We now turn to the development of interactive graphics programs. Interactive computer graphics opens up a myriad of applications, ranging from interactive design of buildings, to control of large systems through graphical interfaces, to virtual-reality systems, to computer games.

Our discussion has three main parts. First, we introduce the variety of devices available for interaction. We consider input devices from two different perspectives: (1) the way that the physical devices can be described by their real-world properties, and (2) the way that these devices appear to the application program. We then consider client–server networks and client–server graphics. We use these ideas to develop event-driven input for our graphics programs. Finally, we develop a paint program that demonstrates the important features of interactive graphics programming.

3.1 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 Project Sketchpad 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.

In the 40 years since Sutherland’s work, there have been many advances in both hardware and software, but the viewpoint and ideas that he introduced still dominate interactive computer graphics. These influences range from how we conceptualize the human–computer interface to how we can employ graphical data structures that allow for efficient implementations.

In this chapter, we take an approach slightly different from that in the rest of the book. Although rendering is the prime concern of most modern APIs, including OpenGL, interactivity is an important component of many 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, windowing and input functions were left out of the API. Although this decision makes renderers portable, it makes more difficult discussions of interaction that do not include specifics of the windowing system. In addition, because any application program must have at least a minimal interface to the windowing environment, we cannot avoid such issues completely if we want to write complete, nontrivial programs. If interaction is omitted from the API, the application programmer is forced to worry about the often arcane details of her particular environment.

We can avoid such potential difficulties by using a simple library, or toolkit, as we did in Chapter 2. The toolkit can provide 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 through the toolkit’s API. We adopt this approach, even though it may not provide all the features of any particular windowing system, and may produce code that neither makes use of the full capabilities of the window system, nor is as efficient as would be the code written in terms of the particular environment.

We use the term windowing system, as we did in Chapter 2, 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 windowing-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 X window. Our use of the GLUT toolkit will enable us to avoid the complexities inherent in the interactions among the windowing system, the window manager, and the graphics system. Just as it did in Chapter 2, GLUT will allow our sample programs to be independent of any particular window system.

We start by describing several interactive devices and the variety of ways that we can interact with them. We then put these devices in the setting of a client–server network. Then, we introduce an API for minimal interaction. Finally, we shall generate sample programs.

### 3.2 Input Devices

We can think about input devices in two distinct ways. The obvious one is to look at them as physical devices, such as a keyboard or a mouse, and to discuss how they work. Certainly, we need to know something about the physical properties of our input devices, so such a discussion is necessary if we are to obtain a full understanding of input. However, from the perspective of an application programmer, we almost never want to use anything about the particular characteristics of a physical device in an application program. Rather,
we prefer to treat input devices as logical devices whose properties are specified in terms of what they do from the perspective of the application program. A logical device is characterized by its high-level interface with the user program, rather than by its physical characteristics. Logical devices are familiar to all writers of high-level programs. For example, data input and output in C are done through functions such as printf, scanf, getchar, and putchar, whose arguments use the standard C data types, and through input (cin) and output (cout) streams in C++. When we output a string using printf, the physical device on which the output appears could be a printer, a terminal, or a disk file. This output could even be the input to another program. The details of the format required by the destination device are of minor concern to the writer of the application program.

In computer graphics, the use of logical devices is slightly more complex, because the forms that input can take are more varied than the strings of bits or characters to which we are usually restricted in nongraphical applications. For example, we can use the mouse—a physical device—either to select a location on the screen of our CRT, or to indicate which item in a menu we wish to select. In the first case, an x, y pair (in some coordinate system) is returned to the user program; in the second, the application program may receive an integer as the identifier of an entry in the menu. The separation of physical from logical devices allows us to use the same physical devices in two markedly different logical ways. It also allows the same program to work, without modification, if the mouse is replaced by another physical device, such as a data tablet or trackball.

3.2.1 Physical Input Devices

From the physical perspective, each device has properties that make it more suitable for certain tasks than for others. We take the view used in most of the workstation literature that there are two primary types of physical devices: pointing devices and keyboard devices. The pointing device allows the user to indicate a position on the screen, 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 codes1 to a program.

The mouse (Figure 3.1) and trackball (Figure 3.2) are similar in use and, often, in construction. A typical mechanical mouse when turned over 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. Optical mice measure distance

---

1. We use the American Standard Code for Information Interchange (ASCII), although nothing we do restricts us to this particular choice, other than that ASCII is the prevailing code used.
traveled by counting lines on a special pad. 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.

We can view the output of the mouse or trackball as two independent values provided by the device. These values can be considered as positions and converted—either within the graphics system or by the user program—to a two-dimensional location in either screen or world coordinates. If it is configured in this manner, we can use the device to position a marker (cursor) automatically on the display; however, we rarely use these devices in this direct manner.

It is not necessary that the output of the mouse or trackball encoders be interpreted as a position. Instead, either the device driver or a user program can interpret the information from the encoder as two independent velocities (see Exercise 3.4). The computer can then integrate these values to obtain a two-dimensional position. Thus, as a mouse moves across a surface, the integrals of the velocities yield \( x, y \) values that can be converted to indicate the position for a cursor on the screen, as shown in Figure 3.3. By integrating the distance traveled by the ball as a velocity, we can use the device as a variable-sensitivity input device. Small deviations from rest cause slow or small changes; large deviations cause rapid large changes. With either device, if the ball does not rotate, then there is no change in the integrals, and a cursor tracking the position of the mouse will not move. In this mode, these devices are \textit{relative-positioning} devices, because changes in the position of the ball (or of the mouse) is not used by the application program.

Relative positioning, as provided by a mouse or trackball, is not always desirable. In particular, these devices are not suitable for an operation such as tracing a diagram. If, while the user is attempting to follow a curve on the screen with a mouse, she lifts and moves the mouse, the absolute position on the curve being traced is lost. \textbf{Data tablets} provide absolute positioning. 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 many portable computers. These touchpads can be configured as either relative- or absolute-positioning devices.

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. Because each redisplay of the frame buffer starts at a precise time, we can use the time at which this signal occurs to determine a position on the CRT screen (see Exercise 3.19). 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.

One other device, the joystick (Figure 3.6), is 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 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 mechanical feel, which is not possible with the other devices, makes the joystick well suited for applications such as flight simulators and games.

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. Thus, the device measures six independent values and 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-based structured-lighting systems and laser-ranging systems, measure three-dimensional positions. 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.

We shall not use three-dimensional input in our code, although there is nothing in the API that restricts the input to two dimensions.
3.2.2 Logical Devices

We can now return to looking at input from inside the application program—that is, from the logical point of view. Two major characteristics describe the logical behavior of an input device: (1) the measurements that the device returns to the user program, and (2) the time when the device returns those measurements.

Some APIs, such as PHIGS and GKS, consider 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 show the variety of input forms that a developer of graphical applications may want, and they also show how OpenGL can provide similar functionality.

