ATME College of Engineering

# Department of Computer Science and Engineering

## Notes of Lesson

**SUBJECT TITLE AND CODE: COMPUTER GRAPHICS – BCS515A**

**SEMESTER AND SCHEME : V SEMESTER**

## INSTITUTIONAL MISSION AND VISION

**Objectives**

- To provide quality education and groom top-notch professionals, entrepreneurs and leaders for different fields of engineering, technology and management.
- To open a Training-R & D-Design-Consultancy cell in each department, gradually introduce doctoral and postdoctoral programs, encourage basic & applied research in areas of social relevance, and develop the institute as a center of excellence.
- To develop academic, professional and financial alliances with the industry as well as the academia at national and transnational levels.
- To develop academic, professional and financial alliances with the industry as well as the academia at national and transnational levels.
- To cultivate strong community relationships and involve the students and the staff in local community service.
- To constantly enhance the value of the educational inputs with the participation of students, faculty, parents and industry.

**Vision**

- Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources.

**Mission**

- To keep pace with advancements in knowledge and make the students competitive and capable at the global level.
- To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torch bearers of tomorrow's society.
- To strive to attain ever-higher benchmarks of educational excellence.

# ***Department of Computer Science And Engineering***

## *Vision of the Department*

Nurture graduates in Computer Science and Engineering to face challenges in industry, education and society at global level.
Promote research that responds swiftly to the needs of the 21st century and to build a Center of Excellence.

## *Mission of the Department*

➢ *To inculcate professional behavior, strong ethical values, innovative research capabilities and leadership abilities in the young minds & to provide a teaching environment that emphasizes depth, originality and critical thinking.*

➢ *Motivate students to put their thoughts and ideas adoptable by industry or to pursue higher studies leading to research.*

## *Program Educational Objectives (PEO'S):*

1. Empower students with a strong basis in the mathematical, scientific and engineering fundamentals to solve computational problems and to prepare them for employment, higher learning and R&D.
2. Gain technical knowledge, skills and awareness of current technologies of computer science engineering and to develop an ability to design and provide novel engineering solutions for software/hardware problems through entrepreneurial skills.
3. Exposure to emerging technologies and work in teams on interdisciplinary projects with effective communication skills and leadership qualities.
4. Ability to function ethically and responsibly in a rapidly changing environment by applying innovative ideas in the latest technology, to become effective professionals in Computer Science to bear a life-long career in related areas.

# Module 1

**Syllabus: Graphics Systems and Models:** Applications of Computer Graphics, A Graphics System, Images: Physical and Synthetic, Imaging Systems, The Synthetic-Camera Model, The Programmer's Interface, Graphics Architectures, Programmable Pipelines, Performance Characteristics.

## 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

### 1.1.1 Display of Information

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

ATMECE

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.

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.

### 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 over determined, 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.

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.

### 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.

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.

ATMECE

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.

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.

### 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.

Although we are familiar with the style of graphical user interface used on most workstations, 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

ATMECE

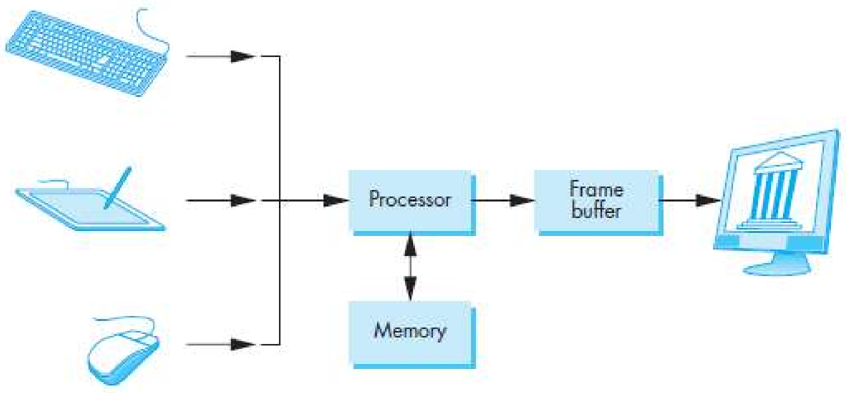
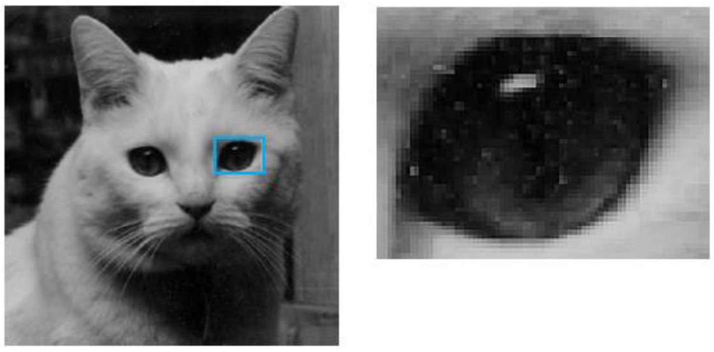


Figure 1.1

### 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 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 28 (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,

ATMECE

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.

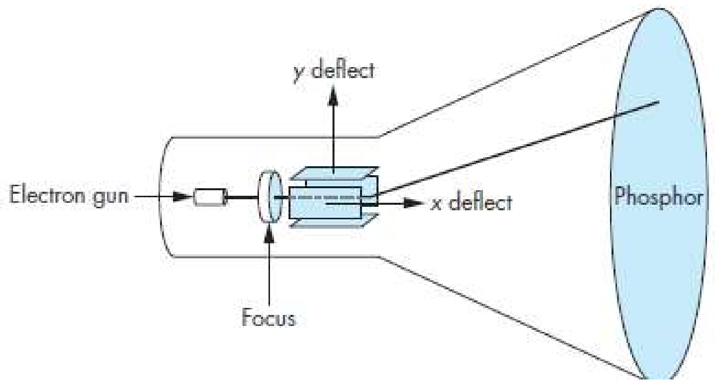
The frame buffer usually is implemented with special types of memory chips that enable fast redisplay of the contents of the frame buffer. In software-based systems, such as those used for high-resolution rendering or for generating complex visual effects that cannot be produced in real time, the frame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

In a 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.

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.

ATMECE

### 1.2.2 Output Devices



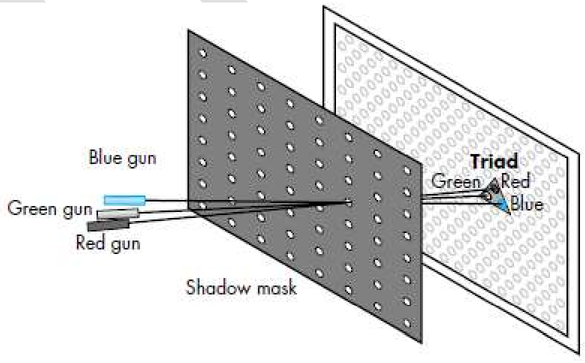
For many years, the dominant type of display (or monitor) has been the cathode ray tube (CRT). Although various flat-panel technologies are now more popular, the basic functioning of the CRT has much in common with these newer displays. A simplified picture of a CRT is shown in Figure 1.3. When electrons 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 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.

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.



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

ATMECE

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.

### 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.

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

Computer-generated images are synthetic or artificial, in the sense that the objects being imaged may not exist physically. 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.

### 1.3.1 Objects and Viewers

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

ATMECE

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.

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.

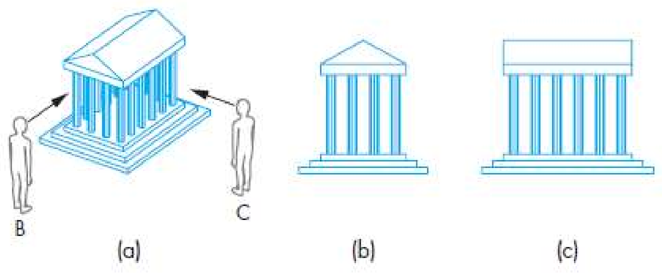


Figure 1.6

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.

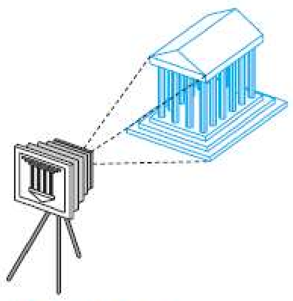


FIGURE 1.7 Camera system.

ATMECE

### 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.

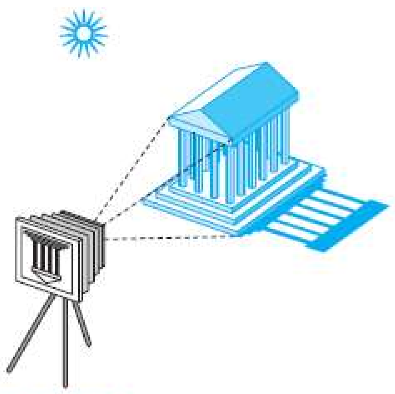


FIGURE 1.8 A camera system with an object and a light source.

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.

### 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.

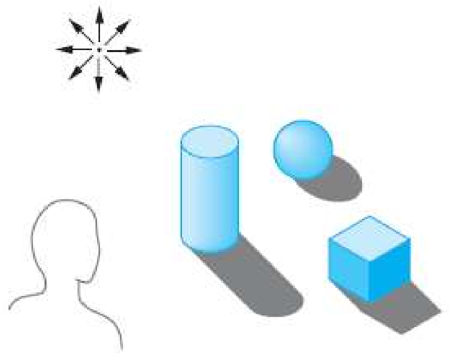


FIGURE 1.10 Scene with a single point light source.

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 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.

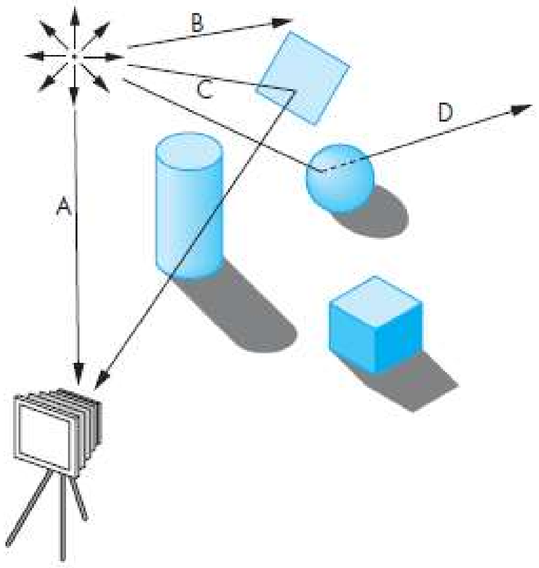


FIGURE 1.11 Ray interactions. Ray A enters camera directly. Ray B goes off to infinity. Ray C is reflected by a mirror. Ray D goes through a transparent sphere.

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.

## 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. The human visual system is extremely complex but still obeys the physical principles of other optical imaging systems.

### 1.4.1 The Pinhole Camera

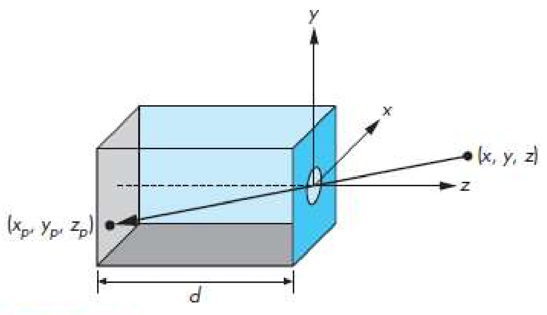


FIGURE 1.12 Pinhole camera.

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 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 yp, where

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

A similar calculation, using a top view, yields

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

ATMECE

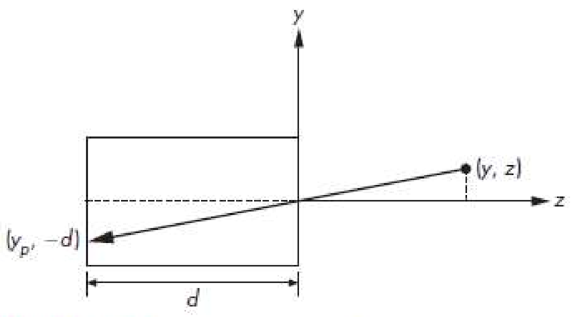


FIGURE 1.13 Side view of pinhole camera.

The point (xp, yp, −d) is called the projection of the point (x, y, z). Note that all points along the line between (x, y, z) and (xp, yp, −d) project to (xp, yp, −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 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.

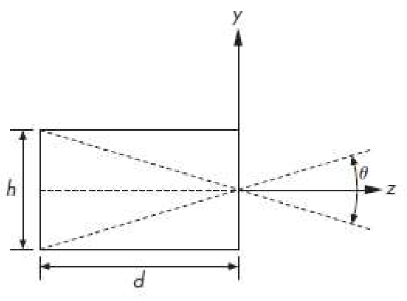


FIGURE 1.14 Angle of view.

### 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

ATMECE

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.

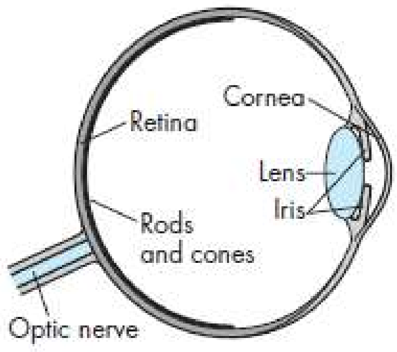


FIGURE 1.15 The human visual system.

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.

## 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. 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**.**

ATMECE

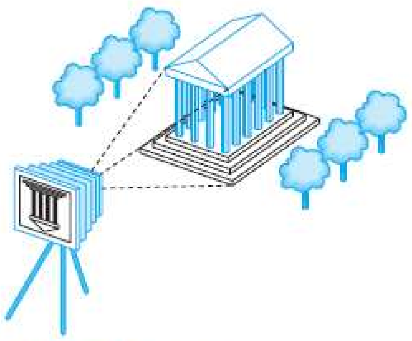


FIGURE 1.16 Imaging system.

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.

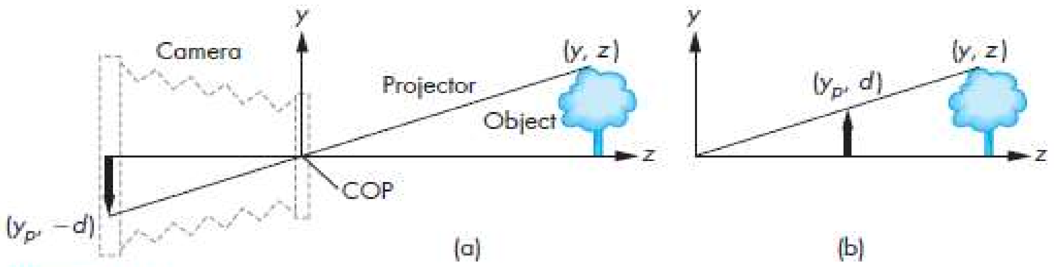


FIGURE 1.17 Equivalent views of image formation. (a) Image formed on the back of the camera. (b) Image plane moved in front of the camera.

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.

ATMECE

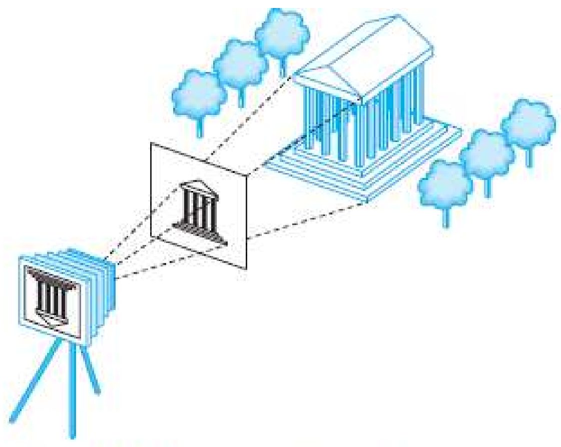


FIGURE 1.18 Imaging with the synthetic camera.

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

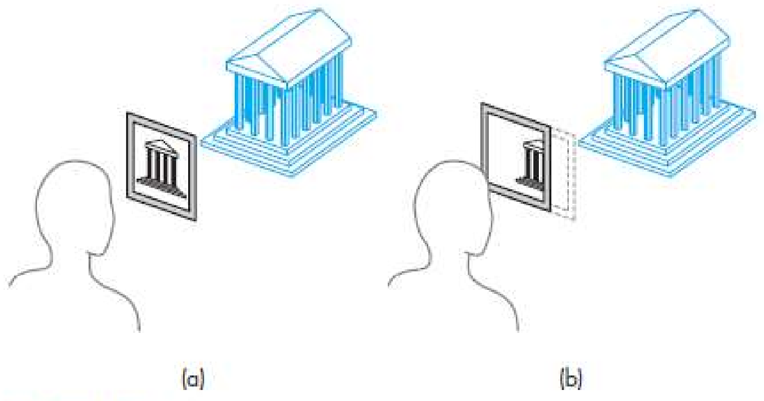


FIGURE 1.19 Clipping. (a) Window in initial position. (b) Window shifted.

