a modeler, most modelers use the synthetic-camera model to produce these images in real time.

This paradigm has become popular as a method for generating computer games and images over the Internet. 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 a game engine. We will examine scene graphs in Chapter 10. It is also the standard method used in the animation industry where interactive programs such as Maya and Lightwave are used interactively to build characters using wireframes or unlit polygons. Final rendering can take hours per frame.

1.7 GRAPHICS ARCHITECTURES

On one side of the API is the application program. On the other side 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 display). This information had to be sent to the display at a rate high enough to avoid flicker on the display. 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.

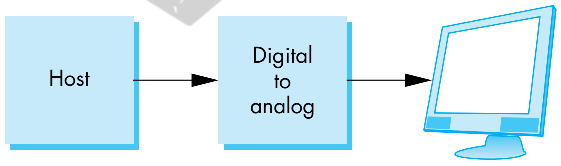

FIGURE 1.28 Early graphics system.

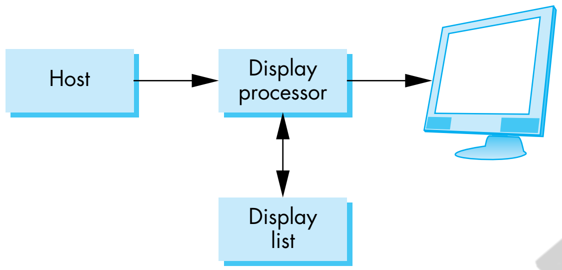

FIGURE 1.29 Display-processor architecture.

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 execute repetitively 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 we will discuss in Chapter 3.

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 technology 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.

Pipelining is similar to an assembly line in a car plant. As the chassis passes down the line, a series of operations is performed on it, each using specialized tools and workers, until at the end, the assembly process is complete. At any one time, multiple cars are under construction and there is a significant delay or **latency** between when a chassis starts down the assembly line and the finished vehicle is complete. However, the number of cars produced in a given time, the **throughput**, is much higher than if a single team built each car.

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 $a + (b * c)$, then 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 computa-

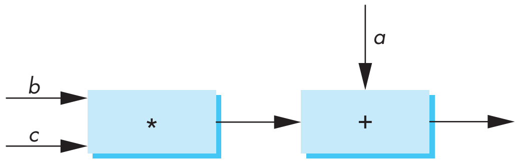

FIGURE 1.30 Arithmetic pipeline.

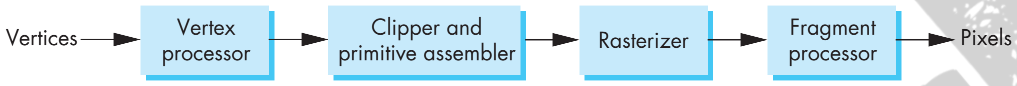

FIGURE 1.31 Geometric pipeline.

tion 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 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 flows through the system, the throughput of the system, has been doubled. Note that as we add more boxes to a pipeline, the latency of the system increases and we must balance latency 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 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 frame buffer. 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 overview these steps and show that they can be pipelined.

1.7.4 Vertex Processing

In the first block of our pipeline, each vertex is processed independently. The two major functions of this block are to carry out coordinate transformations and to compute a color for each 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 the synthetic camera paradigm, 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. In Chapter 5, we see 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 will 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 will 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 in 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 7.

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 further processed to generate pixels in the frame buffer. For example, if three vertices specify a triangle filled with a solid color, the rasterizer must determine which pixels in the frame buffer are inside the polygon. We discuss this rasterization (or scan-conversion) process in Chapter 8 for line segments and polygons. 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 frame buffer. Fragments can also carry along depth information that allows later stages to determine if 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 frame buffer. 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 shown in Color Plates 6 and 7. The color of the pixel that corresponds to a fragment can also be read from the frame buffer and blended with the fragment's color to create translucent effects. These effects will be covered in Chapters 8 and 9.

1.8 PROGRAMMABLE PIPELINES

Graphics architectures have gone through multiple 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—leads to real-time performance. Hence, the commodity graphics market is dominated by graphics cards that have pipelines built into the graphics processing unit. All of these commodity cards implement the pipeline that we have just described, albeit with more options, many of which we will 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. For example, there was only one lighting model for how to compute a shade using the specified light sources and materials.

Recently, there has been a major advance in pipeline architectures. Both the vertex processor and the fragment processor are now programable by the application program. 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.

Vertex programs 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 programs allow us to use textures in new ways. Bump mapping, which is illustrated in Color Plate 6, is but one example of an algorithm that is now programmable through texture mapping but formerly could only be done off-line. Chapter 9 is devoted to these new methodologies.

1.9 PERFORMANCE CHARACTERISTICS

There are two fundamentally different types of processing in our pipeline 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, reducing a matrix multiplication to a single instruction. Today, 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 frame buffer. This back-end processing is fundamentally different from front-end processing, and we implement it most effectively using architectures that have the ability to move blocks of bits quickly. The overall performance of a system 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 frame buffer. Consequently, the fastest graphics workstations are characterized by one or more 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 frame buffers. These GPUs are so powerful that they are being used for purposes other than graphics applications.

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 in a system. Commodity graphics cards incorporate the pipeline within their GPUs. Cards that cost less than $100 can render millions of shaded texture-mapped polygons per second. However, we can also make as strong a case for pipelining being the basis of a complete software implementation of an API. The power of the synthetic-camera paradigm is that the latter works well in both cases.

However, where realism is important, other types of renderers can perform better at the expense of requiring more computation time. Pixar's RenderMan interface was created to interface their off-line renderer. Physically based techniques, such as ray tracing and radiosity, can create photorealistic images with great fidelity, but not in real time.