1. **String** A string device is a logical device that provides ASCII strings to the user program. Usually, this logical device is implemented by means of a physical keyboard. In this case, the terminology is consistent with the terminology used in most window systems and OpenGL, which do not distinguish between the logical string device and the 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 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* 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 choice device, allowing us to select one of \( n \) alternatives.

5. **Dial** Dials (or valuators) provide analog input to the user program. Here again, widgets within various toolkits usually provide this facility through graphical devices, such as slidebars.

6. **Stroke** A stroke device returns an array of locations. Although we can think of a stroke as similar to multiple uses of a locator, it is often implemented such that an action, such as 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.
3.2.3 Measure and Trigger

The manner by which physical and logical 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 contains a string, and the trigger can be the “return” or “enter” key. For a locator, the measure includes the position, and the associated trigger can be a button on the pointing device.

In addition to its obvious parts, the measure can include other information, such as status. For example, a pick device will return in its measure the identifier of the object to which the user is pointing. If the physical device is a mouse, the trigger is a mouse button. When we develop an application program, we have to account for the user triggering the device while she is not pointing to an object. If the measure consists of only an object identifier, we face problems in constructing code that takes care of this situation correctly. We can resolve the problem more easily if part of the measure is a status variable that indicates that the user was not pointing to an object, or that the cursor was outside the window, when the trigger occurred.

3.2.4 Input Modes

In addition to the multiple types of logical input devices, we 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. Normally, the initialization of the input device starts a measure process. The initialization may require an explicit function call in some APIs, or it may occur automatically. In either case, once the 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, 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 nongraphical applications, such as a typical C program that requires character input. When we use a function such as `scanf`, the program halts when it encounters this 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 depressed. 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 as 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.

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. We usually interface with the devices through functions such as

```c
request_locator(device_id, &measure);
sample_locator(device_id, &measure);
```

which have as their parameters the identifier of the particular device and a location in the user program to place the measure of the device. Consequently, we ignore any other information that becomes available from any input device other than the one specified in the function call. 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 might have multiple input devices—such as a joystick, dials, buttons, and switches—most of which can be used at any time by the pilot. 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, event mode, 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 event-mode 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, with 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 user 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 event’s type and then decide what to do. This method is used in the APIs for GKS and PHIGS.

Another approach is to associate a function called a **callback** with a specific type of event. We take this approach, because it is the one currently used with the major windowing systems and because it has been proved to work well in client–server environments.

### 3.3 Clients and Servers

So far, our approach to input has been isolated from all other activities that might be happening in our computing environment. We have looked at our graphics system as a monolithic box that has limited connections to the outside world, other 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 supercomputers, accessible from user programs; file servers that allow users to share files and programs, regardless of into what machine they are logged; and terminal servers that handle dial-in access. Users and user programs that make use of these services are clients or client programs.

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 potentially are available to clients anywhere on the network.

Our OpenGL application programs are clients that use the graphics server. Within an isolated system, this distinction may not be apparent as we write, compile, and run the software on a single machine. However, we also can run the same application program using other graphics servers on the network.

### 3.4 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. As we saw in Chapter 1, the original architecture of a graphics system was based on a general-purpose computer (or host) connected, through digital-to-analog converters, to a CRT (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.

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 CRT. 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 noninteractive 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.

Today, the display processor of old has become a graphics server, and the user 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 (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 display, and no memory of it is retained in the system.  

To redisplay the primitive after a clearing of the screen, or in a new position after an interaction, the program must redefine the

---

2. This rate depends on the phosphors in the CRT, but we usually assume it to be in the range of 50 to 85 Hz, or one-half that rate if the display is interlaced.

3. The image of the primitive is retained in the frame buffer, but objects and images are not the same.
primitive and then must resend the primitive to the display. 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, 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 also allows the client to take advantage of any special-purpose graphics hardware that might be available in the graphics server. Thus, in many situations, the optimum configuration consists of a good numerical-processing computer that executes the client program and a special-purpose graphics computer for the server—an old idea used with great efficiency in modern systems.

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 often is offset by the efficiency of the execution of the display list, it might not be if the data are changing.

### 3.4.1 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 what contents of a display list are permissible should be flexible enough to allow considerable freedom to the user. OpenGL4 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’ use.

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

---

4. PHIGS contains *structures* and GKS contains *segments*, both of which provide many of the characteristics of OpenGL display lists.
3.4 Display Lists

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. The code is similar to code from Chapter 2, but this time places the information in a display list:

```c
#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, 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

```c
glCallList(BOX);
```

Just as it does with other OpenGL functions, the present 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 even will no longer appear, as the following code fragment demonstrates:

```c
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);
}
```

Each time that `glCallList` is executed, the box is redrawn, albeit with a different clipping rectangle.

In succeeding chapters, we introduce various transformation matrices that will enable us to use display lists for modeling. Note that, because we can change state from within a display list, we have to be careful to avoid allowing these changes to have undesirable—and often unexpected—effects later. For example, our box display list changes the drawing color. Each time that the display list is executed, the drawing color is set to red; unless the color is set to some other value,
primitives defined subsequently in the program also will be colored red. The easiest safeguard is to use the matrix and attribute stacks provided by OpenGL. 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

```cpp
    glPushAttrib(GL_ALL_ATTRIB_BITS);
    glPushMatrix();
```

at the beginning of a display list, and

```cpp
    glPopAttrib();
    glPopMatrix();
```

at the end. We use matrix and attributes stacks extensively in Chapter 9 to build and display hierarchical models.

A few additional functions are available that make it easier to work with display lists. Often, we want to work with multiple display lists, as we demonstrate with the next example. We can create multiple lists with consecutive identifiers more easily if we use the function `glGenLists(number)`, which returns the first integer (or base) of `number` consecutive integers that are unused labels. The function `glCallLists` allows us to execute multiple display lists with a single function call. Text generation is a good example of how we can make excellent use of the options available through display lists. Section 3.4.2 contains OpenGL details; you may want to skip it the first time that you read this chapter. It illustrates the flexibility that the API provides to the application programmer for dealing with troublesome problems that arise in working with text. We encounter many of these issues again in Chapter 9 when we discuss graphics and the World Wide Web.

### 3.4.2 Text and Display Lists

In Chapter 2, 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 a 12 × 10 pattern of bits. It takes 15 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 15 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 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 15 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.

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

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

Within each case, we have to be careful about the spacing; each character in the string must be displayed to the right of the previous character. We can use the `glTranslatef` function to get the desired spacing. 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

```c
case 'O':
    glTranslatef(0.5, 0.5, 0.0); /* move to center */
glBegin(GL_QUAD_STRIP);
for (i=0; i<=12; i++) /* 12 vertices */
```
This code approximates the circle with 12 quadrilaterals. Each will be filled according to the present 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. 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. 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:

```c
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();
}
```

---

5. Each character may have a different size and thus be in a box of unique dimensions, or, if we are defining a fixed-width monotype font (such as the one used to set code in this book), all characters will have boxes of the same size.
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
with

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.

3.4.3 Fonts in GLUT

In general, we prefer to use an existing font, rather than to define our own. GLUT
provides a few raster and stroke fonts.6 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 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.

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

glutBitmapCharacter(GLUT_BITMAP_8_BY_13, int character)

6. We can also access fonts that are provided by the windowing system at the expense of code portability.
Positioning of bitmap characters is considerably simpler than is that of stroke characters, 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 function `glRasterPos*()`. The user program typically moves the raster position one character to the right each time that `glutBitmapCharacter` is invoked. This change does not affect subsequent rendering of geometric primitives. If characters have different widths, we can use the function `glutBitmapWidth(font, char)` to return the width of a particular character. Thus, when we define a sequence of characters, we often see code such as

```c
glRasterPos2i(rx, ry);
glutBitmapCharacter(GLUT_BITMAP_8_BY_13, k);
rx+=glutBitmapWidth(GLUT_BITMAP_8_BY_13, k);
```

We use bitmap characters in our example of a painting program later in this chapter, using display lists.

The case for always using display lists is strong, and we return to them when we discuss hierarchical modeling in Chapter 9. At this point, however, we are more interested in clarity and brevity than in efficiency. Hence, we do not use display lists in many of our examples. Most user code can be encapsulated between a `glNewList` and `glEndList`. Thus, you should be able to convert most code to use display lists with little effort.

### 3.5 Programming Event-Driven Input

In this section, we develop event-driven input through a number of simple examples that use the callback mechanism that we introduced in Section 3.2. We examine various events that are recognized by the window system and, for those of interest to our application, we write callback functions that govern how the application program responds to these events.

#### 3.5.1 Using the Pointing Device

We start by altering the `main` function in the gasket program from Chapter 2. In the original version, we used functions in the GLUT library to put a window on the screen, and then entered the event loop by executing the function `glutMainLoop`. In that chapter, we entered the loop but did nothing. We could not even terminate the program, except through an external mechanism. Our first example will remedy this omission by using the pointing device to terminate a program. We accomplish this task by having the program execute a standard termination function called `exit` when a particular mouse button is depressed.
We discuss only those events recognized by GLUT. A window system such as the X Window system recognizes many more events. However, the GLUT library will recognize a set of events that is common to most window systems, and is sufficient for developing basic interactive graphics programs that can be used with multiple window systems. Two types of events are associated with the pointing device, which is conventionally assumed to be a mouse. A **move event** is generated when the mouse is moved with one of the buttons depressed. If the mouse is moved without a button being held down, this event is classified as 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 depressed or released. A button being held down does not generate an event until the button is released. The information returned—the measure—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 screen coordinates. We specify the mouse callback function, usually in the main function, by means of the GLUT function

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

If any other mouse event—such as a depression of one of the other buttons—occurs, no response action will occur, because no callback corresponding to these events has been defined, or **registered**, with the window system.

Our next example illustrates the benefits of the program structure that we introduced in the previous chapter. We write a program to draw a small box at each location on the screen where the mouse cursor is located at the time that the left button is pushed. A push of the middle button terminates the program.

---

7. Certain systems count the pushing and the releasing of a button as only a single event.
First, we look at the main program, which is much the same as our previous examples.\footnote{We use naming conventions for callbacks similar to those in the \textit{OpenGL Programmer's Guide} \cite{Ope01a}.}

```c
int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutCreateWindow("square");
    myinit();
    glutReshapeFunc(myReshape);
    glutMouseFunc(mouse);
    glutDisplayFunc(display);
    glutMainLoop();
}
```

The \texttt{reshape} event is generated whenever the window is resized, such as by a user interaction; we discuss it next. We do not use the required display callback in this example, because the only time that primitives will be generated is when a mouse event occurs. Because GLUT requires that every program have a display callback, we must include this callback, although it can have an empty body:

```c
void display(){ }
```

The mouse callbacks are again in the function \texttt{mouse}.

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

Because only the primitives are generated in \texttt{drawSquare}, the desired attributes must have been set elsewhere, such as in our initialization function \texttt{myinit}.

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 \texttt{drawSquare}. If we want to change the size of the squares that we draw, we may find it beneficial also to make the square-size parameter global. Our initialization routine selects a clipping window that is the same size as the window created in \texttt{main}, and selects the 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, since we are merely setting them to
the default values. However, it is illustrative to compare this code with what we do in the reshape callback in Section 3.5.2.

/* globals */
GLsizei wh = 500, ww = 500; /* initial window size */
GLfloat size = 3.0; /* one-half of side length of square */

void myinit(void)
{
    /* set viewing conditions */
    glViewport(0,0,ww,wh);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, (GLdouble) ww, 0.0, (GLdouble) wh);
    glMatrixMode(GL_MODELVIEW);
    /* set clear color to black, and clear window */
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();
}

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 \( wh \)). We pick a random color using the standard random-number generator \( \text{rand()} \).

void drawSquare(int x, int y)
{
    y = wh - y;
    glColor3ub((char) rand()%256, (char) rand()%256, (char) rand()%256);
    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();
}

After we insert the necessary include statements, we have a program that works, as long as the window size remains unchanged.
3.5.2 Window Events

Most window systems allow a user to resize the window, 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. If such an event occurs, the user program can decide what to do.\(^9\) If the window size changes, we have to consider three questions:

1. Do we redraw all the objects that were in the window before it was resized?
2. What do we do if the aspect ratio of the new window is different from that of the old window?
3. Do we change the sizes or attributes of new primitives if the size of the new window is different from that of the old?

There is no single answer to any of these questions. If we are displaying the image of a real-world scene, our reshape function probably should make sure that no shape distortions occur. But this choice may mean that part of the resized window is unused, or that part of the scene cannot be displayed in the window. If we want to redraw the objects that were in the window before it was resized, we need a mechanism for storing and recalling them. Often, we do this recall by encapsulating all drawing in a single function, such as the display function used in Chapter 2, which was registered as the display callback function. In the present example, however, that is probably not the best choice, because we decide what we draw interactively.

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 that 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 callback

```c
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();
}
```

---

\(^9\) Unlike most other callbacks, there is a default reshape callback, although it might not do what the user desires.
3.5 Programming Event-Driven Input

/* adjust viewport and clear */

glViewport(0,0,w,h);
glClearColor (0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT);
glFlush();
}

The complete square-drawing program is given in Appendix A.

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 (brought to the front). There are callbacks for these events, and we can write simple functions similar to myReshape for them, or we can rely on the default behavior 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 function

glutMotionFunc(drawSquare);

Each time the system senses the motion, a new square is drawn—an action that allows us to draw pictures using a brush with a square tip.

3.5.3 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 depressed. Although in many window systems the release of a key generates a second event, GLUT does not have a callback for this event. The ASCII code for the key pressed and the location of the mouse are returned. All the keyboard callbacks are registered in a single callback function, such as

glutKeyboardFunc(keyboard);

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

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

GLUT includes functions that allow the user to define actions using the meta keys, such as the control and alt keys. These special keys can be important when...
we are using one- or two-button mice as we can then define the same functionality as having left, right, and middle buttons as we have assumed in this chapter.

### 3.5.4 The Display and Idle Callbacks

Of the remaining callbacks, two merit special attention. We have already seen the display callback, which we used in Chapter 2. This function is specified in GLUT by the function