## 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 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**.**

By clicking on these items, the user guides the software and produces images without having to write programs.

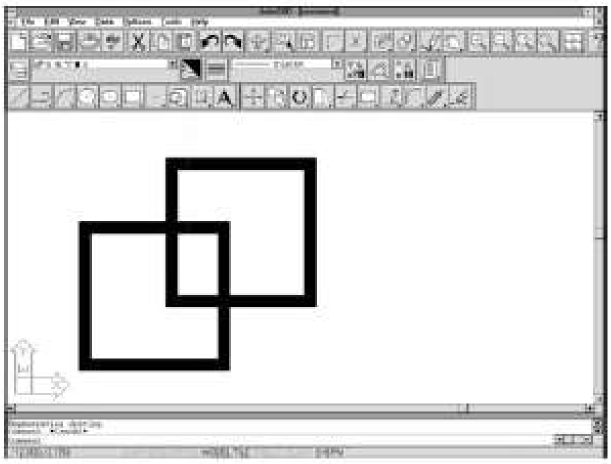


FIGURE 1.20 Interface for a painting program.

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.

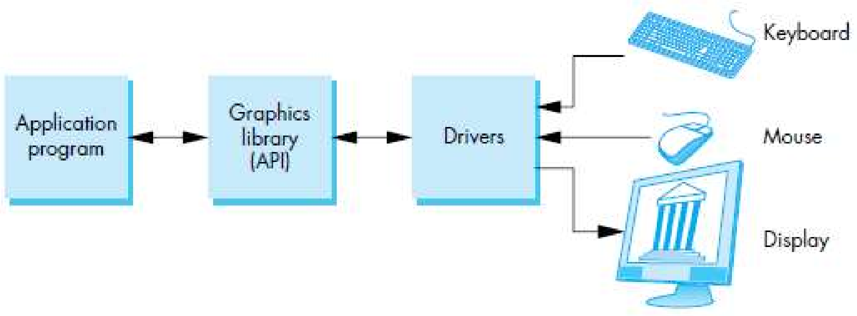


FIGURE 1.21 Application programmer's model of graphics system.

### 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, 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

ATMECE

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);
lineto(1, 0);
moveto(1, 1);
lineto(1,5,1.866);
```

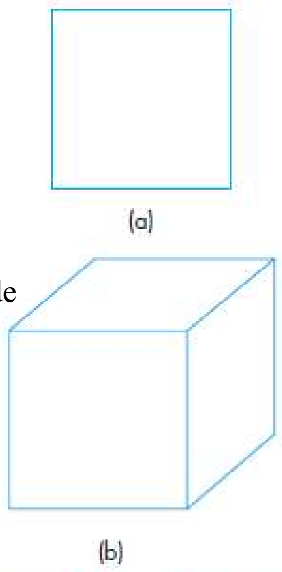


FIGURE 1.23 Output of pen-plotter program for (a) a square, and (b) a projection of a cube.

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-

ATMECE

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.

### 1.6.2 Three-Dimensional APIs

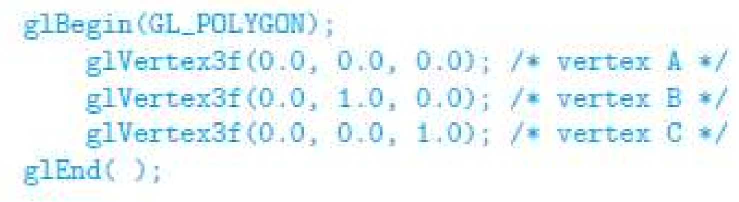
The synthetic-camera model is the basis for all the popular APIs, including OpenGL, Direct3D, and Open Scene Graph. If we are to follow the synthetic-camera model, we need functions in the API to specify the following:

- 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 bemultiple 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:

ATMECE