```c
glutDisplayFunc(display);
```

It is invoked when GLUT determines that the window should be redisplayed. One such situation occurs when the window is opened initially. 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 noninteractive output. GLUT requires every program to have a display callback.

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 reexecution of the display function. Rather than call it directly, we use the GLUT function

```c
glutPostRedisplay()
```

Using this function, rather than invoking the display callback directly, avoids extra or unnecessary drawings of the screen 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 at most once each time the program goes through the main loop.

The **idle callback** is invoked when there are no other events. Its default is the null function. A typical use of the idle callback is to continue to generate graphical primitives through a display function while nothing else is happening (see Exercise 3.2). We illustrate both the display and the idle callbacks in the paint programs that we develop in Sections 3.8 and 3.9.

We can change most callback functions during program execution by specifying a new callback function. We can also disable a callback by setting its callback function to `NULL`. 
3.5.5 Windows Management

GLUT also supports both multiple windows and subwindows of a given window. We can open a second top-level window (with the label “second window”) by

```c
id = glutCreateWindow("second window");
```

The returned integer value allows us to select this window as the current window into which objects will be rendered by

```c
glutSetWindow(id);
```

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.

3.6 Menus

We could use our graphics primitives and our mouse callbacks to construct various graphical input devices. For example, we could construct a slidebar as shown in Figure 3.16, using filled rectangles for the device, text for any labels, and the mouse to get the position. Much of the code would be tedious to develop, however, especially if we tried to create visually appealing and effective graphical devices (widgets). Most window systems provide a toolkit that contains a set of widgets, but, because our philosophy is not to restrict our discussion to any particular window system, we shall not discuss the specifics of such widgets sets. Fortunately, 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 entries in the menu. We must link the menu to a particular mouse button. Finally, we must define a callback function corresponding to each menu entry. 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

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

```c
void demo_menu(int id)
{
    if(id == 1) exit( );
    else if (id == 2) size = 2 * size;
    else if (size > 1) size = size/2;
    glutPostRedisplay( );
}
```

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

GLUT also supports hierarchical menus, as shown in Figure 3.17. 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:

```c
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);
```

Writing the callback functions, `size_menu` and `top_menu`, should be a simple exercise for you (Exercise 3.5).

### 3.7 Picking

Picking is the logical input operation that allows the user to identify an object on the display. Although the action of picking uses the pointing device, the
information that the user wants returned to the application program is not a position. A pick device is considerably more difficult to implement on a modern system than is a locator.

Such was not always the case. Old display processors could accomplish picking easily by means of a lightpen. Each redisplay of the screen would start at a precise time. The lightpen would generate an interrupt when the redisplay passed its sensor. By comparing the time of the interrupt with the time that the redisplay began, the processor could identify an exact place in the display list, and subsequently could determine which object was being displayed.

One reason for the difficulty of picking in modern systems is the forward nature of their rendering pipelines. Primitives are defined in an application program and move forward through a sequence of transformations and clippers until they are rasterized into the frame buffer. Although much of this process is reversible in a mathematical sense, the hardware is not reversible. Hence, converting from a location on the display to the corresponding primitive is not a direct calculation. There are also potential uniqueness problems; see Exercises 3.11 and 3.12.

There are at least three ways to deal with this difficulty. One process, known as selection, involves adjusting the clipping region and viewport such that we can keep track of which primitives in a small clipping region are rendered into a region near the cursor. These primitives go into a hit list that can be examined later by the user program. OpenGL supports this approach, and we shall discuss it after we discuss two simpler, but less general, strategies.

A simple approach is to use bounding rectangles, or extents, for objects of interest. The extent of an object is the smallest rectangle, aligned with the coordinates axes, that contains the object. It is relatively easy to determine which rectangles in world coordinates correspond to a particular point in screen coordinates. If the application program maintains a simple data structure to relate objects and bounding rectangles, approximate picking can be done within the application program. We demonstrate a simple example of this approach in Section 3.8.

Another simple approach involves using the back buffer and an extra rendering. Recall that the back buffer is not displayed and thus we can use it for purposes other than forming the image we will display when we swap the front and back buffers. Suppose that we render our objects into the back buffer, each in a distinct color. The application programmer is free to determine an object’s contents by simply changing colors wherever a new object definition appears in the program.

We can perform picking in four steps that are initiated by a pick function in the application. First, we draw the objects into the back buffer with the pick colors. Second, we get the position of the mouse using the mouse callback. Third, we use the function glReadPixels() to find the color at the position in the frame buffer corresponding to the mouse position. Finally, we search a table of colors to find which object corresponds to the color read. We must follow this process by a normal rendering into the back buffer. There are some subtleties to the use of glReadPixels(), which we shall consider in Chapter 7.
3.7.1 Picking and Selection Mode

The difficult problem in implementing picking within the OpenGL pipeline is that we cannot go backward directly from the position of the mouse to primitives that were rendered close to that point on the screen. OpenGL provides a somewhat complex process using a rendering mode called selection mode to do picking at the cost of an extra rendering each time we do a pick. The basic idea of selection mode is that the objects in a scene can be rendered but not necessarily to the color buffer. As we render objects, OpenGL can keep track of which objects render to any chosen area by determining whether they are in a specified clipping volume, which does not have to be the same as the clipping volume used to render into the color buffer. There are a number of steps and functions that are required to do picking. We shall examine each step and then put them together in a simple program. The function `glRenderMode` lets us select one of three modes: normal rendering to the color buffer (GL_RENDER), selection mode (GL_SELECT), or feedback mode (GL_FEEDBACK). Feedback mode can be used to obtain a list of the primitives that were rendered. We will not discuss this mode. The return value from `glRenderMode` can be used to determine the number of hits in selection mode or primitives in feedback mode.

When we enter selection mode and render a scene, each primitive that renders within the clipping volume generates a message called a hit that is stored in a buffer called the name stack. We use the function `glSelectBuffer` to identify an array for the selection data. There are four functions for initializing the name stack, for pushing and popping information on it, and for manipulating the top entry on the stack. The information that we produce is called the hit list and can be examined after the rendering to obtain the information needed for picking. The function

```c
void glSelectBuffer(GLsizei n, GLuint *buff)
```

specifies the array `buff` of size `n` in which to place selection data. The function

```c
void glInitNames()
```

initializes the name stack. The function

```c
void glPushName(GLuint name)
```

pushes `name` on the name stack. The function

```c
void glPopName()
```

pops the top name from the name stack. Finally, the function

```c
void glLoadName(GLuint name)
```
replaces the top of the name stack with name. In general, each object that we wish to identify is a set of primitives to which we assign the same integer name. Before we render the object, we load its name on the name stack. We cannot load a name onto an empty stack, so we usually enter an unused name onto the stack when we initialize it, for example, by