```
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( );
```

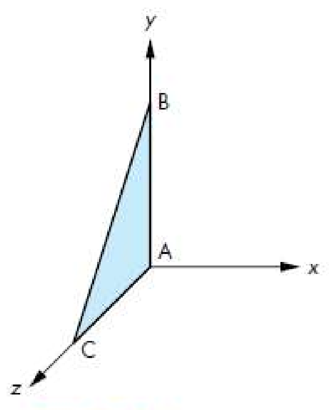


FIGURE 1.24 A triangle.

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:

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.

### 1.6.3 A Sequence of Images

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.

ATMECE

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. 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.

### 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 modelling 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 Render Man interface follows this approach and uses a file 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.

ATMECE

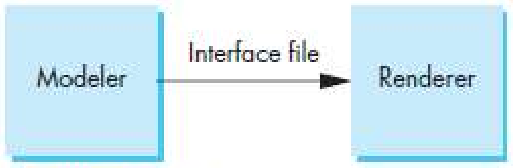


FIGURE 1.27 The modeling–rendering pipeline.

## 1.7 GRAPHICS ARCHITECTURES

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.

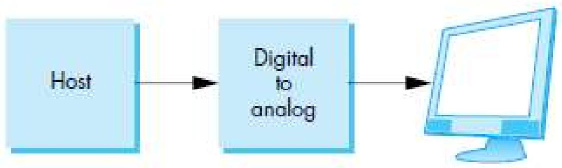


FIGURE 1.28 Early graphics system.

### 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.

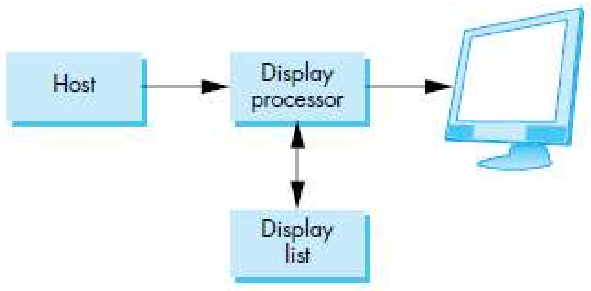


FIGURE 1.29 Display-processor architecture.

### 1.7.2 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 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 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.

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.

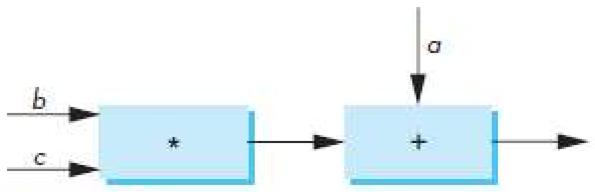


FIGURE 1.30 Arithmetic pipeline.

### 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

### 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 ($4 \times 4$), we have the opportunity to use parallelism within the transformation blocks in the pipeline.

ATMECE

### 1.7.4 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.

### 1.7.5 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. 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.

### 1.7.6 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.

ATMECE

## 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.

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.

## 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 $4 \times 4$ 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.

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.

ATMECE

# Module 2

**Syllabus: Input and Interaction:** Interaction, Input devices, Clients and Servers, Display Lists, Display Lists and Modeling, Programming Event Driven Input, Menus.

## 2.1 INTERACTION

One of the most important advances in computer technology was enabling users to interact with computer displays. More than any other event, Ivan Sutherland's Sketchpad project launched the present era of interactive computer graphics. The basic paradigm that he introduced is deceptively simple. The user sees an image on the display. She reacts to this image by means of an interactive device, such as a mouse. The image changes in response to her input. She reacts to this change, and so on. Whether we are writing programs using the tools available in a modern window system or using the human computer interface in an interactive museum exhibit, we are making use of this paradigm.

Although rendering is the prime concern of most modern APIs, including OpenGL, interactivity is an important component of most applications. OpenGL, however, does not support interaction directly. The major reason for this omission is that the system architects who designed OpenGL wanted to increase its portability by allowing the system to work in a variety of environments. Consequently, window and input functions were left out of the API. Although this decision makes renderers portable, it makes discussions of interaction that do not include specifics of the window system more difficult. In addition, because any application program must have at least a minimal interface to the window environment, we cannot avoid such issues completely if we want to write complete, nontrivial programs.

We can avoid such potential difficulties by using the GLUT toolkit. This toolkit provides the minimal functionality that is expected on virtually all systems, such as opening of windows, use of the keyboard and mouse, and creation of pop-up menus. We adopt this approach, even though it may not provide all the features of any particular windowing system and produces code that neither makes use of the full capabilities of any particular window system nor proves as efficient as code written for a particular environment. However, writing code for the standard window systems is based on the principles that we can illustrate most simply using GLUT.

We use the term window system, to include the total environment provided by systems such as the X Window System, Microsoft Windows, and the Macintosh Operating System. Graphics programs that we develop will render into a window within one of these environments. The terminology used in the window system literature may obscure the distinction between, for example, an X window and the OpenGL window into which our graphics are rendered. However, you will usually be safe if you regard the OpenGL window as a particular type of window on your system that can display output from OpenGL programs. Our use of the GLUT toolkit will enable us to

ATMECE

avoid the complexities inherent in the interactions among the window system, the window manager, and the graphics system.

## 2.2 INPUT DEVICES

### 2.2.1 Physical Input Devices

The pointing device allows the user to indicate a position on a display and almost always incorporates one or more buttons to allow the user to send signals or interrupts to the computer. The keyboard device is almost always a physical keyboard but can be generalized to include any device that returns character codes. For example, a tablet PC uses recognition software to decode the user's writing with a stylus but in the end produces character codes identical to those of the standard keyboard.

We will use the American Standard Code for Information Interchange (ASCII) in our examples. ASCII assigns a single unsigned byte to each character. Nothing we do restricts us to this particular choice, other than that ASCII is the prevailing code used. Note, however, that other codes, especially those used for Internet applications, use multiple bytes for each character, thus allowing for a much richer set of supported characters.

The mouse (Figure 3.1) and trackball (Figure 3.2) are similar in use and often in construction as well. When turned over, a typical mechanical mouse looks like a trackball. In both devices, the motion of the ball is converted to signals sent back to the computer by pairs of encoders inside the device that are turned by the motion of the ball. The encoders measure motion in two orthogonal directions.

There are many variants of these devices. Some use optical detectors rather than mechanical detectors to measure motion. Small trackballs are popular with portable computers because they can be incorporated directly into the keyboard. There are also various pressure-sensitive devices used in keyboards that perform similar functions to the mouse and trackball but that do not move; their encoders measure the pressure exerted on a small knob that often is located between two keys in the middle of the keyboard.

ATMECE

FIGURE 3.1 Mouse.

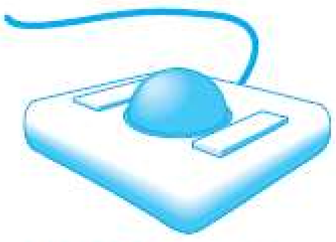


FIGURE 3.2 Trackball.

A typical data tablet (Figure 3.4) has rows and columns of wires embedded under its surface. The position of the stylus is determined through electromagnetic interactions between signals traveling through the wires and sensors in the stylus. Touch-sensitive transparent screens that can be placed over the face of a CRT have many of the same properties as the data tablet. Small, rectangular, pressure-sensitive touchpads are embedded in the keyboards of most portable computers. These touchpads can be configured as either relative- or absolute positioning devices. Some are capable of detecting simultaneous input from two fingers touching different spots on the pad and can use this information to enable more complex behaviors.

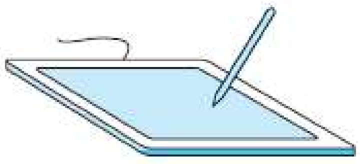


FIGURE 3.4 Data tablet.

The lightpen has a long history in computer graphics. It was the device used in Sutherland's original Sketchpad. The lightpen contains a light-sensing device, such as a photocell (Figure 3.5). If the lightpen is positioned on the face of the CRT at a location opposite where the electron beam strikes the phosphor, the light emitted exceeds a threshold in the photodetector and a signal is sent to the computer. The light pen was originally used on random scan devices so the time of the interrupt could easily be matched to a piece of code in the display list, thus making the light pen ideal for selecting application-defined objects. With raster scan devices, the position on the display can be determined by the time the scan begins and the time it takes to scan each line. Hence, we have a direct-positioning device. The lightpen is not as popular as the mouse, data tablet, and trackball. One of its major deficiencies is that it has difficulty obtaining a position that corresponds to a dark area of the screen. However, tablet PCs are used in a manner that mimics how the light pen was used originally; the user has a stylus with which she can move randomly about the tablet (display) surface.

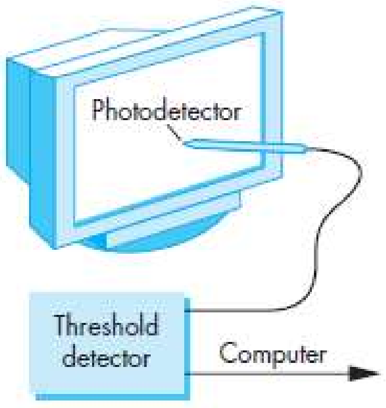


FIGURE 3.5 Lightpen.

ATMECE

One other device, the joystick (Figure 3.6), is particularly worthy of mention. The motion of the stick in two orthogonal directions is encoded, interpreted as two velocities, and integrated to identify a screen location. The integration implies that if the stick is left in its resting position, there is no change in the cursor position and that the farther the stick is moved from its resting position, the faster the screen location changes. Thus, the joystick is a variable-sensitivity device. The other advantage of the joystick is that the device can be constructed with mechanical elements, such as springs and dampers, that give resistance to a user who is pushing the stick. Such a mechanical feel, which is not possible with the other devices, makes the joystick well suited for applications such as flight simulators and game controllers.

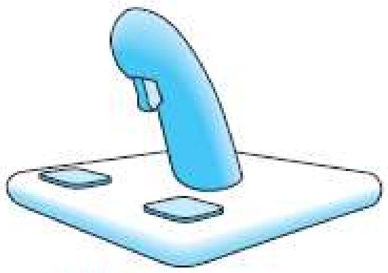


FIGURE 3.6 Joystick.

For three-dimensional graphics, we might prefer to use three-dimensional input devices. Although various such devices are available, none have yet won the widespread acceptance of the popular two-dimensional input devices. A spaceball looks like a joystick with a ball on the end of the stick (Figure 3.7); however, the stick does not move. Rather, pressure sensors in the ball measure the forces applied by the user. The spaceball can measure not only the three direct forces (up–down, front–back, left–right) but also three independent twists. The device measures six independent values and thus has six degrees of freedom. Such an input device could be used, for example, both to position and to orient a camera. Other three-dimensional devices, such as laser scanners, measure three dimensional positions directly. Numerous tracking systems used in virtual reality applications sense the position of the user. Virtual reality and robotics applications often need more degrees of freedom than the two to six provided by the devices that we have described. Devices such as data gloves can sense motion of various parts of the human body, thus providing many additional input signals. Recently, in addition to being wireless, input devices such as Nintendo’s Wii incorporate gyroscopic sensing of position and orientation.

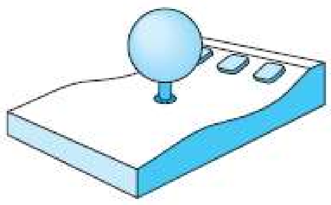


FIGURE 3.7 Spaceball.

ATMECE

### 2.2.2 Logical Devices

Some earlier APIs defined six classes of logical input devices. Because input in a modern window system cannot always be disassociated completely from the properties of the physical devices, OpenGL does not take this approach. Nevertheless, we describe the six classes briefly because they illustrate the variety of input forms available to a developer of graphical applications. We will see how OpenGL can provide the functionality of each of these classes.

1. **String**: A string device is a logical device that provides ASCII strings to the user program. This logical device is usually implemented by means of a physical keyboard. In this case, the terminology is consistent with that used in most window systems and OpenGL, which usually do not distinguish between the logical string device and a physical keyboard.
2. **Locator**: A locator device provides a position in world coordinates to the user program. It is usually implemented by means of a pointing device, such as a mouse or a trackball. In OpenGL, we usually use the pointing device in this manner, although we have to do the conversion from screen coordinates to world coordinates within our own programs.
3. **Pick:** A pick device returns the identifier of an object on the display to the user program. It is usually implemented with the same physical device as a locator, but has a separate software interface to the user program. In OpenGL, we can use a process called selection (Section 3.8) to accomplish picking.
4. **Choice:** Choice devices allow the user to select one of a discrete number of options. In OpenGL, we can use various widgets provided by the window system. A widget is a graphical interactive device, provided by either the window system or a toolkit. Typical widgets include menus, scrollbars, and graphical buttons. Most widgets are implemented as special types of windows. For example, a menu with n selections acts as a choice device, allowing us to select one of n alternatives. Widget sets are the key element defining a graphical user interface, or GUI.
5. **Valuators:** Valuators provide analog input to the user program. On some graphics systems, there are boxes or dials to provide valuator input. Here again, widgets within various toolkits usually provide this facility through graphical devices such as slide bars and radio boxes.
6. **Stroke:** A stroke device returns an array of locations. Although we can think of a stroke device as similar to multiple uses of a locator, it is often implemented such that an action, say, pushing down a mouse button, starts the transfer of data into the specified array, and a second action, such as releasing the button, ends this transfer.

ATMECE

### 2.2.3 Input Modes

The manner by which input devices provide input to an application program can be described in terms of two entities: a measure process and a device trigger. The measure of a device is what the device returns to the user program. The trigger of a device is a physical input on the device with which the user can signal the computer. For example, the measure of a keyboard should include a single character or a string of characters, and the trigger can be the Return or Enter key.

The application program can obtain the measure of a device in three distinct modes. Each mode is defined by the relationship between the measure process and the trigger. Once a measure process is started, the measure is taken and placed in a buffer, even though the contents of the buffer may not yet be available to the program. For example, the position of a mouse is tracked continuously by the underlying window system and a cursor is displayed regardless of whether the application program needs mouse input.

In request mode, the measure of the device is not returned to the program until the device is triggered. This input mode is standard in non graphical applications. For example, if a typical C program requires character input, we use a function such as scanf. When the program needs the input, it halts when it encounters the scanf statement and waits while we type characters at our terminal. We can backspace to correct our typing, and we can take as long as we like. The data are placed in a keyboard buffer whose contents are returned to our program only after a particular key, such as the Enter key (the trigger), is pressed. For a logical device, such as a locator, we can move our pointing device to the desired location and then trigger the device with its button; the trigger will cause the location to be returned to the application program. The relationship between measure and trigger for request mode is shown in Figure 3.8.

Sample-mode input is immediate. As soon as the function call in the user program is encountered, the measure is returned. Hence, no trigger is needed (Figure 3.9). In sample mode, the user must have positioned the pointing device or entered data using the keyboard before the function call, because the measure is extracted immediately from the buffer.

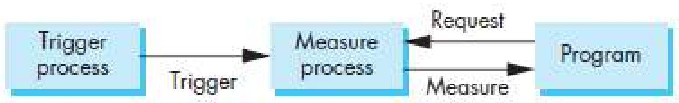


FIGURE 3.8 Request mode.

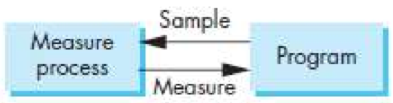


FIGURE 3.9 Sample mode.

One characteristic of both request- and sample-mode input in APIs that support them is that the user must identify which device is to provide the input. Consequently, we ignore any other information that becomes available from any input device other than the one specified. Both request and sample modes are useful for situations where the program guides the user but are not useful in applications where the user controls the flow of the program. For example, a flight simulator or computer game might have multiple input devices such as a joystick, dials, buttons, and switches most of which can be used at any time. Writing programs to control the simulator with only sample- and request-mode input is nearly impossible because we do not know what devices the pilot will use at any point in the simulation. More generally, sample- and request-mode input are not sufficient for handling the variety of possible human computer interactions that arise in a modern computing environment.

Our third mode, eventmode, can handle these other interactions. We introduce it in three steps. First, we show how event mode can be described as another mode within our measure–trigger paradigm. Second, we discuss the basics of client and servers where event mode is the preferred interaction mode. Third, we show an eventmode interface to OpenGL using GLUT, and we write demonstration programs using this interface.

Suppose that we are in an environment with multiple input devices, each with its own trigger and each running a measure process. Each time that a device is triggered, an event is generated. The device measure, including the identifier for the device, is placed in an event queue. This process of placing events in the event queue is completely independent of what the application program does with these events. One way that the application program can work with events is shown in Figure 3.10. The application program can examine the front event in the queue or, if the queue is empty, can wait for an event to occur. If there is an event in the queue, the program can look at the first event's type and then decide what to do. If, for example, the first event is from the keyboard but the application program is not interested in keyboard input, the event can be discarded and the next event in the queue can be examined.

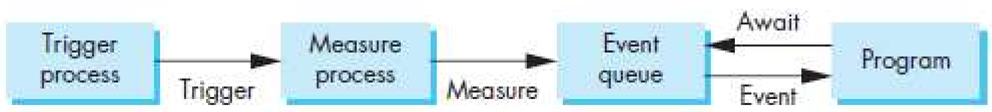


FIGURE 3.10 Event–mode model.

### 2.3 CLIENTS AND SERVERS

We have looked at our graphics system as a monolithic box with limited connections to the outside world, rather than through our carefully controlled input devices and a display. Networks and multiuser computing have changed this picture dramatically, and to such an extent that, even if we had a single-user isolated system, its software probably would be configured as a simple client–server network.

If computer graphics is to be useful for a variety of real applications, it must function well in a world of distributed computing and networks. In this world, our building blocks are entities called servers that can perform tasks for clients. Clients and servers can be distributed over a network (Figure 3.11) or contained entirely within a single computational unit. Familiar examples of servers include print servers, which can allow sharing of a high-speed printer among users; compute servers, such as remotely located high-performance computers, accessible from user programs; file servers that allow users to share files and programs, regardless of the machine they are logged into; and terminal servers that handle dial-in access. Users and user programs that make use of these services are clients or client programs. Servers can also exist at a lower level of granularity within a single operating system. For example, the operating system might provide a clock service that multiple client programs can use. It is less obvious what we should call a workstation connected to the network: It can be both a client and a server, or perhaps more to the point, a workstation may run client programs and server programs concurrently.

The model that we use here was popularized by the X Window System. We use much of that system's terminology, which is now common to most window systems and fits well with graphical applications.

A workstation with a raster display, a keyboard, and a pointing device, such as a mouse, is a graphics server. The server can provide output services on its display and input services through the keyboard and pointing device. These services are potentially available to clients anywhere on the network.

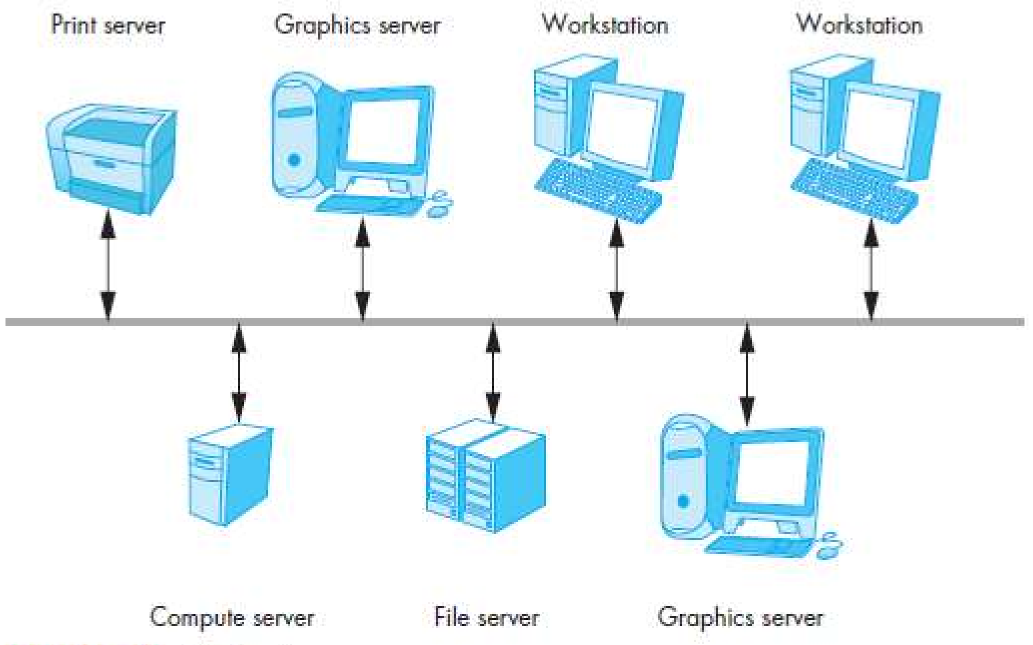


FIGURE 3.11 Network.

## 2.4 DISPLAY LISTS

Display lists illustrate how we can use clients and servers on a network to improve interactive graphics performance. Display lists have their origins in the early days of computer graphics.

The original architecture of a graphic system was based on a general-purpose computer (or host) connected to a display (Figure 3.12). The computer would send out the necessary information to redraw the display at a rate sufficient to avoid noticeable flicker. At that time (circa 1960), computers were slow and expensive, so the cost of keeping even a simple display refreshed was prohibitive for all but a few applications.

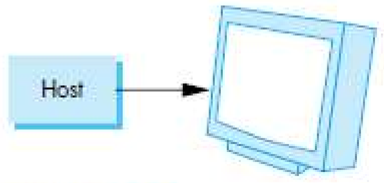


FIGURE 3.12 Simple graphics architecture.

The solution to this problem was to build a special-purpose computer, called a **display processor**, with an organization like that illustrated in Figure 3.13. The display processor had a limited instruction set, most of which was oriented toward drawing primitives on the display. The user program was processed in the host computer, resulting in a compiled list of instructions that was then sent to the display processor, where the instructions were stored in a display memory as a display file, or display list. For a simple non-interactive application, once the display list was sent to the display processor, the host was free for other tasks, and the display processor would execute its display list repeatedly at a rate sufficient to avoid flicker. In addition to resolving the bottleneck due to burdening the host, the display processor introduced the advantages of special-purpose rendering hardware.

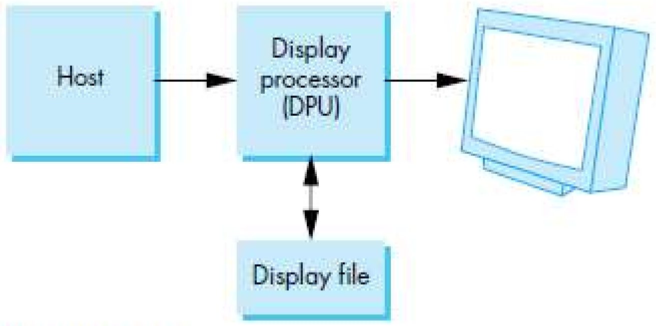


FIGURE 3.13 Display-processor architecture.

Today, the display processor of old has become a graphics server, and the application program on the host computer has become a client. The major bottleneck is no longer the rate at which we have to refresh the display

ATMECE

(although that is still a significant problem), but rather the amount of traffic that passes between the client and server. In addition, the use of special-purpose hardware now characterizes high-end systems.

We can send graphical entities to a display in one of two ways. We can send the complete description of our objects to the graphics server. For our typical geometric primitives, this transfer entails sending vertices, attributes, and primitive types, in addition to viewing information. In our fundamental mode of operation, immediate mode, as soon as the program executes a statement that defines a primitive, that primitive is sent to the server for possible display and no memory of it is retained in the system.2 To redisplay the primitive after a clearing of the screen, or in a new position after an interaction, the program must re-specify the primitive and then must resend the information through the display process. For complex objects in highly interactive applications, this process can cause a considerable quantity of data to pass from the client to the server.

Display lists offer an alternative to this method of operation. This second method is called retained-mode graphics. We define the object once, and then put its description in a display list. The display list is stored in the server and redisplayed by a simple function call issued from the client to the server. In addition to conferring the obvious advantage of reduced network traffic, this model allows much of the overhead in executing commands to be done once and have the results stored in the display list on the graphics server.

There are, of course, a few disadvantages to the use of display lists. Display lists require memory on the server, and there is the overhead of creating a display list. Although this overhead is often offset by the efficiency of the execution of the display list, it might not be if the data are changing.

### 2.4.1 Definition and Execution of Display Lists

Display lists have much in common with ordinary files. There must be a mechanism to define (create) and manipulate (place information in) them. The definition of which contents of a display list are permissible should be flexible enough to allow considerable freedom to the user. OpenGL has a small set of functions to manipulate display lists and places only a few restrictions on display list contents. We develop several simple examples to show the functions' uses.

Display lists are defined similarly to geometric primitives. There is a glNewList at the beginning and a glEndList at the end, with the contents in between. Each display list must have a unique identifier—an integer that is usually macro defined in the C program by means of a #define directive to an appropriate name for the object in the list. For example, the following code defines a red box.

ATMECE

```
#define BOX 1 /* or some other unused integer */

glNewList(BOX,  GL_COMPILE);
    glBegin(GL_POLYGON);
         glColor3f(1.0, 0.0, 0.0);
         glVertex2f(-1.0, -1.0);
         glVertex2f( 1.0, -1.0);
         glVertex2f( 1.0,  1.0);
         glVertex2f(-1.0,  1.0);
    glEnd();
glEndList();
```

The flag GL_COMPILE tells the system to send the list to the server but not to display its contents. If we want an immediate display of the contents while the list is being constructed, we can use the GL_COMPILE_AND_EXECUTE flag instead. Each time that we wish to draw the box on the server, we execute the function as follows: **glCallList(BOX);**

Just as it does with other OpenGL functions, the current state determines which transformations are applied to the primitives in the display list. Thus, if we change the model-view or projection matrices between executions of the display list, the box will appear in different places or will no longer appear, as the following code fragment demonstrates:

```
glMatrixMode(GL_PROJECTION);
for(i= 1 ; i<5; i++)
{
     glLoadIdentity();
     gluOrtho2D(-2.0*i  , 2.0*i , -2.0*i , 2.0*i );
     glCallList(BOX);
}
```

A stack is a data structure in which the item placed most recently in the structure is the first removed. We can save the present values of attributes and matrices by placing, or pushing them on the top of the appropriate stack; we can recover them later by removing, or popping them from the stack. A standard and safe procedure is always to push both the attributes and matrices on their own stacks when we enter a display list, and to pop them when we exit. Thus, we usually see the function calls

**glPushAttrib(GL_ALL_ATTRIB_BITS);**
**glPushMatrix();**

at the beginning of a display list and

**glPopAttrib();**
**glPopMatrix();**

at the end.

ATMECE

### 2.4.2 Text and Display Lists

Earlier, we introduced both stroke and raster text. Regardless of which type we choose to use, we need a reasonable amount of code to describe a set of characters. For example, suppose that we use a raster font in which each character is stored as an 8 × 13 pattern of bits. It takes 13 bytes to store each character. If we want to display a string by the most straightforward method, we can send each character to the server each time that we want it displayed. This transfer requires the movement of at least 13 bytes per character. If we define a stroke font using only line segments, each character can require a different number of lines. If we use filled polygons for characters, as shown in Figure 3.14, we see that an "I" is fairly simple to define, but we may need many line segments to get a sufficiently smooth "O." On the average, we need many more than 13 bytes per character to represent a stroke font. For applications that display large quantities of text, sending each character to the display every time that it is needed can place a significant burden on our graphics systems.

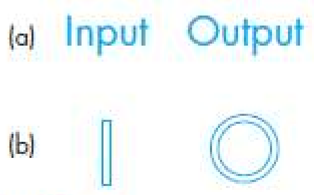


FIGURE 3.14 Stroke characters. (a) Filled strings. (b) Magnified outlines.

A more efficient strategy is to define the font once, using a display list for each character, and then to store the font on the server using these display lists. This solution is similar to what is done for bitmap fonts on standard alphanumeric display terminals. The patterns are stored in read-only memory (ROM) in the terminal, and each character is selected and displayed based on a single byte: its ASCII code. The difference here is one of both quantity and quality. We can define as many fonts as our display memory can hold, and we can treat stroke fonts like other graphical objects, allowing us to translate, scale, and rotate them as desired.

The basics of defining and displaying a character string (1 byte per character) using a stroke font and display lists provide a simple but important example of the use of display lists in OpenGL. The procedure is essentially the same for a raster font. We can define either the standard 96 printable ASCII characters or we can define patterns for a 256-character extended ASCII character set.

First, we define a function OurFont(char c), which will draw any ASCII character c that can appear in our string. The function might have a form like the following:

ATMECE

```
void OurFont(char c)
   {
        switch(c)
        {
            case '
                 ..
            break;
            case '
                 ..
            break;
                 ..
        }
```

Within each case, we have to be careful about the spacing; each character in the string must be displayed to the right. We can use the translate function to get the desired spacing or shift the vertex positions. Suppose that we are defining the letter "O" and we wish it to fit in a unit square. The corresponding part of OurFont might be as follows:

```
case 'O':
    glTranslatef(0.5, 0.5, 0.0); /* move to center */
    glBegin(GL_QUAD_STRIP);
    for (i=0; i<=12; i++)  /* 12 vertices */
    {
          angle = 3.14159 /6.0 * i; /* 30 degrees in radians */
          glVertex2f(0.4*cos(angle)+0.5; 0.4*sin(angle)+0.5);
          glVertex2f(0.5*cos(angle)+0.5, 0.5*sin(angle)+0.5);
    }
    glEnd();
    break;
```

This code approximates the circle with 12 quadrilaterals. Each will be filled according to the current state. Although we do not discuss the full power of transformations until Chapter 4, here we explain the use of the translation function in this code.

We are working with two-dimensional characters. Hence, each character is defined in the plane z = 0, and we can use whatever coordinate system we wish to define our characters. We assume that each character fits inside a box.3 The usual strategy is to start at the lower-left corner of the first character in the string and to draw one character at a time, drawing each character such that we end at the lower-right corner of that character's box, which is the lower-left corner of the successor's box.

The first translation moves us to the center of the "O" character's box, which we set to be a unit square. We then define our vertices using two concentric circles centered at this point (Figure 3.15). One way to envision the translation function is to say that it shifts the origin for all the drawing commands that follow. After the 12 quadrilaterals in the strip are defined, we move to the lower-right corner of the box.

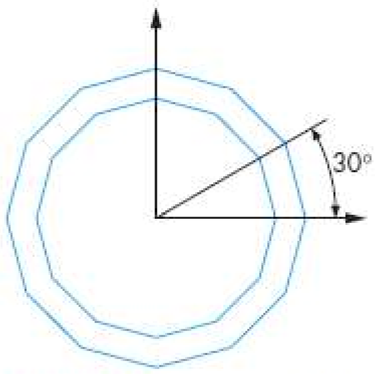


FIGURE 3.15 Drawing of the letter "O."

The two translations accumulate; as a result of these translations, we are in the proper position to start the next character. Note that, in this example, we do not want to push and pop the matrices. Other characters can be defined in a similar manner. Although our code is inelegant, its efficiency is of little consequence because the characters are generated only once and then are sent to the graphics server as a compiled display list.

Suppose that we want to generate a 256-character set. The required code, using the OurFont function, is as follows:

```
base = glGenLists(256); /* return index of first of 256
                           consecutive available ids */
for(i=0; i<256; i++)
{
      glNewList(base + i, GL_COMPILE);
      OurFont(i);
      glEndList();
}
```

When we wish to use these display lists to draw individual characters, rather than offsetting the identifier of the display lists by base each time, we can set an offset as follows:

**glListBase(base);**

Finally, our drawing of a string is accomplished in the server by the function call

**char *text_string;**

**glCallLists( (GLint) strlen(text_string), GL_BYTE, text_string);**

which makes use of the standard C library function strlen to find the length of input string text_string. The first argument in the function glCallLists is the number of lists to be executed. The third is a pointer to an array of a type given by the second argument. The identifier of the kth display list executed is the sum of the list base (established by glListBase) and the value of the kth character in the array of characters.

ATMECE

### 2.4.3 Fonts in GLUT

GLUT provides a few raster and stroke fonts. They do not make use of display lists; in the final example in this chapter, however, we create display lists to contain one of these GLUT fonts. We can access a single character from a monotype, or evenly spaced, font by the following function call:

**glutStrokeCharacter(GLUT_STROKE_MONO_ROMAN, int character)**

GLUT_STROKE_ROMAN provides proportionally spaced characters. You should use these fonts with caution. Their size (approximately 120 units maximum) may have little to do with the units of the rest of your program; thus, they may have to be scaled. We usually control the position of a character by using a translation before the character function is called. In addition, each invocation of glutStrokeCharacter includes a translation to the bottom right of the character's box, to prepare for the next character. Scaling and translation affect the OpenGL state, so here we should be careful to use glPushMatrix and glPopMatrix as necessary to prevent undesirable positioning of objects defined later in the program. Our discussion of transformations in Chapter 4 should enable you to use stroke fonts effectively.

Raster and bitmap characters are produced in a similar manner. For example, a single 8 × 13 character is obtained using the following:

**glutBitmapCharacter(GLUT_BITMAP_8_BY_13, int character)**

Positioning of bitmap characters is considerably simpler than the positioning of stroke characters is because bitmap characters are drawn directly in the frame buffer and are not subject to geometric transformations, whereas stroke characters are. OpenGL keeps, within its state, a raster position. This position identifies where the next raster primitive will be placed; it can be set using the glRasterPos*() function. The user program typically moves the raster position to the desired location before the first character in a string defined by glutBitmapCharacter is invoked.

## 2.5 DISPLAY LISTS AND MODELING

Looking at the spectrum of applications of interactive computer graphics, we can observe that a large fraction of them involve some sort of interactive modeling. For example, CAD applications allow the user to interactively design buildings, model circuits, and build computer animations. From the perspective of computer graphics, we must design the user interfaces for such applications and will do so over the next few sections. But underlying these applications as with most computer applications are both conceptual and implemented models employing a variety of data structures that are designed to support efficient interactions.

ATMECE

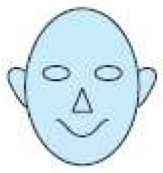


FIGURE 3.16 Simple animated face

Because display lists can call other display lists, they are powerful tools for building hierarchical models that can incorporate relationships among parts of a model. Consider a simple face modeling system that can produce images such as those shown in Figure 3.16 that might be used for animation. Each face has two identical eyes and two identical ears plus the outline, a nose, and a mouth. We could specify the constituent parts through display lists, such as the following:

```
#define EYE 1 /* or some other integer */

glNewList(EYE);
/* eye code */
glEndList();
```

The code for a face would then use transformations (Chapter 5) to bring each component into its desired location as follows:

```
#define FACE 2 /* or some other integer */

glNewList(FACE);
   /* draw the outline */
   glTranslatef(....) /* right eye position*/


      glCallList(EYE);
      glTranslatef(....) /*left eye position*/
      glCallList(EYE);
      glTranslatef(....) /* nose position */
      glCallList(NOSE);

   /* similar code for ears and mouth */

   glEndList();
```

There are some significant advantages to this approach. First, we can make use of the fact that there are multiple instances of the same component by calling the same display list multiple times. Second, if we want to create a different character, we can change one or more of the display lists for the constituent parts but we can leave the display list for the face unchanged because the structure of a face as described in it remains unchanged.

ATMECE

## 2.6 PROGRAMMING EVENT-DRIVEN INPUT

### 2.6.1 Using the Pointing Device

Two types of events are associated with the pointing device, which is conventionally assumed to be a mouse but could be a trackball or a data tablet.

**A move event** is generated when the mouse is moved with one of the buttons pressed. If the mouse is moved without a button being held down, this event is called a passive move event. After a move event, the position of the mouse its measure is made available to the application program.

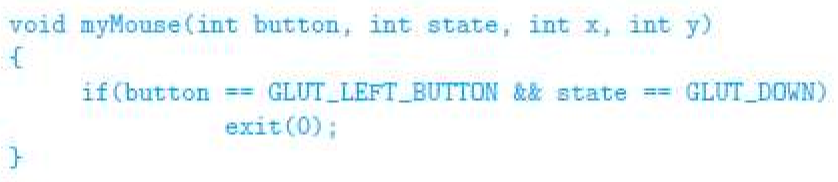
**A mouse event** occurs when one of the mouse buttons is either pressed or released. A button being held down does not generate a mouse event until the button is released. The information returned includes the button that generated the event, the state of the button after the event (up or down), and the position of the cursor tracking the mouse in window coordinates (with the origin in the upper-left corner of the window).We register the mouse callback function, usually in the main function, by means of the GLUT function as follows:

**glutMouseFunc(myMouse);**

The mouse callback must have the form

**void myMouse(int button, int state, int x, int y)**

and is written by the application programmer. Within the callback function, we define the actions that we want to take place if the specified event occurs. There may be multiple actions defined in the mouse callback function corresponding to the many possible button and state combinations. For our simple example, we want the pressing of the left mouse button to terminate the program. The required callback is the following single-line function:



```
void myMouse(int button, int state, int x, int y)
{
     if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
             exit(0);
}
```

If any other mouse event such as the pressing of one of the other buttons occurs, no response action will occur, because no action corresponding to these events has been defined in the callback function.

First, we look at the main program, which is much the same as our previous examples.

ATMECE

```
int main(int argc, char **argv)
{
  glutInit(&argc,argv);
  glutInitWindowSize(ww, wh); /* globally defined initial window size */
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
  glutCreateWindow("square");
  myInit();
  glutReshapeFunc(myReshape);
  glutMouseFunc(myMouse);
  glutDisplayFunc(myDisplay);
  glutMainLoop();
}
```

The reshape event is generated whenever the window is resized, such as by a user interaction; we discuss it next. We do not need the required display callback for drawing in this example because the only time that primitives will be generated is when a mouse event occurs. Because GLUT requires that all programs have a display callback, we must include this callback, although it can have a simple body:

```
void myDisplay()
{
   glClear(GL_COLOR_BUFFER_BIT);
}
```

The mouse callbacks are again in the function myMouse.

```
void myMouse(int btn, int state, int x, int y)
{
  if(btn==GLUT_LEFT_BUTTON && state==GLUT_DOWN) drawSquare(x,y);
  if(btn==GLUT_RIGHT_BUTTON && state==GLUT_DOWN) exit();
}
```

We need three global variables. The size of the window may change dynamically, and its present size should be available, both to the reshape callback and to the drawing function drawSquare. If we want to change the size of the squares we draw, we may find it beneficial to make the square-size parameter global as well. Our initialization routine selects a clipping window that is the same size as the window created in main and specifies a viewport to correspond to the entire window. This window is cleared to black. Note that we could omit the setting of the window and viewport here because we are merely setting them to the default values.

ATMECE

```
/* globals */

GLsizei wh = 500, ww = 500; /* initial window width and height */
GLfloat size = 3.0;   /*one half of side length */

void myInit()
{
   /* set initial viewing conditions */

      glViewport(0,0,ww,wh);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluOrtho2D(0.0, (GLdouble) ww , 0.0, (GLdouble) wh);
      glMatrixMode(GL_MODELVIEW);
      glClearColor (0.0, 0.0, 0.0, 1.0);
      glColor3f(1.0, 0.0, 0.0); /*red squares*/
```

Our square-drawing routine has to take into account that the position returned from the mouse event is in the window system's coordinate system, which has its origin at the top left of the window. Hence, we have to flip the y value returned, using the present height of the window (the global wh) as follows:

```
void drawSquare(int x, int y)
{
      y=wh-y;

      glBegin(GL_POLYGON);
          glVertex2f(x+size, y+size);
          glVertex2f(x-size, y+size);
          glVertex2f(x-size, y-size);
          glVertex2f(x+size, y-size);
      glEnd();
      glFlush();
}
```

### 2.6.2 Window Events

Most window systems allow a user to resize the window interactively, usually by using the mouse to drag a corner of the window to a new location. This event is an example of a window event.

In our square-drawing example, we ensure that squares of the same size are drawn, regardless of the size or shape of the window. We clear the screen each time it is resized, and we use the entire new window as our drawing area. The reshape event returns in its measure the height and width of the new window. We use these values to create a new OpenGL clipping window using gluOrtho2D, as well as a new viewport with the same aspect ratio. We then clear the window to black. Thus, we have the following callback:

ATME

```
void myReshape(GLsizei w, GLsizei h)
{

    /* adjust clipping box */

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(0.0, (GLdouble)w, 0.0, (GLdouble)h);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();

   /* adjust viewport and clear */


     glViewport(0,0,w,h);

     /* save new window size in global variables

     ww=w;
     wh=h;
 }
```

There are other possibilities here. We could change the size of the squares to match the increase or decrease of the window size. We have not considered other events, such as a window movement without resizing, an event that can be generated by a user who drags the window to a new location. And we have not specified what to do if the window is hidden behind another window and then is exposed (or brought to the front) by the user. There are callbacks for these events, and we can write simple functions similar to myReshape for them or we can rely on the default behaviour of GLUT. Another simple change that we can make to our program is to have new squares generated as long as one of the mouse buttons is held down. The relevant callback is the motion callback, which we set through the following function:

**glutMotionFunc(drawSquare);**

### 2.6.3 Keyboard Events

We can also use the keyboard as an input device. Keyboard events are generated when the mouse is in the window and one of the keys is pressed or released. The GLUT function **glutKeyboardFunc** is the callback for events generated by pressing a key, whereas **glutKeyboardUpFunc** is the callback for events generated by releasing a key.

When a keyboard event occurs, the ASCII code for the key that generated the event and the location of the mouse are returned. All the keyboard callbacks are registered in a single callback function, such as the following:

**glutKeyboardFunc(myKey);**

For example, if we wish to use the keyboard only to exit the program, we can use the following callback function:

ATMECE

```
void myKey(unsigned char key, int x, int y)
{
    if(key=='q' || key == 'Q') exit( );
}
```

### 2.6.4 The Display and Idle Callbacks

This callback is specified in GLUT by the following function call:

**glutDisplayFunc(myDisplay);**

It is invoked when GLUT determines that the window should be redisplayed. One such situation occurs when the window is opened initially; another happens after a resize event. Because we know that a display event will be generated when the window is first opened, the display callback is a good place to put the code that generates most non-interactive output.

The display callback can be used in other contexts, such as in animations, where various values defined in the program may change. We can also use GLUT to open multiple windows. The state includes the present window, and we can render different objects into different windows by changing the present window. We can also iconify a window by replacing it with a small symbol or picture. Consequently, interactive and animation programs will contain many calls for the re-execution of the display function. Rather than call it directly, we use the GLUT function as follows:

**glutPostRedisplay();**

Using this function, rather than invoking the display callback directly, avoids extra or unnecessary screen drawings by setting a flag inside GLUT's main loop indicating that the display needs to be redrawn. At the end of each execution of the main loop, GLUT uses this flag to determine whether the display function will be executed. Thus, using **glutPostRedisplay** ensures that the display will be drawn only once each time the program goes through the event loop.

### 2.6.5 Window Management

GLUT also supports multiple windows and subwindows of a given window. We can open a second top-level window (with the label "second window") as follows:

**id=glutCreateWindow("second window");**

The returned integer value allows us to select this window as the current window into which objects will be rendered as follows: **glutSetWindow(id);**

ATMECE

We can make this window have properties different from those of other windows by invoking the **glutInitDisplayMode** before **glutCreateWindow**. Furthermore, each window can have its own set of callback functions because callback specifications refer to the present window.

## 2.7 MENUS

GLUT provides one additional feature, pop-up menus, that we can use with the mouse to create sophisticated interactive applications.

Using menus involves taking a few simple steps. We must define the actions corresponding to each entry in the menu. We must link the menu to a particular mouse button. Finally, we must register a callback function for each menu. We can demonstrate simple menus with the example of a pop-up menu that has three entries.

The first selection allows us to exit our program. The second and third change the size of the squares in our **drawSquare** function. We name the menu callback demo_ menu. The function calls to set up the menu and to link it to the right mouse button should be placed in our main function. They are as follows:

```
glutCreateMenu(demo_menu);
glutAddMenuEntry("quit",1);
glutAddMenuEntry("increase square size", 2);
glutAddMenuEntry("decrease square size", 3);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

The function **glutCreateMenu** registers the callback function **demo_menu**. The second argument in each entry's definition is the identifier passed to the callback when the entry is selected. Hence, our callback function is as follows:

```
void demo_menu(int id)
{
     switch(id)
     {
          case 1: exit(0);
          break;
          case 2: size = 2 * size;
          break;
          case 3: if(size > 1) size = size/2;
          break;
     }
     glutPostRedisplay( );
}
```

The call to glutPostRedisplay requests a redraw through the glutDisplayFunc callback, so that the screen is drawn again without the menu.

ATMECE

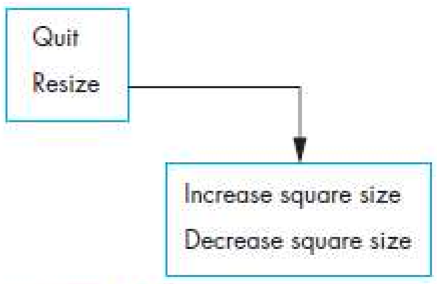


FIGURE 3.18 Structure of hierarchical menus.