```c
glInitNames();
glPushName(0);
```

We usually use the mouse callback to enter selection mode and leave selection mode before the end of the mouse callback. When we return to render mode, `glRenderMode` returns the number of hits that have been processed. We then examine the hit list. We also change the clipping volume within the mouse callback so that we obtain hits in the desired region, usually an area that is close to the location of the mouse. We can set the clipping volume in two ways. We could simply set a the view volume through `gluOrtho2D` (or use other viewing functions). We would probably first want to save the present clipping volume with a `glPushMatrix`. Then any primitive that fell within this new clipping volume would generate a hit regardless of where the mouse is located. This option works for selection, but when we pick we want only those objects that render near the cursor.

Suppose that we want all the objects that render into a small user-defined rectangle centered at the cursor. The size of the rectangle is a measure of how sensitive we want our picking to be. This rectangle is a small part of the viewport. Given the viewport, the location of the cursor, the size of the rectangle, and the clipping window, we can find a new clipping window such that all the objects in the new clipping window render into the full viewport. Mathematically, this is an exercise in proportions and involves the inverse of the projection matrix. We can let OpenGL do this calculation for us through the GLU function `gluPickMatrix`, which is applied before `glOrtho2D` when we are in selection mode. The function call has the form

```c
gluPickMatrix( x, y, w, h, *vp);
```

and creates a projection matrix for picking that restricts drawing to a \( w \times h \) area centered at \((x, y)\) in window coordinates within the viewport \( \text{vp} \). Figure 3.18 illustrates the process for a two-dimensional application. In Figure 3.18(a) we see the normal window and the image on the display. We also see the cursor with a small box around it indicating the area in which, if a primitive is rendered, we will count it as a hit. Figure 3.18(b) shows the window and display after the window has been changed by `gluPickMatrix`. Note that the world window has been changed so that only those objects that were in the pick rectangle are in the new window and these objects now occupy the whole viewport. It is for this reason that we do not want this image to appear on the screen.

Assuming that we have set up the viewing conditions for normal rendering during initialization or in the reshape callback, the mouse and display callbacks are of the form
Chapter 3  Input and Interaction

Figure 3.18  (a) Normal window and display. (b) Window and display after applying pick matrix.

```c
void mouse(int button, int state, int x, int y
{
    GLuint nameBuffer[SIZE]; /* define SIZE elsewhere */
    GLint hits;
    GLint viewport[4];
    If( button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    {
        /* initialize the name stack */
        glInitNames();
        glPushName(0);
        glSelectBuffer(SIZE, nameBuffer);
        /* set up viewing for selection mode */
        glGetIntegerv(GL_VIEWPORT, viewport);
        glMatrixMode(GL_PROJECTION);
        /* save original viewing matrix */
        glPushMatrix();
        glLoadIdentity();
        /* N x N pick area around cursor */
    }
```
/* must invert mouse y to get in world coords */

    gluPickMatrix( (GLdouble) x, (GLdouble)
    (viewport[3] - y), N, N, viewport);

/* same clipping window as in reshape callback */

    gluOrtho2D (xmin, xmax, ymin, ymax);

draw_objects(GL_SELECT);
    glMatrixMode(GL_PROJECTION);
/* restore viewing matrix */
    glPopMatrix();
    glFlush();
/* return to normal render mode */
    hits = glRenderMode(GL_RENDER);
/* process hits from selection mode rendering */
    processHits(hits, nameBuff);
/* normal render */
    glutPostRedisplay();
}

void display( )
{
    glClear( GL_COLOR_BUFFER_BIT);
    draw_objects(GL_RENDER);
    glFlush();
}

Note that we have to call the function that draws the objects directly rather
than use glutPostRedisplay twice in the mouse callback because GLUT will only
do one execution of the display callback each time through the event loop. Here
is a simple function that draws two partially overlapping rectangles:

void drawObjects(GLenum mode)
{
    if(mode == GL_SELECT) glLoadName(1);
    glColor3f(1.0, 0.0, 0.0);
    glRectf(-0.5, -0.5, 1.0, 1.0);
...
Note that we need to change only the top element on the name stack. If we had a hierarchical object, as we shall discuss in Chapter 9, we could use `glPushMatrix` so that we could have multiple names on the stack for a given hit. For an object with multiple parts, all the parts that were close to the cursor would have their names placed in the same stack. The final piece we need to write is the function that examines the name stack. We will have it print out how many were placed in the stack for each left mouse click and the names of any objects that were picked. The hit buffer contains one record for each hit. Thus, every object that is rendered near the mouse cursor will generate a record. If there are no names on the hit list because no primitives were rendered near the mouse cursor, then there is a zero for the hit record. Otherwise, we find three types of information, all stored as integers. First there is the number of names on the name stack when there was a hit. For our example, this number can only be 1. It is followed by two integers which give scaled minimum and maximum depths for the hit primitive. As we are working in two dimensions, these values will not provide us useful information. For three-dimensional applications, we can use these values to determine the front object that was picked. These three integers are followed by entries in the name stack. For our example, we will find the identifier of either the red or the blue rectangle here (a 1 or a 2). The following function will print out this information.

```c
void processHits (GLint hits, GLuint buffer[])
{
    unsigned int i, j;
    GLuint names, *ptr;

    printf ("hits = %d\n", hits);
    ptr = (GLuint *) buffer;

    /* Loop over number of hits */
    for (i = 0; i < hits; i++)
    {
        names = *ptr;

        /*skip over number of names and depths */
        ptr += 3;

        /* check each name in record */
        for (j = 0; j < names; j++)
        {
            // Code continues here
        }
    }
}
```
Picking presents a number of other difficulties for the implementor. One is that, if an object is defined hierarchically, like those we discuss in Chapter 9, then it is a member of a set of objects. When the object is indicated by the pointing device, a list of all objects in the set of which it is a member should be returned to the user program.

Graphics APIs usually do not specify how close to an object we must point for the object to be identified. One reason for this omission is to allow multiple ways of picking. For example, if we use the bounding-rectangle strategy, we might not point to any point contained in the object. Another reason for this lack of preciseness is that humans are not accurate at positioning devices. Although the display at which we point might have a high resolution, we may find it difficult to indicate reliably with our pointing device the location on the screen corresponding to a given pixel.

### 3.8 A Simple Paint Program

We illustrate the use of callbacks, display lists, and interactive program design by developing a simple paint program. Paint programs have most of the features of sophisticated CAD programs. Any paint program should demonstrate most of the following features:

- It should have the ability to work with geometric objects, such as line segments and polygons. Given that geometric objects are defined through vertices, the program should allow us to enter vertices interactively.
- It should have the ability to manipulate pixels, and thus to draw directly into the frame buffer.
- It should provide control of attributes such as color, line type, and fill patterns.
- It should include menus for controlling the application.
- It should behave correctly when the window is moved or resized.

Our sample program demonstrates these principles, and incorporates many of the features that we introduced in the small examples that we have developed.

Figure 3.19 shows the initial display that a user sees. The five boxes are buttons that select from the five drawing modes: line segment, rectangle, triangle,
pixel, and text. The left mouse button selects the mode. If line-segment mode is selected, the next two clicks of the left mouse button while the cursor is outside the menu area give the locations of the endpoints. After the second endpoint is located, the line segment is drawn in the present color. Rectangle and triangle modes are similar: In rectangle mode, we select the diagonal corners; in triangle mode, we locate the three vertices. A new mode can be selected at any time. In pixel mode, successive mouse clicks give positions for randomly colored rectangles that can have a side length of two or more pixels. In text mode, the user types in a string of characters that appear on the display starting where the mouse was clicked most recently.
The menus (Figure 3.20) are controlled by the middle and right mouse buttons. The right button allows us either to clear the screen or to terminate the program. The middle button allows us to change the drawing color, to select between fill and no fill for the triangles and rectangles, and to change the size of the pixel rectangles.

The most difficult part of the design process is deciding what functionality to place in each of the functions. The decisions reflected in our code are consistent with our previous examples. Many of the functions in our program are the same as those in our previous examples. We employ the following functions:

```c
void mouse(int btn, int state, int x, int y); /* mouse callback */
void key(unsigned char c, int x, int y); /* keyboard callback */
void display(void); /* display callback */
void drawSquare(int x, int y); /* random-color square function*/
void myReshape(GLsizei, GLsizei); /* reshape callback */
void myinit(void); /* initialization function */
void screen_box(int x, int y, int s); /* box-drawing function */
void right_menu(int id); /* menu callbacks */
void middle_menu(int id);
void color_menu(int id);
void pixel_menu(int id);
void fill_menu(int id);
int pick(int x, int y); /* mode-selection function */
```

We have used most of these functions in our previous examples; thus, the main function is similar to the main function in our other examples:

```c
int main(int argc, char **argv)
{
    int c_menu, p_menu, f_menu;
    glutInit(&argc,argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutCreateWindow("Paint");
    glutDisplayFunc(display);
    c_menu = glutCreateMenu(color_menu);
    glutAddMenuEntry("Red",1);
    glutAddMenuEntry("Green",2);
    glutAddMenuEntry("Blue",3);
    glutAddMenuEntry("Cyan",4);
    glutAddMenuEntry("Magenta",5);
    glutAddMenuEntry("Yellow",6);
    glutAddMenuEntry("White",7);
    glutAddMenuEntry("Black",8);
    ```
The function `myinit` clears the window, sets up global variables, and defines and compiles display lists for 128 characters.

```c
void myinit(void)
{
    /* set up a font in display list */
    int i;
    base = glGenLists(128);
    for(i=0;i<128;i++)
    {
        glNewList(base+i, GL_COMPILE);
        glutBitmapCharacter(GLUT_BITMAP_9_BY_15, i);
        glEndList();
    }

    glListBase(base);

    glViewport(0,0,ww,wh);

    /* Pick 2D clipping window to match size of X window.
    This choice avoids the need to scale object coordinates
    each time that the window is resized */
```
The function `display` clears the window, sets the background to white, and then draws the buttons. The buttons are drawn as rectangles with sides that are 10% of the window height and width; they thus change size when the window is resized. Otherwise, the code here draws only simple primitives.

The function `myReshape` is the same reshape callback that we introduced in Section 3.5.

The function `pick` detects which object on the screen has been selected by the mouse and thus works in conjunction with the mouse callback function `mouse`. OpenGL supports a method for doing general object selection that is complex to implement. When we are interested in only rectangular areas of the screen, such as the drawing area and the buttons, it is simpler to implement our own pick method within the application program. Our procedure works as follows. Once a mouse event is detected, `mouse` uses `pick` to identify on which part of the screen the mouse is located. Because the buttons are rectangular boxes, we can detect directly whether the mouse is inside a button or in the drawing area; we return this information to `mouse`, which then does most of the work of drawing. Once an area has been identified by means of `pick`, the code chooses a proper mode, and determines whether it needs to output a primitive or to store the mouse location. For example, if the user indicated a triangle by first clicking on the appropriate button, the program must store two positions; then, when it receives the third position, it can draw the triangle. The appropriate code in `mouse` is

```c
switch(count) /* switch on number of vertices */
{
    case(0): /* store first vertex */
        count++;  
        xp[0] = x; 
        yp[0] = y; 
        break;
    case(1): /* store second vertex */
        count++;  
        xp[1] = x; 
        yp[1] = y; 
        break;
    // other cases...
}
```
case(2): /* third vertex: draw triangle */
    if(fill) glBegin(GL_POLYGON);
    else glBegin(GL_LINE_LOOP);
    glVertex2i(xp[0],wh-yp[0]);
    glVertex2i(xp[1],wh-yp[1]);
    glVertex2i(x,wh-y);
    glEnd();
    draw_mode=0; /* reset mode */
    count=0;

Note that the first mouse click selects triangle mode; subsequent clicks count and store vertices. However, if the mouse is clicked on another button before the third vertex is selected, a new object is selected and drawn.

The function key renders characters entered from the keyboard to characters on the display.

void key(unsigned char k, int xx, int yy)
{
    if(draw_mode!=TEXT) return; /* draw characters until mode changes */
    glRasterPos2i(rx,ry);
    glCallList(k); /* Display list for character k */
    rx+=glutBitmapWidth(GLUT_BITMAP_9_BY_15,k);
}

A typical display from the middle of a drawing is shown in Figure 3.21. Although our paint program is simple and is limited in the facilities that it provides the user, we should be able to add more functionality easily. Straightforward
exercises are adding the ability to switch from raster to stroke text, allowing multiple fonts, adding line attributes (width and type), allowing selection of fill methods (solid or patterns), and adding more objects (arbitrary polygons, circle approximations, and curves). However, making the paint program function as smoothly as commercial products do is much more difficult (although possible), for reasons that we discuss in Section 3.9.

The complete program is given in Appendix A.

3.9 Animating Interactive Programs

So far, our programs have been static; once a primitive was placed on the display, its image did not change until the screen was cleared. Suppose that we want to create a picture in which one or more objects are changing or moving, and thus their images must change. For example, we might want to have an animated character walk across the display, or we might want to move the viewer over time. We can gain insight into how we can handle these cases through a simple example in which we have a single rotating object.

3.9.1 The Rotating Square

Consider a two-dimensional point where

\[ x = \cos \theta, \]
\[ y = \sin \theta. \]

This point lies on a unit circle regardless of the value of \( \theta \). The three points \((-\sin \theta, \cos \theta), (-\cos \theta, -\sin \theta), \) and \((\sin \theta, -\cos \theta)\) also lie on the unit circle, and the four points are equidistant apart along the circumference of the circle, as shown in Figure 3.22. Thus, if we connect the points to form a polygon, we will have a square centered at the origin whose sides are of length \( \sqrt{2} \).