GLUT also supports hierarchical menus, as shown in Figure 3.18. For example, suppose that we want the main menu that we create to have only two entries. The first entry still causes the program to terminate, but now the second causes a submenu to pop up. The submenu contains the two entries for changing the size of the square in our square-drawing program. The following code for the menu (which is in main) should be clear:

```
sub_menu = glutCreateMenu(size_menu);
glutAddMenuEntry("Increase square size", 2);
glutAddMenuEntry("Decrease square size", 3);
glutCreateMenu(top_menu);
glutAddMenuEntry("Quit",1);
glutAddSubMenu("Resize", sub_menu);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

**Program to draw Square using Mouse and Keyboard Callbacks**

```
#include <stdlib.h>
#include <GL/glut.h>
GLsizei wh = 500, ww = 500; /* initial window width and height */
GLfloat size = 3.0; /*one half of side length */

void myDisplay()
{
        glClear(GL_COLOR_BUFFER_BIT);
}

void myMouse(int btn, int state, int x, int y)
{
```

ATMECE

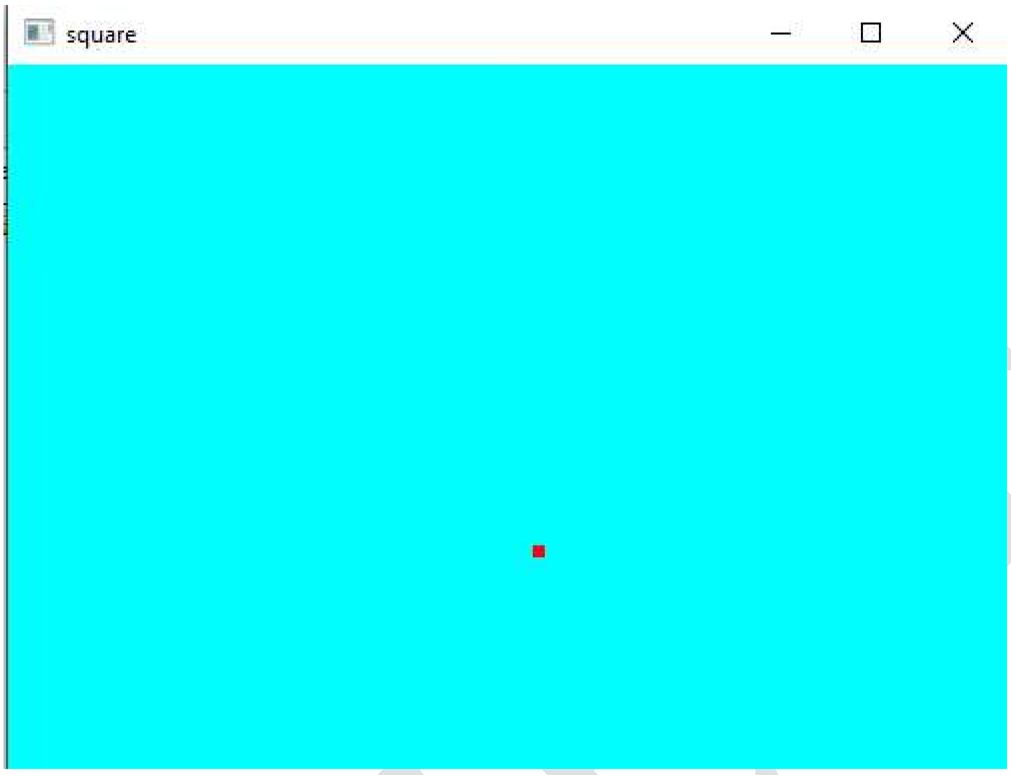
```
        if(btn==GLUT_LEFT_BUTTON && state==GLUT_DOWN)
        drawSquare(x,y);
        if(btn==GLUT_RIGHT_BUTTON && state==GLUT_DOWN)
        exit(0);
}

void myInit()
{
        /* set initial viewing conditions */
        glViewport(0,0,ww,wh);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0, (GLdouble) ww , 0.0, (GLdouble) wh);
        glMatrixMode(GL_MODELVIEW);
        glClearColor (0.0, 0.0, 0.0, 1.0);
        glColor3f(1.0, 0.0, 0.0); /*red squares*/
}

void drawSquare(int x, int y)
{
        y=wh-y;
        glBegin(GL_POLYGON);
        glVertex2f(x+size, y+size);
        glVertex2f(x-size, y+size);
        glVertex2f(x-size, y-size);
        glVertex2f(x+size, y-size);
        glEnd();
        glFlush();
}

void myReshape(GLsizei w, GLsizei h)
{
        /* adjust clipping box */
        glMatrixMode(GL_PROJECTION);
```

ATMECE

```
        glLoadIdentity();
        gluOrtho2D(0.0, (GLdouble)w, 0.0, (GLdouble)h);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        /* adjust viewport and clear */
        glViewport(0,0,w,h);
        /* save new window size in global variables */
        ww=w;
        wh=h;
}

void myKey(unsigned char key, int x, int y)
{
        if(key=='q' || key == 'Q')
        exit(0);
}

int main(int argc, char **argv)
{
        glutInit(&argc,argv);
        glutInitWindowSize(ww, wh); /* globally defined initial window size */
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutCreateWindow("square");
        myInit();
        glutReshapeFunc(myReshape);
        glutMouseFunc(myMouse);
        glutDisplayFunc(myDisplay);
        glutMotionFunc(drawSquare);
        glutKeyboardFunc(myKey);
        glutMainLoop();
}
```

ATMECE

**Output:**



ATMECE

**Module 3**

# Geometric Objects and Transformations

## 3.1 FRAMES IN OPENGL

OpenGL is based on a pipeline model, the first part of which is a sequence of operations on vertices. We can characterize these geometric operations by a sequence of transformations or, equivalently, as a sequence of changes of frames for the objects defined by a user program. In a typical application, there are six frames embedded in the pipeline, although normally the programmer will not see more than a few of them directly. In each of these frames, a vertex has different coordinates. The following is the order that the systems occur in the pipeline:

1. Object or model coordinates
2. World coordinates
3. Eye (or camera) coordinates
4. Clip coordinates
5. Normalized device coordinates
6. Window (or screen) coordinates

In most applications, we tend to specify or use an object with a convenient size, orientation, and location in its own frame called the model or object frame. For example, a cube would typically have its faces aligned with axes of the frame, its center at the origin, and have a side length of 1 or 2 units. The coordinates in the corresponding function calls are in **object or model coordinates**.

The application program generally applies a sequence of transformations to each object to size, orient, and position it within a frame that is appropriate for the particular application. For example, if we were using an instance of a square for a window in an architectural application, we would scale it to have the correct proportions and units, which would probably be in feet or meters. The origin of application coordinates might be a location in the center of the bottom floor of the building. This application frame is called the world frame and the values are in **world coordinates.**

Object and world coordinates are the natural frames for the application program. However, the image that is produced depends on what the camera or viewer sees. Virtually all graphics systems use a frame whose origin is the center of the camera's lens and whose axes are aligned with the sides of the camera. This frame is called the **camera frame or eye frame.** Because there is an affine transformation that corresponds to each change of frame, there are 4 × 4 matrices that represent the transformation from model coordinates to world coordinates and from world coordinates to eye coordinates.

Once objects are in eye coordinates, OpenGL must check whether they lie within the view volume. If an object does not, it is clipped from the scene prior to rasterization. OpenGL can carry out this process most efficiently if

ATMECE

it first carries out a projection transformation that brings all potentially visible objects into a cube centered at the origin in **clip coordinates.**

After this transformation, vertices are still represented in homogeneous coordinates. The division by the w component, called perspective division, yields three-dimensional representations in **normalized device coordinates.**

The final transformation takes a position in normalized device coordinates and, taking into account the viewport, creates a three-dimensional representation in **window coordinates**. Window coordinates are measured in units of pixels on the display but retain depth information. If we remove the depth coordinate, we are working with two-dimensional screen coordinates.

From the application programmer's perspective, OpenGL starts with two frames: the eye frame and the object frame. The model-view matrix positions the object frame relative to the eye frame. Thus, the model-view matrix converts the homogeneous- coordinate representations of points and vectors to their representations in the eye frame. Because the model-view matrix is part of the state of the system, there is always a current camera frame and a current object frame. OpenGL provides matrix stacks, so we can store model-view matrices or, equivalently, frames.

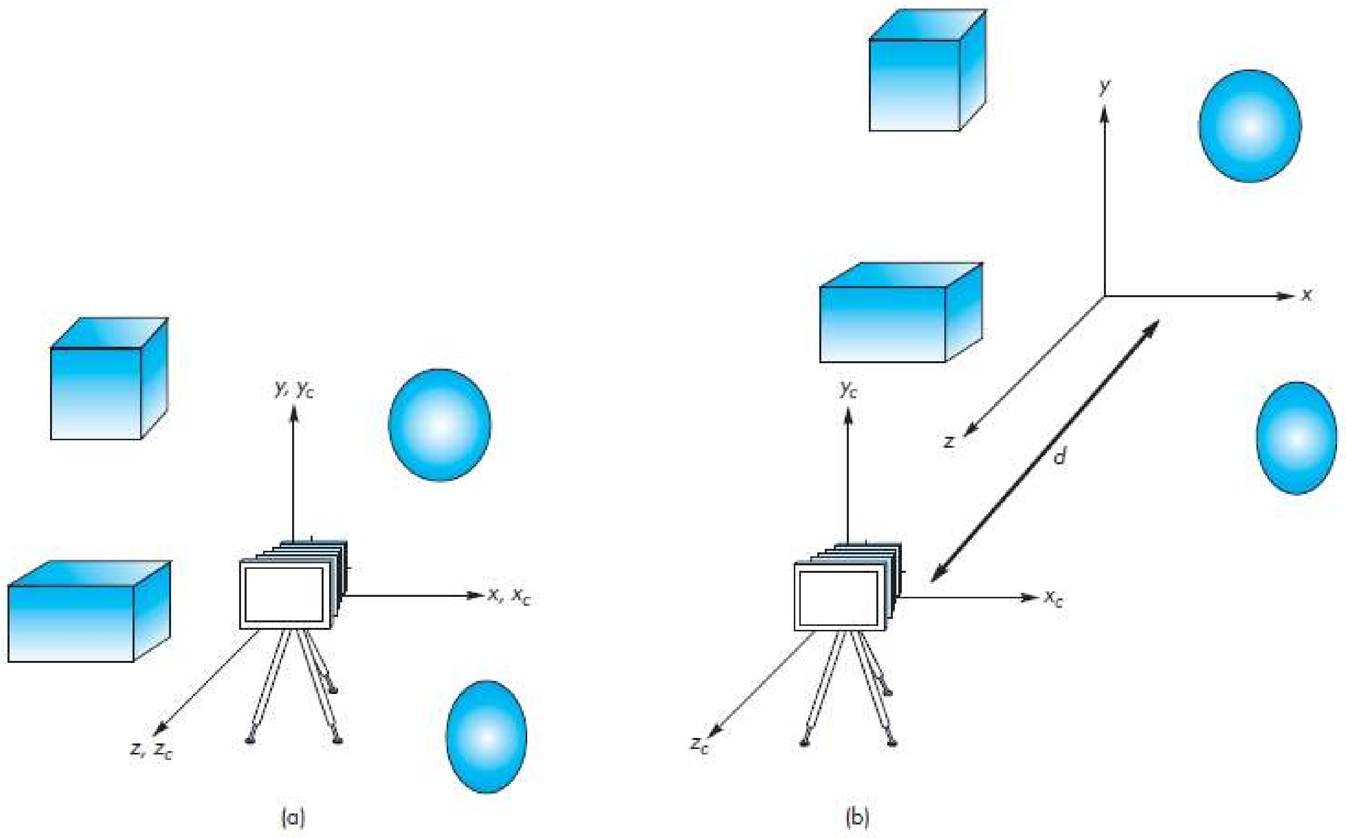


FIGURE 4.26 Camera and object frames. (a) In default positions. (b) After applying model-view matrix.

Initially, the model-view matrix is an identity matrix, so the object frame and eye frame are identical. Thus, if we do not change the model-view matrix, we are working in eye coordinates. As we saw in Chapter 2, the camera is at the origin of its frame, as shown in Figure 4.26(a). The three basis vectors in eye space correspond to (1) the up direction of the camera, the y direction; (2) the direction the camera is pointing, the negative z direction; and (3) a third orthogonal direction, x, placed so that the x, y, z directions form a right-handed coordinate system. We obtain other frames in which to place objects by performing homogeneous coordinate transformations that define new frames relative to the camera frame.

If we regard the camera frame as fixed and the modelview matrix as positioning the object frame relative to the camera frame, then the model-view matrix,

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -d \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

moves a point (x, y, z) in the object frame to the point (x, y, z − d) in the camera frame. Thus, by making d a suitably large positive number, we "move" the objects in front of the camera by moving the world frame relative to the camera frame, as shown in Figure 4.26(b). Note that, as far as the user—who is working in world coordinates is concerned, she is positioning objects as before. The model-view matrix takes care of the relative positioning of the object and eye frames. This strategy is almost always better than attempting to alter the positions of the objects by changing their vertices to place them in front of the camera.

## 3.2 MODELING A COLORED CUBE

One frame of an animation might be as shown in Figure 4.28. However, before we can rotate the cube, we will consider how we can model it efficiently. Although three-dimensional objects can be represented, like two-dimensional objects, through a set of vertices, we will see that data structures will help us to incorporate the relationships among the vertices, edges, and faces of geometric objects. Such data structures are supported in OpenGL through a facility called vertex arrays.

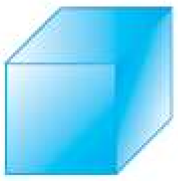


FIGURE 4.28 One frame of cube animation.

ATMECE

Vertices will flow through a number of transformations in the pipeline, all of which will use our homogeneous-coordinate representation. At the end of the pipeline awaits the rasterizer. At this point, we can assume it will do its job automatically, provided we perform the preliminary steps correctly.

### 3.2.1 Modeling the Faces

The cube is as simple a three-dimensional object as we might expect to model and display. There are a number of ways, however, to model it. A CSG system would regard it as a single primitive. At the other extreme, the hardware processes the cube as an object defined by eight vertices. Our decision to use surface-based models implies that we regard a cube either as the intersection of six planes or as the six polygons, called facets, that define its faces. A carefully designed data structure should support both the high-level application view of the cube and the low-level view needed for the implementation.

We start by assuming that the vertices of the cube are available through an array of vertices; for example, we could use the following:

```
GLfloat vertices[8][3] =
   {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
   {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
   {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};
```

We can adopt a more object-oriented formif we first define a three-dimensional point type as follows:

**typedef GLfloat point3[3];**

The vertices of the cube can be specified as follows:

```
point3 vertices[8] ={{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
  {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
  {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};
```

OpenGL represents all vertices internally in four-dimensional homogeneous coordinates. Function calls using a three-dimensional type, such as glVertex3fv, have the values placed into a four-dimensional form within the graphics system.

We can then use the list of points to specify the faces of the cube. For example, one face is

```
glBegin(GL_POLYGON);
  glVertex3fv(vertices[0]);
  glVertex3fv(vertices[3]);
  glVertex3fv(vertices[2]);
  glVertex3fv(vertices[1]);
glEnd();
```

and we can define the other five faces similarly. Note that we have defined three-dimensional polygons with exactly the same mechanism that we used to define two-dimensional polygons.

ATMECE

### 3.2.2 Inward- and Outward-Pointing Faces

We have to be careful about the order in which we specify our vertices when we are defining a three-dimensional polygon. We used the order 0, 3, 2, 1 for the first face. The order 1, 0, 3, 2 would be the same, because the final vertex in a polygon definition is always linked back to the first. However, the order 0, 1, 2, 3 is different. Although it describes the same boundary, the edges of the polygon are traversed in the reverse order—0, 3, 2, 1 as shown in Figure 4.29. The order is important because each polygon has two sides. Our graphics systems can display either or both of them.

From the camera's perspective, we need a consistent way to distinguish between the two faces of a polygon. The order in which the vertices are specified provides this information.

We call a face outward facing if the vertices are traversed in a counter clockwise order when the face is viewed from the outside. This method is also known as the right-hand rule because if you orient the fingers of your right hand in the direction the vertices are traversed, the thumb points outward.

In our example, the order 0, 3, 2, 1 specifies an outward face of the cube whereas the order 0, 1, 2, 3 specifies the back face of the same polygon. Note that each face of an enclosed object, such as our cube, is an inside or outside face, regardless of from where we view it, as long as we view the face from outside the object. By specifying front and back carefully, we will be able to eliminate (or cull) faces that are not visible or to use different attributes to display front and back faces.

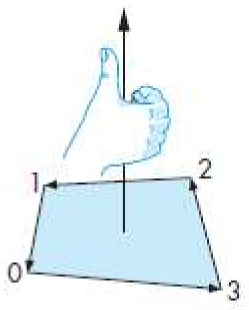


FIGURE 4.29 Traversal of the edges of a polygon.

### 3.2.3 Data Structures for Object Representation

We could now describe our cube through a set of vertex specifications. For example, we could use

**glBegin(GL_POLYGON)**

six times, each time followed by four vertices (via glVertex) and a glEnd, or we could use

**glBegin(GL_QUADS)**

followed by 24 vertices and a glEnd. Both of these methods work, but they both fail to capture the essence of the cube's topology, as opposed to the cube's geometry.

If we think of the cube as a polyhedron, we have an object—the cube—that is composed of six faces. The faces are each quadrilaterals that meet at vertices; each vertex is shared by three faces. In addition, pairs of vertices

ATMECE

define edges of the quadrilaterals; each edge is shared by two faces. These statements describe the topology of a six-sided polyhedron. All are true, regardless of the location of the vertices that is, regardless of the geometry of the object.

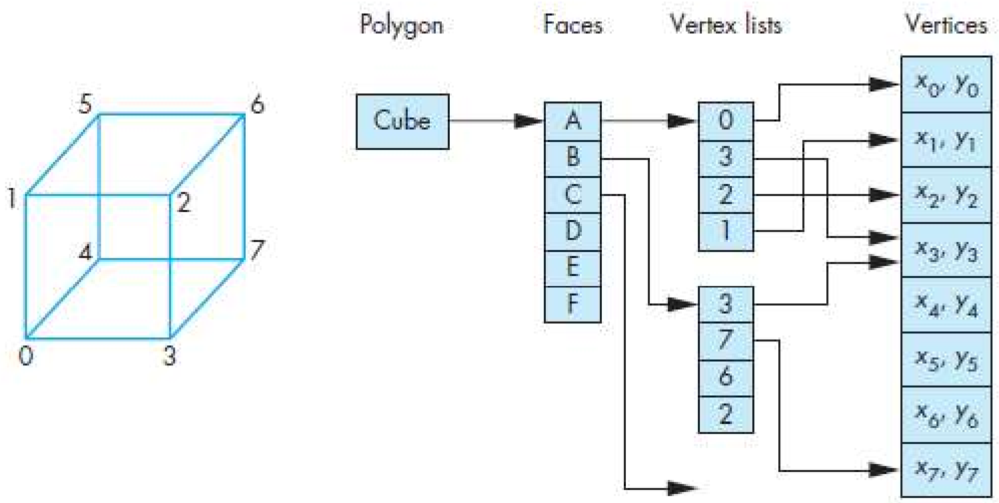


FIGURE 4.30 Vertex-list representation of a cube.

The data specifying the location of the vertices contain the geometry and can be stored as a simple list or array, such as in vertices[8]—the vertex list. The toplevel entity is a cube; we regard it as being composed of six faces. Each face consists of four ordered vertices. Each vertex can be specified indirectly through its index. This data structure is shown in Figure 4.30.

**3.2.4 The Color Cube**

We can use the vertex list to define a color cube. We assign the colors of the vertices of the color solid of (black, white, red, green, blue, cyan, magenta, yellow) to the vertices. We define a function quad to draw quadrilateral polygons specified by pointers into the vertex list. We assign a color for the face using the index of the first vertex. Finally, the colorcube specifies the six faces, taking care to make them all outward-facing as follows:

```
GLfoat vertices[8][3] = {{-1.0,-1.0,1.0},{-1.0,1.0,1.0},
   {1.0,1.0,1.0}, {1.0,-1.0,1.0}, {-1.0,-1.0,-1.0},
   {-1.0,1.0,-1.0}, {1.0,1.0,-1.0}, {1.0,-1.0,-1.0}};

GLfloat colors[8][3] = {{0.0,0.0,0.0},{1.0,0.0,0.0},
   {1.0,1.0,0.0}, {0.0,1.0,0.0}, {0.0,0.0,1.0},
   {1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}};
```

ATMECE

```
void quad(int a, int b, int c, int d)
{
  glBegin(GL_QUADS);
     glColor3fv(colors[a]);
     glVertex3fv(vertices[a]);
     glColor3fv(colors[b]);
     glVertex3fv(vertices[b]);
     glColor3fv(colors[c]);
     glVertex3fv(vertices[c]);
     glColor3fv(colors[d]);
     glVertex3fv(vertices[d]);
  glEnd();
}

void colorcube()
{
   quad(0,3,2,1);
   quad(2,3,7,6);
   quad(0,4,7,3);
   quad(1,2,6,5);
   quad(4,5,6,7);
   quad(0,1,5,4);
}
```

### 3.2.5 Vertex Arrays

Although we used vertex lists to model our cube, when we draw the cube we are making many calls to OpenGL functions. If we assign a color to each vertex, we make 60 OpenGL calls: six faces, each of which needs a glBegin, a glEnd, four calls to glColor, and four calls to glVertex. Each call involves overhead and data transfer, we also specify a different normal vector at each vertex, we make even more function calls to draw our cube. Hence, although we have used a data structure that helps us to conceptualize the cube as a three dimensional geometric object, the code to draw it may not execute quickly.

Vertex arrays provide a method for encapsulating the information in our data structure such that we can draw polyhedral objects with only a few function calls.

They allow us to define a data structure using vertices and pass this structure to the implementation. When the objects defined by these arrays need to be drawn, we can ask OpenGL to traverse the structure with just a few function calls.

There are three steps in using vertex arrays. First, we enable the functionality of vertex arrays. Second, we tell OpenGL where and in what format the arrays are. Third, we render the object. The first two steps can be part of the initialization; the third is typically part of the display callback.We illustrate for the cube.

OpenGL allows many different types of arrays, including vertex, color, color index, normal, and texture coordinate, corresponding to items that can be set between a glBegin and a glEnd. Any given application usually requires only a subset of these types. For our rotating cube, we need only colors and vertices. We enable the corresponding arrays as follows:

ATMECE

```
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
```

Note that, unlike most OpenGL state information, the information for vertex arrays resides on the client side, not the server side—hence the function name glEnable- ClientState. The arrays are the same as before and can be set up as globals:

```
GLfloat vertices[8][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
{1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
{1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};

GLfloat colors[8][3] = {{0.0,0.0,0.0},{1.0,0.0,0.0},
{1.0,1.0,0.0}, {0.0,1.0,0.0}, {0.0,0.0,1.0},
{1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}};
```

Next, we identify where the arrays are as follows:

```
glVertexPointer(3, GL_FLOAT, 0, vertices);
glColorPointer(3, GL_FLOAT, 0, colors);
```

The first three arguments state that the elements are three-dimensional colors and vertices stored as floats and that the elements are contiguous in the arrays. The fourth argument is a pointer to the array holding the data.
Next, we have to provide the information in our data structure about the relationship between the vertices and the faces of the cube. We do so by specifying an array that holds the 24 ordered vertex indices for the six faces:

```
GLubyte cubeIndices[24] = {0,3,2,1,   /* Face 0 */
                           2,3,7,6,   /* Face 1 */
                           0,4,7,3,   /* Face 2 */
                           1,2,6,5,   /* Face 3 */
                           4,5,6,7,   /* Face 4 */
                           0,1,5,4};  /* Face 5 */
```

Thus, the first face is determined by the indices (0, 3, 2, 1), the second by (2, 3, 7, 6), and so on.Note that we have put the indices in an order that preserves outward-facing polygons. The index array can also be specified as a global.
Now we can render the cube through use of the arrays. When we draw elements of the arrays, all the enabled arrays (in this example, colors and vertices) are rendered. We have a few options regarding how to draw the arrays. We can use the following function:

**glDrawElements(GLenum type, GLsizei n, GLenum format, void *pointer)**

For our cube, within the display callback we could make six calls to glDraw- Elements, one for each face as follows:

```
for(i=0;i<6;i++)
   glDrawElements(GL_POLYGON, 4, GL_UNSIGNED_BYTE, &cubeIndices[4*i]);
```

Thus, once we have set up and initialized the arrays, each time that we draw the cube, we have only six function calls. We can rotate the cube as before; glDrawElements uses the present state when it draws the cube.
We can do even better though. Each face of the cube is a quadrilateral. Thus, if we use GL_QUADS, rather than GL_POLYGON, we can draw the cube with the single function call

**glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, cubeIndices);**

because GL_QUADS starts a new quadrilateral after each four vertices.

## 3.3 AFFINE TRANSFORMATIONS

A transformation is a function that takes a point (or vector) and maps that point (or vector) into another point (or vector). We can picture such a function by looking at Figure 4.33 or by writing down the functional form

$$\mathbf{Q = T(P)}$$

for points, or

$$\mathbf{v = R(u)}$$

for vectors. If we can use homogeneous coordinates, then we can represent both vectors and points as four-dimensional column matrices and we can define the transformation with a single function,

$$\mathbf{q = f\,(p),}$$
$$\mathbf{v = f\,(u),}$$

that transforms the representations of both points and vectors in a given frame.
This formulation is too general to be useful, as it encompasses all single-valued mappings of points and vectors. In practice, even if we were to have a convenient description of the function f , we would have to carry out the transformation on every point on a curve. For example, if we transform a line segment, a general transformation might require us to carry out the transformation for every point between the two endpoints.
Consider instead a restricted class of transformations. Let's assume that we are working in four-dimensional, homogeneous coordinates. In this space, both points and vectors are represented as 4-tuples.6 We can obtain a useful class of transformations if we place restrictions on f . The most important restriction is linearity. A function f is a linear function if and only if, for any scalars α and β and any two vertices (or vectors) p and q,

$$\mathbf{f\,(\alpha p + \beta q) = \alpha f\,(p) + \beta f\,(q).}$$

Using homogeneous coordinates, we work with the representations of points and vectors. A linear transformation then transforms the representation of a given point (or vector) into another representation of that point (or vector) and can always be written in terms of the two representations, u and v, as a matrix multiplication: **v = Cu,** where C is a square matrix. Comparing this expression with the expression.
When we work with homogeneous coordinates, A is a 4 × 4 matrix that leaves unchanged the fourth (w) component of a representation. The matrix C is of the form

ATMECE

$$\mathbf{C} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and is the transpose of the matrix M that we derived in Section 4.3.4. The 12 values can be set arbitrarily, and we say that this transformation has 12 degrees of freedom. However, points and vectors have slightly different representations in our affine space. Any vector is represented as

$$\mathbf{u} = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ 0 \end{bmatrix}.$$

Any point can be written as

$$\mathbf{p} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ 1 \end{bmatrix}.$$

If we apply an arbitrary C to a vector,

$$\mathbf{v} = \mathbf{C}\mathbf{u},$$

ATMECE

## 3.4 Transformations: TRANSLATION, ROTATION, AND SCALING

In OpenGL, transformations are tools that allow you to move, rotate, and resize shapes. They are accomplished through matrix transformations that move 3D points into 2D coordinates on the screen**.**

### 3.4.1 Translation

Translation is an operation that displaces points by a fixed distance in a given direction, as shown in Figure 4.34. To specify a translation, we need only to specify a displacement vector d, because the transformed points are given by **P' = P + d**

for all points P on the object. Note that this definition of translation makes no reference to a frame or representation. Translation has three degrees of freedom because we can specify the three components of the displacement vector arbitrarily**.**

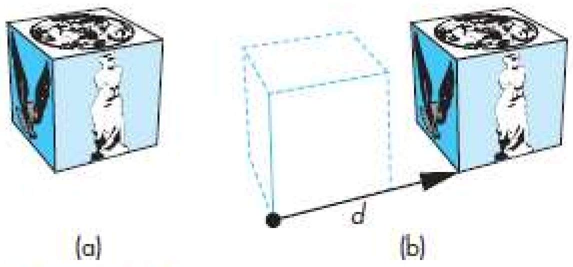


FIGURE 4.34 Translation. (a) Object in original position. (b) Object translated.

### 3.4.2 Rotation

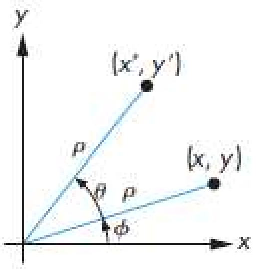


FIGURE 4.35 Two-dimensional rotation.

Rotation is more difficult to specify than translation because we must specify more parameters. We start with the simple example of rotating a point about the origin in a two-dimensional plane, as shown in Figure 4.35. Having specified a particular point the origin we are in a particular frame. A two-dimensional point at (x, y) in this frame is rotated about the origin by an angle θ to the position (x' , y' ). We can obtain the standard equations describing this rotation by representing (x, y) and (x' , y' ) in polar form:

$$x = \rho \cos \phi,$$

$$y = \rho \sin \phi,$$

$$x' = \rho \cos(\theta + \phi),$$

$$y' = \rho \sin(\theta + \phi).$$

Expanding these terms using the trigonometric identities for the sine and cosine of the sum of two angles, we find

$$x' = \rho \cos \phi \cos \theta - \rho \sin \phi \sin \theta = x \cos \theta - y \sin \theta,$$

$$y' = \rho \cos \phi \sin \theta + \rho \sin \phi \cos \theta = x \sin \theta + y \cos \theta.$$

These equations can be written in matrix form as

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

ATMECE

Note three features of this transformation that extend to other rotations:

1. There is one point—the origin, in this case—that is unchanged by the rotation. We call this point the fixed point of the transformation. Figure 4.36 shows a two-dimensional rotation about a fixed point in the center of the object rather than about the origin of the frame.
2. Knowing that the two-dimensional plane is part of three-dimensional space, we can reinterpret this rotation in three dimensions. In a right-handed system, when we draw the x- and y-axes in the standard way, the positive z-axis comes out of the page. Our definition of a positive direction of rotation is counter-clockwise when we look down the positive z-axis toward the origin. We use this definition to define positive rotations about other axes.
3. Rotation in the two-dimensional plane $z = 0$ is equivalent to a three-dimensional rotation about the z-axis. Points in planes of constant z all rotate in a similar manner, leaving their z values unchanged.

Rotation and translation are known as rigid-body transformations. No combination of rotations and translations can alter the shape or volume of an object; the can alter only the object's location and orientation. Consequently, rotation and translation alone cannot give us all possible affine transformations. The transformation shown in Figure 4.38 are affine, but they are not rigid-body transformations.

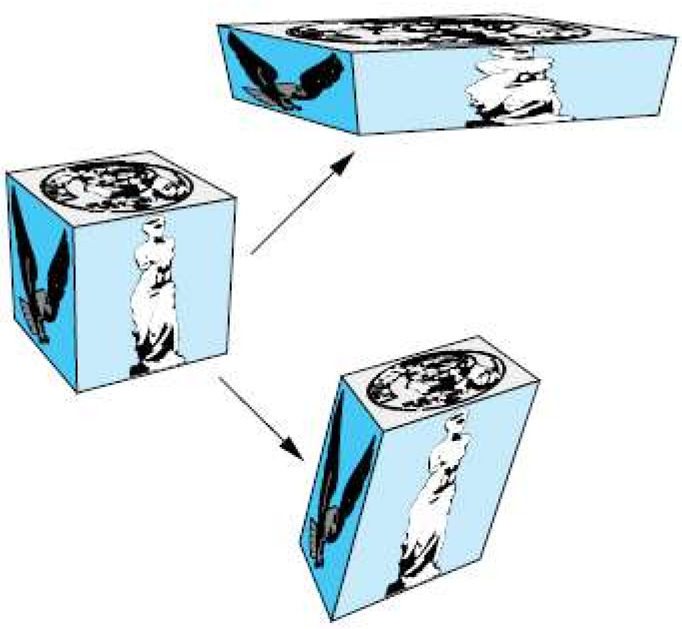


FIGURE 4.38 Non–rigid-body transformations.

### 3.4.3 Scaling

Scaling is an affine non–rigid-body transformation by which we can make an object bigger or smaller. Figure 4.39 illustrates both uniform scaling in all directions and scaling in a single direction. We need nonuniform scaling to build up the full set of affine transformations that we use in modeling and viewing by combining a properly chosen sequence of scalings, translations, and rotations.

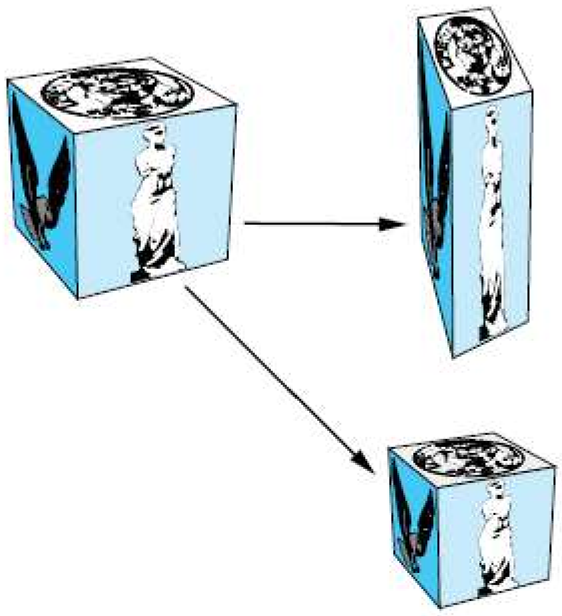


FIGURE 4.39 Uniform and nonuniform scaling.

Scaling transformations have a fixed point, as we can see from Figure 4.40. Hence, to specify a scaling, we can specify the fixed point, a direction in which we wish to scale, and a scale factor (α). For α >1, the object gets longer in the specified direction; for 0 ≤α <1, the object gets smaller in that direction. Negative values of α give us reflection (Figure 4.41) about the fixed point, in the scaling direction. Scaling has six degrees of freedom because we can specify an arbitrary fixed point and three independent scaling factors.

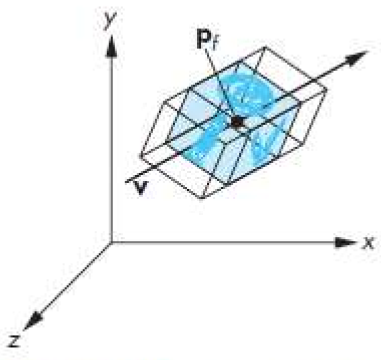


FIGURE 4.40 Effect of scale factor.

ATMECE

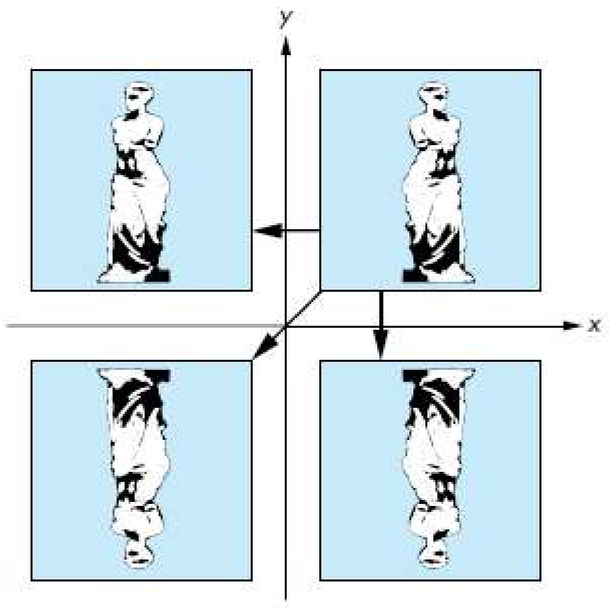


FIGURE 4.41 Reflection.

ATMECE

## 3.5 TRANSFORMATIONS IN HOMOGENEOUS COORDINATES

All graphics APIs force us to work within some reference system. Hence, we cannot work with high-level expressions such as **Q = P + αv.**

Instead, we work with representations in homogeneous coordinates and with expressions such as **q = p + αv.**

Within a frame, each affine transformation is represented by a 4 × 4 matrix of the form

$$\mathbf{A} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

### 3.5.1 Translation

Translation displaces points to new positions defined by a displacement vector. If we move the point p to p' by displacing by a distance d, then **p' = p + d.**

Looking at their homogeneous-coordinate forms

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}, \qquad \mathbf{p}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}, \qquad \mathbf{d} = \begin{bmatrix} \alpha_x \\ \alpha_y \\ \alpha_z \\ 0 \end{bmatrix},$$

we see that these equations can be written component by component as

$$x' = x + \alpha_x,$$
$$y' = y + \alpha_y,$$
$$z' = z + \alpha_z.$$

This method of representing translation using the addition of column matrices does not combine well with our representations of other affine transformations. However, we can also get this result using the matrix multiplication: **p' = Tp,**

where

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

T is called the translation matrix. We sometimes write it as T(αx , αy , αz) to emphasize the three independent parameters. It might appear that using a fourth fixed element in the homogeneous representation of a point is not necessary. However, if we use the three-dimensional forms

$$\mathbf{q} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \qquad \mathbf{q}' = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix},$$

it is not possible to find a 3×3 matrix D such that **q' = Dq** for the given displacement vector d. For this reason, the use of homogeneous coordinates is often seen as a clever trick that allows us to convert the addition of column matrices in three dimensions to matrix–matrix multiplication in four dimensions.

We can obtain the inverse of a translation matrix either by applying an inversion algorithm or by noting that if we displace a point by the vector d, we can return to the original position by a displacement of −d. By either method, we find that

$$\mathbf{T}^{-1}(\alpha_x, \alpha_y, \alpha_z) = \mathbf{T}(-\alpha_x, -\alpha_y, -\alpha_z) = \begin{bmatrix} 1 & 0 & 0 & -\alpha_x \\ 0 & 1 & 0 & -\alpha_y \\ 0 & 0 & 1 & -\alpha_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

### 3.5.2 Scaling

For both scaling and rotation, there is a fixed point that is unchanged by the transformation. We let the fixed point be the origin, and we show how we can concatenate transformations to obtain the transformation for an arbitrary fixed point.

A scaling matrix with a fixed point of the origin allows for independent scaling along the coordinate axes. The three equations are

ATMECE

$$x' = \beta_x;$$

$$y' = \beta_y;$$

$$z' = \beta_z;$$

These three equations can be combined in homogeneous form as **p' = Sp,**

where

$$\mathbf{S} = \mathbf{S}(\beta_x, \beta_y, \beta_z) = \begin{bmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

As is true of the translation matrix and, indeed, of all homogeneous coordinate transformations, the final row of the matrix does not depend on the particular transformation, but rather forces the fourth component of the transformed point to retain the value 1.

We obtain the inverse of a scaling matrix by applying the reciprocals of the scale factors:

$$\mathbf{S}^{-1}(\beta_x, \beta_y, \beta_z) = \mathbf{S}\left(\frac{1}{\beta_x}, \frac{1}{\beta_y}, \frac{1}{\beta_z}\right)$$

## 3.5.3 Rotation

We first look at rotation with a fixed point at the origin. There are three degrees of freedom corresponding to our ability to rotate independently about the three coordinate axes. We have to be careful, however, because matrix multiplication is not a commutative operation (Appendix C). Rotation about the x-axis by an angle θ followed by rotation about the y-axis by an angle φ does not give us the same result as the one that we obtain if we reverse the order of the rotations.

We can find the matrices for rotation about the individual axes directly from the results of the two-dimensional rotation. We saw that the two-dimensional rotation was actually a rotation in three dimensions about the z-axis and that the points remained in planes of constant z. Thus, in three dimensions, the equations for rotation about the z-axis by an angle θ are

$$x' = x\cos\theta - y\sin\theta,$$

$$y' = x\sin\theta + y\cos\theta,$$

$$z' = z;$$

or, in matrix form,

$$\mathbf{p}' = \mathbf{R}_z\mathbf{p},$$

where

$$R_z = R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We can derive the matrices for rotation about the x- and y-axes through an identical argument. If we rotate about the x-axis, then the x values are unchanged, and we have a two-dimensional rotation in which points rotate in planes of constant x; for rotation about the y-axis, the y values are unchanged. The matrices are

$$R_x = R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$R_y = R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The signs of the sine terms are consistent with our definition of a positive rotation in a right-handed system. Suppose that we let R denote any of our three rotation matrices. A rotation by θ can always be undone by a subsequent rotation by −θ; hence,

$$R^{-1}(\theta) = R(-\theta).$$

In addition, noting that all the cosine terms are on the diagonal and the sine terms are off-diagonal, we can use the trigonometric identities

$$\cos(-\theta) = \cos\theta$$

$$\sin(-\theta) = -\sin\theta$$

to find

$$R^{-1}(\theta) = R^T(\theta).$$

as a product of individual rotations about the three axes

$$R = R_z R_y R_x$$

Using the fact that the transpose of a product is the product of the transposes in the reverse order, we see that for any rotation matrix, $R^{-1} = R^T$.

A matrix whose inverse is equal to its transpose is called an **orthogonal matrix.** Normalized orthogonal matrices correspond to rotations about the origin.

ATMECE

## 3.5.4 Shear

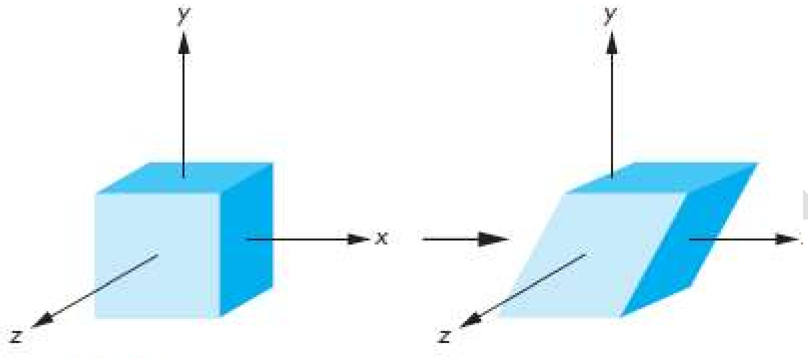


FIGURE 4.42 Shear.

Although we can construct any affine transformation from a sequence of rotations, translations, and scalings, there is one more affine transformation **the shear** transformation—that is of such importance that we regard it as a basic type, rather than deriving it from the others. Consider a cube centered at the origin, aligned with the axes and viewed from the positive z-axis, as shown in Figure 4.42. If we pull the top to the right and the bottom to the left, we shear the object in the x direction. Note that neither the y nor the z values are changed by the shear, so we can call this operation x shear to distinguish it from shears of the cube in other possible directions. Using simple trigonometry on Figure 4.43, we see that each shear is characterized by a single angle θ; the equations for this shear are

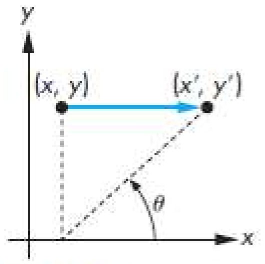


FIGURE 4.43 Computation of the shear matrix.

$$x' = x + y \cot\theta,$$

$$y' = y,$$

$$z' = z,$$

leading to the shearing matrix

$$\mathbf{H}_x(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

We can obtain the inverse by noting that we need to shear in only the opposite direction; hence, $\mathbf{H}_x^{-1}(\theta) = \mathbf{H}_x(-\theta$

ATMECE

## 3.6 CONCATENATION OF TRANSFORMATIONS

We create examples of affine transformations by multiplying together, or concatenating, sequences of the basic transformations that we just introduced. Using this strategy is preferable to attempting to define an arbitrary transformation directly. The approach fits well with our pipeline architectures for implementing graphics systems. Suppose that we carry out three successive transformations on a point p, creating a new point q. Because the matrix product is associative, we can write the sequence as **q = CBAp,**

without parentheses. Note that here the matrices A, B, and C (and thus M) can be arbitrary 4×4 matrices, although in practice they will most likely be affine. The order in which we carry out the transformations affects the efficiency of the calculation. In one view, shown in Figure 4.44, we can carry out A, followed by B, followed by C an order that corresponds to the grouping **q = (C(B(Ap))).**

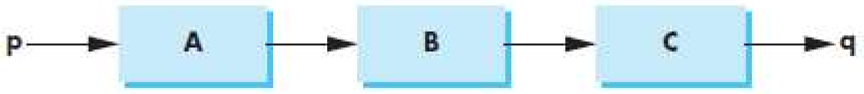


FIGURE 4.44 Application of transformations one at a time.

If we are to transform a single point, this order is the most efficient because each matrix multiplication involves multiplying a column matrix by a square matrix. If we have many points to transform, then we can proceed in two steps. First, we calculate

$$\mathbf{M = CBA.}$$

Then, we use this matrix on each point

$$\mathbf{q = Mp.}$$

This order corresponds to the pipeline shown in Figure 4.45, where we compute M first, then load it into a pipeline transformation unit. If we simply count operations, we see that although we do a little more work in computing M initially, because M may be applied to tens of thousands of points, this extra work is insignificant compared with the savings we obtain by using a single matrix multiplication for each point. We now derive examples of computing **M.**

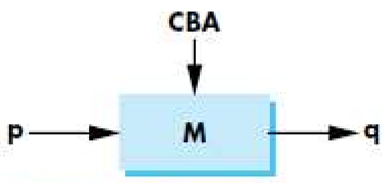


FIGURE 4.45 Pipeline transformation.

### 3.6.1 Rotation About a Fixed Point

Consider a cube with its center at pf and its sides aligned with the axes. We want to rotate the cube about the z-axis, but this time about its center pf , which becomes the fixed point of the transformation, as shown in Figure 4.46. If pf were the origin, we would know how to solve the problem: We would simply use Rz(θ). This observation suggests the strategy of first moving the cube to the origin. We can then apply Rz(θ) and finally move the object back such that its center is again at pf .

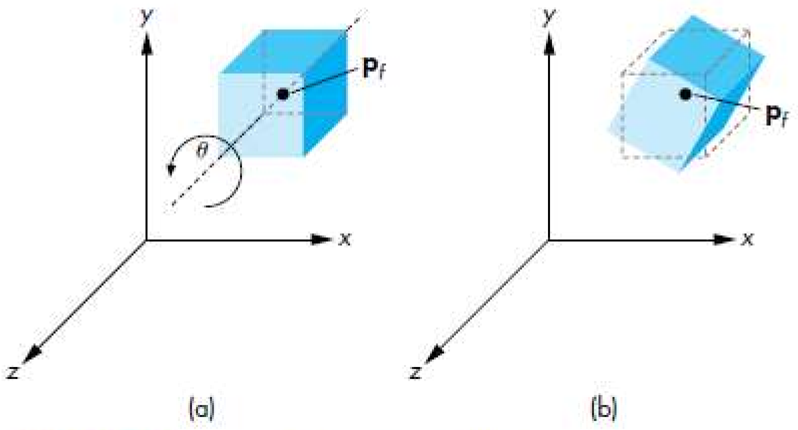


FIGURE 4.46 Rotation of a cube about its center.

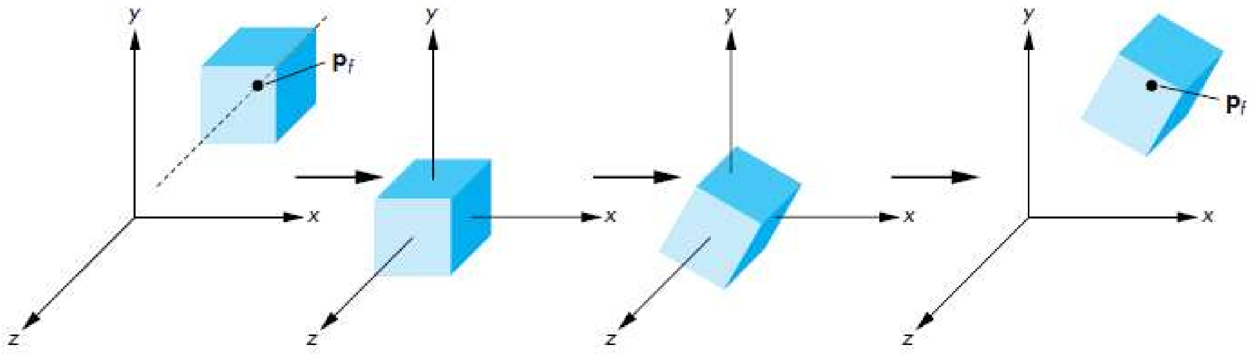


FIGURE 4.47 Sequence of transformations.

This sequence is shown in Figure 4.47. In terms of our basic affine transformations, the first is T(−pf ), the second is Rz(θ), and the final is T(pf ). Concatenating them together, we obtain the single matrix

$$\mathbf{M} = \mathbf{T}(\mathbf{p}_f)\mathbf{R}_z(\theta)\mathbf{T}(-\mathbf{p}_f).$$

If we multiply out the matrices, we find that

$$\mathbf{M} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & x_f - x_f\cos\theta + y_f\sin\theta \\ \sin\theta & \cos\theta & 0 & y_f - x_f\sin\theta - y_f\cos\theta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

### 3.6.2 General Rotation

We now show that an arbitrary rotation about the origin can be composed of three successive rotations about the three axes. The order is not unique, although the resulting rotation matrix is. We form the desired matrix by first doing a rotation about the z-axis, then doing a rotation about the y-axis, and concluding with a rotation about the x-axis. Consider the cube, again centered at the origin with its sides aligned with the axes, as shown in Figure 4.48(a). We can rotate it about the z-axis by an angle α to orient it, as shown in Figure 4.48(b).We then rotate the cube by an angle β about the y-axis, as shown in a top view in Figure 4.49. Finally, we rotate the cube by an angle γ about the x-axis, as shown in a side view in Figure 4.50. Our final rotation matrix is **R = RxRyRz** .

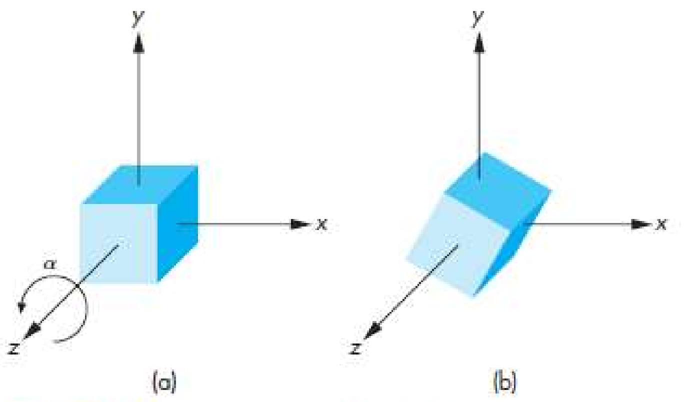


FIGURE 4.48 Rotation of a cube about the *z*-axis. (a) Cube before rotation. (b) Cube after rotation.

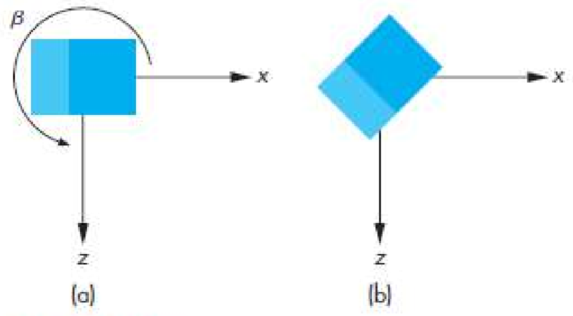


FIGURE 4.49 Rotation of a cube about the *y*-axis.

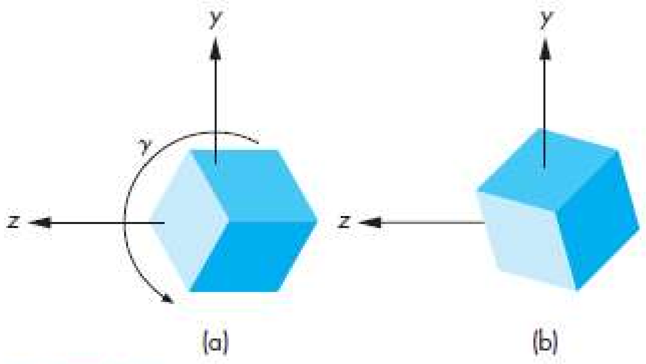


FIGURE 4.50 Rotation of a cube about the *x*-axis.

ATME

## 3.6.3 The Instance Transformation

Our example of a cube that can be rotated to any desired orientation suggests a generalization appropriate for modeling. Consider a scene composed of many simple objects, such as that shown in Figure 4.51. One option is to define each of these objects, through its vertices, in the desired location with the desired orientation and size. An alternative is to define each of the object types once at a convenient size, in a convenient place, and with a convenient orientation. Each occurrence of an object in the scene is an instance of that object's prototype, and we can obtain the desired size, orientation, and location by applying an affine transformation the instance transformation to the prototype. We can build a simple database to describe a scene from a list of object identifiers (such as 1 for a cube and 2 for a sphere) and of the instance transformation to be applied to each object.

The instance transformation is applied in the order shown in Figure 4.52.Objects are usually defined in their own frames, with the origin at the center of mass and the sides aligned with the model frame axes. First, we scale the object to the desired size. Then we orient it with a rotation matrix. Finally, we translate it to the desired orientation. Hence, the instance transformation is of the form **M = TRS.**

Modeling with the instance transformation works well not only with our pipeline architectures but also with the display lists. A complex object that is used many times can be loaded into the server once as a display list.

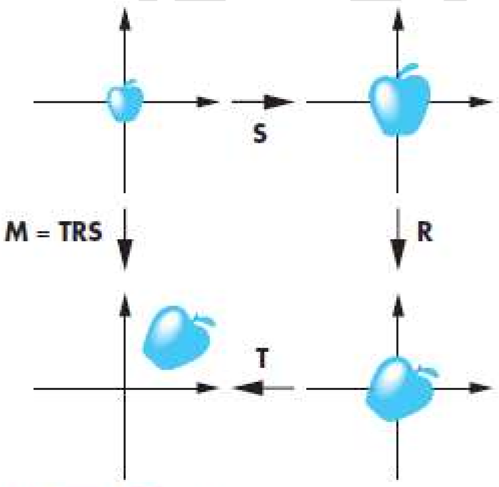


FIGURE 4.52 Instance transformation.

Displaying each instance of it requires only sending the appropriate instance transformation to the server before executing the display list.

## 3.6.4 Rotation About an Arbitrary Axis

Consider rotating a cube, as shown in Figure 4.53.We need three entities to specify this rotation. There is a fixed point p0 that we assume is the center of the cube, a vector about which we rotate, and an angle of rotation.Note that none of these entities relies on a frame and that we have just specified a rotation in a coordinatefree manner.

Nonetheless, to find an affine matrix to represent this transformation, we have to assume that we are in some frame.

The vector about which we wish to rotate the cube can be specified in various ways. One way is to use two points, p1 and p2, defining the vector **u = p2− p1.**

Note that the order of the points determines the positive direction of rotation for θ and that even though we draw u as passing through p0, only the orientation of u matters. Replacing u with a unit-length vector

$$\mathbf{v} = \frac{\mathbf{u}}{|\mathbf{u}|} = \begin{bmatrix} \alpha_x \\ \alpha_y \\ \alpha_z \end{bmatrix}$$

in the same direction simplifies the subsequent steps. We have already seen that moving the fixed point to the origin is a helpful technique. Thus, our first transformation is the translation T(−p0), and the final one is T(p0). After the initial translation, the required rotation problem is as shown in Figure 4.54.

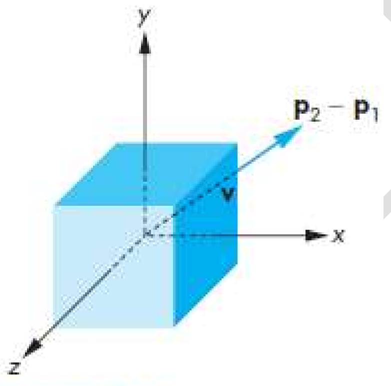


FIGURE 4.54 Movement of the fixed point to the origin.

Our previous example (see Section 4.9.2) showed that we could get an arbitrary rotation from three rotations about the individual axes. This problem is more difficult because we do not know what angles to use for the individual rotations. Our strategy is to carry out two rotations to align the axis of rotation, v, with the z-axis. Then we can rotate by θ about the z-axis, after which we can undo the two rotations that did the aligning. Our final rotation matrix will be of the form

$$\mathbf{R} = \mathbf{R}_x(-\theta_x)\mathbf{R}_y(-\theta_y)\mathbf{R}_z(\theta)\mathbf{R}_y(\theta_y)\mathbf{R}_x(\theta_x).$$

ATMECE

**OpenGL program to model a colour cube with translation and rotation.**