Assuming that the value of \( \theta \) is a global variable, we can display this cube with the display function

```c
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POLYGON);
    thetar = theta/((2*3.14159)/360.0); /* convert degrees to radians */
    glVertex2f(cos(thetar), sin(thetar));
    glVertex2f(-sin(thetar), cos(thetar));
    glVertex2f(-cos(thetar), -sin(thetar));
    glVertex2f(sin(thetar), -cos(thetar));
    glEnd();
}
```
Our display function will work for any value of $\theta$, but suppose that we change $\theta$ as the program is running, thus rotating the square about the origin. We can display the rotated cube by simply reexecuting the display function through a call to \texttt{glutPostRedisplay}.

Now suppose that we want to increase $\theta$ by a fixed amount whenever nothing else is happening. We can use the idle callback for this operation. Thus, in our main program, we specify the idle callback

\begin{verbatim}
grutIdleFunc(idle);
\end{verbatim}

and define the function as

\begin{verbatim}
void idle()
{
    theta+=2 /* or some other amount */
    if(theta >= 360.0 theta-=360.0;
    glutPostRedisplay();
}
\end{verbatim}

This program will work, but we want to make one further change: We would like to be able to turn on and turn off the rotation feature. We can accomplish this change by using the mouse function to change the idle function. We identify the mouse callback as

\begin{verbatim}
grutMouseFunc(mouse)
\end{verbatim}

Then, this function can be written as

\begin{verbatim}
void mouse(int button, int state, int x, int y)
{
    if(button==GLUT_LEFT_BUTTON&state==GLUT_DOWN)
        glutIdleFunc(idle);
    if(button==GLUT_MIDDLE_BUTTON&state==GLUT_DOWN)
        glutIdleFunc(NULL);
}
\end{verbatim}
Thus, the left mouse button starts the cube rotating, and the middle mouse button stops the rotation. We call the full program single_double.c; it is given in Appendix A. If you run the program, the display probably will not look like a rotating cube. Rather, you will see parts of cubes changing over time. What went wrong? We consider this question next.

3.9.2 Double Buffering

When we redisplay our CRT, we want to do so at a rate sufficiently high that we cannot notice the clearing and redrawing of the screen. For most of us, a CRT display must be refreshed at a rate between 50 and 85 times per second, or we will notice the display flickering. In a graphics system, this requirement means that the contents of the frame buffer must be redrawn at this refresh rate. As long as the contents of the frame buffer are unchanged and we refresh at the 50- to 85-Hz rate, we should not notice the refresh taking place. If we change the contents of the frame buffer during a refresh, we may see undesirable artifacts of how we generate the display.

One manifestation of this problem occurs if the display being drawn is complex and cannot be drawn in a single refresh cycle. In this case, we see different parts of objects on successive refreshes. If an object is moving, its image may be distorted on the display. Another example occurs with the repetitive clearing and redrawing of an area of the screen, as occurs in our rotating square program. Even though the square is a simple object and is rendered in a refresh cycle, in our program there is no coupling between when new cubes are drawn into the frame buffer and when the frame buffer is redisplayed by the hardware. Thus, depending on exactly when the frame buffer is displayed, only part of the square may be in this buffer.

Double buffering can provide a solution to these problems. Suppose that we have two color buffers at our disposal, conventionally called the front and back buffers. The front buffer always is the one displayed, whereas the back buffer is the one into which we draw. We can swap the front and back buffers at will from the application program. When we swap buffers, a display callback is invoked. Hence, if we put our rendering into the display callback, the effect will be to update the back buffer automatically. We can set up double buffering by using the option GLUT_DOUBLE, instead of GLUT_SINGLE in glutInitDisplayMode.

The buffer-swap function using GLUT is

```c
glutSwapBuffers();
```

---

10. If the display is interlaced, the odd and even lines in the frame buffer are displayed alternately, giving a true refresh rate of one half of the specified rate. On such displays, you probably will notice the slight shifts of objects up and down on alternate refresh cycles.
If we have a complex display to generate, we can draw it into the back buffer, using multiple refresh cycles if needed, and swap buffers when done.

Double buffering is standard in animation, where the primitives, attributes, and viewing conditions may change continuously. We can generate a smooth display using double buffering by putting the drawing functions into a display callback. Within this callback, the first step is to clear the front buffer through `glClear`, and the final step is to invoke `glutSwapBuffers`.

For our rotating cube, we have to make two small changes to go from single to double buffering. First, we request a double buffered display in our `main` function by

```c
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
```

In our display function, we add the line

```c
    glutSwapBuffers();
```

after we have finished drawing our cube.

Double buffering does not solve all problems that we encounter with animated displays. If the display is complex, we still may need multiple frames to draw the image into the frame buffer. Double buffering does not speed up this process; it only ensures that we never see a partial display. However, we are often able to have visibly acceptable displays at rates as low as 10 to 20 frames per second if we use double buffering.

### 3.9.3 Other Buffering Problems

There are other problems that arise in interactive programming that cannot be solved with double buffering alone. Returning to our paint program, suppose that we want to add an elapsed-time clock in one corner of the screen to remind our users of how much time they have spent using our program (Figure 3.23).

In most systems, we can obtain the time of day from the system either as an ASCII string or as seconds elapsed since some initial time. By calling this function repeatedly, perhaps as part of the idle callback, we can compute elapsed time in seconds, which we can then convert to another form, such as `hours:minutes:seconds`. Finally, we can attempt to display these numbers as text.

Consider what happens if we add this elapsed-time facility to our original paint program, which used only single buffering because double buffering was not necessary. You will probably notice the elapsed-time clock flickering as the program clears the part of the screen under the clock and redisesplays the new time. We can try to fix this problem by changing over to double buffering.

If we do so, the clock part of the display no longer will flicker, but we will have created a more serious problem: The objects we painted will flicker. A moment of thought should reveal the problem. Because we are using immediate-mode graphics, objects that we create by painting are drawn into the frame buffer only
Once. Each object will be in one of the two buffers, but it will not be in both. Thus, although we have stopped the clock from flickering, the objects that we paint may now flicker.

There are solutions to this problem. For our paint program, we could use display lists to store objects as they are painted. We could then clear the entire frame buffer after each swap and redraw into it all the now retained objects. However, we would have created a much more complex program with significant overhead, especially if we used a new display list for each character or even each pixel that we created. Of course, we could modify such a program so that we could edit and store the images we paint; in general, however, this approach is less than ideal.

What we really would like to do is to have each object that we create by painting appear in both the front buffer and the back buffer. OpenGL supports a variety of buffers that make up the frame buffer. The front and back buffers are color buffers that hold information that can be displayed on the screen. Normally, when we use double buffering, we would like OpenGL to draw into the back buffer, but we can control which color buffer is used for drawing through the function `glDrawBuffer()`. The default, when double buffering has been enabled, is equivalent to

```c
glDrawBuffer(GL_BACK);
```

If we use

```c
glDrawBuffer(GL_FRONT_AND_BACK);
```
primitives will be rendered into both the front and back buffers. Thus, by switching the drawing buffer in our program, we can draw the primitives that we specify interactively in both buffers but update the clock only in the back buffer, swapping buffers whenever the clock has been completely redrawn.

In Chapter 7, we shall discuss how some of these buffers can be used to create new effects.