```
#include<GL/glut.h>
#include<stdio.h>

GLfloat vertices[][3]={{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},{1.0,1.0,-1.0},{-1.0,1.0,-1.0},
{-1.0,-1.0,1.0},{1.0,-1.0,1.0},{1.0,1.0,1.0},{-1.0,1.0,1.0}};

GLfloat colors[][3]={{0.0,0.0,0.0},{0.0,0.0,1.0},{0.0,1.0,0.0},{0.0,1.0,1.0},{1.0,0.0,0.0},
{1.0,0.0,1.0},{1.0,1.0,0.0},{1.0,1.0,1.0}};

static GLfloat theta[]={0.0,0.0,0.0};
static GLint axis=2;
static GLdouble viewer[]={0,0,5};

void polygon(int a,int b,int c,int d)
{
        glBegin(GL_POLYGON);
        glColor3fv(colors[a]);
        glVertex3fv(vertices[a]);
        glColor3fv(colors[b]);
        glVertex3fv(vertices[b]);
        glColor3fv(colors[c]);
        glVertex3fv(vertices[c]);
        glColor3fv(colors[d]);
        glVertex3fv(vertices[d]);
        glEnd();
}

void colorcube()
{
        polygon(0,3,2,1);
        polygon(2,3,7,6);
        polygon(1,2,6,5);
        polygon(0,4,5,1);
```

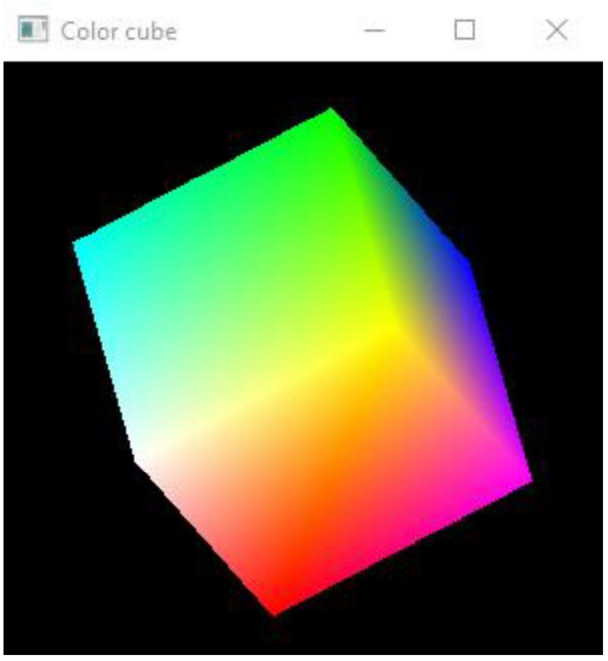
ATMECE

```
        polygon(4,5,6,7);
        polygon(0,3,7,4);
}

void display()
{
        glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        glRotatef(theta[0],1.0,0.0,0.0);
        glRotatef(theta[1],0.0,1.0,0.0);
        glRotatef(theta[2],0.0,0.0,1.0);
        colorcube();
        glFlush();
        glutSwapBuffers();
}

void spincube()
{
        theta[axis]+=2.0;
        if(theta[axis]>360.0)
        theta[axis]-=360.0;
        glutPostRedisplay();
}

void mouse(int btn,int state,int x,int y)
{
        if(btn==GLUT_LEFT_BUTTON&&state==GLUT_DOWN)
        axis=0;
        if(btn==GLUT_MIDDLE_BUTTON&&state==GLUT_DOWN)
        axis=1;
        if(btn==GLUT_RIGHT_BUTTON&&state==GLUT_DOWN)
        axis=2;
        spincube();
}
```

ATMECE

```
void myreshape(int w,int h)
{
        glViewport(0,0,w,h);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        if(w<=h)
        glOrtho(-2.0,2.0,-2.0*(GLfloat)h/(GLfloat)w,2.0*(GLfloat)h/(GLfloat)w,-10.0,10.0);
        else
        glOrtho(-2.0*(GLfloat)w/(GLfloat)h,2.0*(GLfloat)w/(GLfloat)h,-2.0,2.0,-10.0,10.0);
        glMatrixMode(GL_MODELVIEW);
        glutPostRedisplay();
}

int main(int argc,char**argv)
{
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH);
        glutCreateWindow("Color cube");
        glutReshapeFunc(myreshape);
        glutDisplayFunc(display);
        //glutIdleFunc(spincube);
        glutMouseFunc(mouse);
        glEnable(GL_DEPTH_TEST);
        glClearColor(0.0,0.0,0.0,0.0);
        glutMainLoop();
}
```

ATMECE



**OpenGL program to rotate a triangle around the origin and fixed arbitrary point.**

```
#include<GL/glut.h>
#include<stdio.h>

int x,y;
int where_to_rotate=0;
float rotate_angle=0;
float translate_x=0,translate_y=0;

void draw_pixel(float x1, float y1)
{
        glPointSize(5);
        glBegin(GL_POINTS);
        glVertex2f(x1,y1);
        glEnd();
}

void triangle(int x, int y)
{
        glColor3f(1,0,0);
        glBegin(GL_POLYGON);
        glVertex2f(x,y);
        glVertex2f(x+400,y+300);
        glVertex2f(x+300,y+0);
        glEnd();
}

void display()
{
        glClear(GL_COLOR_BUFFER_BIT);
        glLoadIdentity();
        glColor3f(1,1,1);
        draw_pixel(0,0);
        if (where_to_rotate == 1)
```

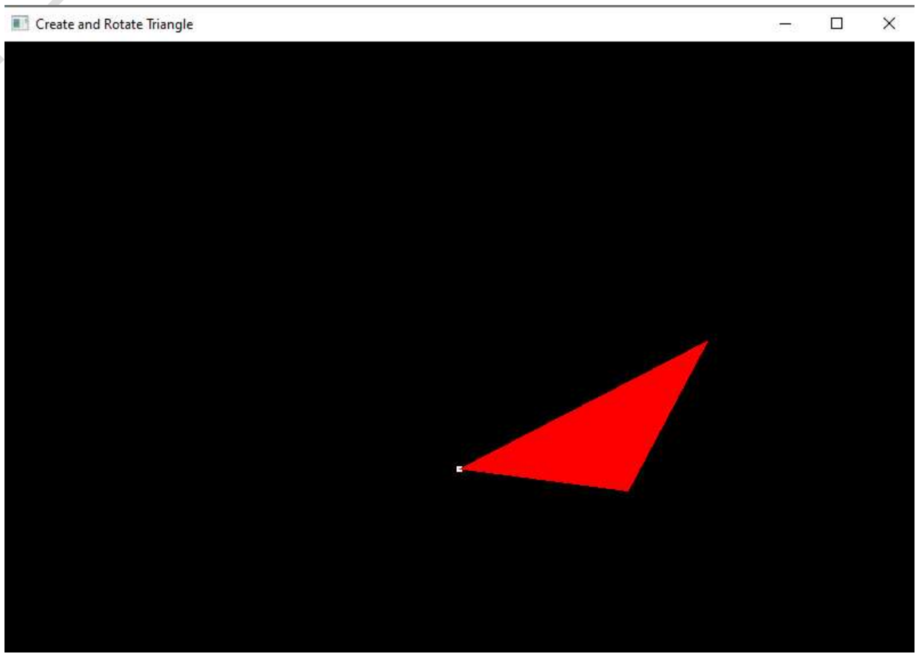
ATMECE

```
        {
                translate_x = 0;
                translate_y = 0;
                rotate_angle += 1;
        }
        if (where_to_rotate == 2)
        {
                translate_x = x;
                translate_y = y;
                rotate_angle += 1;
                glColor3f(0,0,1);
                draw_pixel(x,y);
        }
        glTranslatef(translate_x, translate_y, 0);
        glRotatef(rotate_angle, 0, 0, 1);
        glTranslatef(-translate_x, -translate_y, 0);
        triangle(translate_x,translate_y);
        glutPostRedisplay();
        glutSwapBuffers();
}

void init()
{
        glClearColor(0,0,0,1);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(-800, 800, -800, 800);
        glMatrixMode(GL_MODELVIEW);
}

void rotateMenu (int option)
{
        if(option==1)
        where_to_rotate=1;
```

ATMECE

```
        if(option==2)
        where_to_rotate=2;
        if(option==3)
        where_to_rotate=3;
}

int main(int argc, char **argv)
{
        printf( "Enter Fixed Points (x,y) for Rotation: \n");
        scanf("%d %d", &x, &y);
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
        glutInitWindowSize(800, 800);
        glutInitWindowPosition(0, 0);
        glutCreateWindow("Create and Rotate Triangle");
        init();
        glutDisplayFunc(display);
        glutCreateMenu(rotateMenu);
        glutAddMenuEntry("Rotate around ORIGIN",1);
        glutAddMenuEntry("Rotate around FIXED POINT",2);
        glutAddMenuEntry("Stop Rotation",3);
        glutAttachMenu(GLUT_RIGHT_BUTTON);
        glutMainLoop();
}
```

ATMECE


Create and Rotate Triangle