3.10 Design of Interactive Programs

Defining what characterizes a good interactive program is difficult, but recognizing and appreciating a good interactive program is easy. Such programs include features such as these:

1. A smooth display, showing neither flicker nor any artifacts of the refresh process
2. A variety of interactive devices on the display
3. A variety of methods for entering and displaying information
4. An easy-to-use interface that does not require substantial effort to learn
5. Feedback to the user
6. Tolerance for user errors
7. A design that incorporates consideration of both the visual and motor properties of the human

The importance of these features and the difficulty of designing a good interactive program should never be underestimated. The field of human–computer interaction (HCI) is an active one, and we shall not shortchange you by condensing it into a few pages. Our concern in this book is computer graphics; within this topic, our primary interest is rendering. However, there are a few topics common to computer graphics and HCI that we can pursue to improve our interactive programs.

3.10.1 Toolkits, Widgets, and the Frame Buffer

In our paint program, we have used interactive tools, such as pop-up menus, that were provided by GLUT, and graphical buttons that we constructed in our programs. There are many more possibilities, such as slidebars, dials, hot areas of the screen, sound, and icons. Usually, these tools are supplied with various toolkits, although there is nothing to prevent us from writing our own. In general, these toolkits use callbacks to interface with application programs, and should be a simple extension of our development.

However, the simple model of rendering geometric objects into a color buffer is not sufficient to support many of these operations. Two examples illustrate the limitations of geometric rendering and show why, at times, we need to work directly in the frame buffer. First, consider our pop-up menus. When a menu callback is invoked, the menu appears over whatever was on the display. After we
make our selection, the menu disappears, and the screen is restored to the state in which it was before the menu was displayed.

Our second example is rubberbanding, a technique used for displaying line segments (and other primitives) that change as they are manipulated interactively. In our painting program, we indicated the endpoints of our desired line segment by two successive mouse locations. We did not have a mechanism for ensuring that any particular relationship would hold between these two locations, and we were not able to interact with the display to help us place the points. Suppose that, after we locate the first point, then, as we move the mouse, a line segment is drawn automatically (and is updated on each refresh), from the first location to the present position of the mouse, as shown in Figure 3.24. This process is called rubberbanding because the line segment that we see on the display appears to be an elastic band, with one end fixed to the first location and the second end stretched to wherever the cursor moves. Note that, before each new line segment appears, the previous line segment must be erased. Usually, the rubberbanding begins when a mouse button is depressed, and continues until the button is released, at which time a final line segment is drawn.

We cannot implement this sequence of operations using only what we have presented so far. One way to implement these operations is to store the part of the display under the menu (or line) and to copy it back when the menu (or line) is no longer needed. Unfortunately, this description of what we want to do is in terms of what bits are on the display, or, equivalently, of what is contained in the frame buffer. One potential solution does not involve primitives in the application program; it involves only their scan-converted images. Consequently, a set of operations for implementing such operations should be described in terms of the contents of the frame buffer—that is, in terms of blocks of bits with what are called bit-block-transfer (bitblt) operations. We can take this approach once we have discussed bits and pixels in Chapter 7. There is another approach, supported by most OpenGL implementations that can provide many of these capabilities in a simple manner that we shall explore next.

### 3.11 Logic Operations

When a program specifies a primitive that is visible, OpenGL renders it into a set of colored pixels that are written into the present drawing buffer. In the default mode of operation, these pixels simply replace the corresponding pixels that are already in the frame buffer. Thus, if we start with a color buffer that has been cleared to black and draw a blue rectangle that is 10 × 10 pixels, then 100 blue pixels are copied into the color buffer, replacing 100 black pixels. In this mode, called copy or replacement mode, it does not matter what color the original pixels under the rectangle were before they were colored blue, because the blue pixels replace pixels of any color that were there. Although this mode of writing into a buffer may seem obvious, it is not the only way to do the write operation.
Consider the model in Figure 3.25, where we consider the writing of a single pixel into a color buffer. The pixel that we want to write is called the source pixel. The pixel in the drawing buffer that the source pixel will affect is called the destination pixel. In copy mode, the source pixel replaces the destination pixel. But suppose that the implementation can first look at the destination pixel and use its value in combination with the source pixel to determine the value to place in the color buffer.

Although we could pick the function that combines the source and destination pixels in an infinite number of ways, most hardware supports only bitwise operations between the pixels. There are 16 possible functions between two bits. Each defines a writing mode. We shall discuss these modes in more detail in Chapter 7. At this point, we are interested in two of these functions. The first is replacement mode in which each bit in the source pixel replaces the corresponding bit in the destination pixel.

The other mode is the exclusive OR or XOR mode in which corresponding bits in each pixel are combined using the exclusive or logical operation. If $s$ and $d$ are corresponding bits in the source and destination pixels, we can denote the new destination bit as $d'$, and it is given by

$$d' = d \oplus s,$$

where $\oplus$ denotes the XOR operation. The most interesting property of the XOR operation is that if we apply it twice, we return to the original state. That is,

$$d = (d \oplus s) \oplus s,$$

Thus, if we draw something in XOR mode, we can erase it by simply drawing it a second time.

OpenGL supports all 16 logic modes. Copy mode (GL_COPY) is the default. If we wish to change modes, we must enable logic operations

```cpp
glEnable(GL_COLOR_LOGIC_OP);
```
and change to XOR mode when desired by

```c
glLogicOp(GL_XOR);
```

### 3.11.1 Drawing Erasable Lines

Using the XOR drawing mode, we can draw erasable lines in a variety of ways. Here is one simple method. Suppose that the OpenGL window is 500 × 500 pixels and the clipping window is a unit square with the origin at the lower-left corner. We use the mouse to get the first endpoint and store this point in world coordinates

```c
xm = x/500.;
ym = (500-y)/500.;
```

We then can get the second point and draw a line segment in XOR mode

```c
xmm = x/500.;
ymm = (500-y)/500.;
glLogicOp(GL_XOR);
glBegin(GL_LINES);
    glVertex2f(xm, ym);
    glVertex2f(xmm, ymm);
glEnd();
glLogicOp(GL_COPY);
glFlush();
```

We switch back to copy mode in case we want to draw other objects in the normal mode. If we enter another point with the mouse, we first draw the same line in XOR mode and then a second line using the first endpoint and the mouse input

```c
glLogicOp(GL_XOR);
glBegin(GL_LINES);
    glVertex2f(xm, ym);
    glVertex2f(xmm, ymm);
glEnd();
glFlush();
xmm = x/500.0;
ymm = (500-y)/500.0;
glBegin(GL_LINES);
    glVertex2f(xm, ym);
    glVertex2f(xmm, ymm);
glLogicOp(GL_COPY);
glEnd();
glFlush();
```
We can sequence these operations by using the right mouse button to enter these data and the left mouse button to indicate that we accept the line segment and wish to draw it in its final form

\[\text{glLogicOp(GL\_COPY);}\]
\[\text{glBegin(GL\_LINES);}\]
\[\text{glVertex2f(xm, ym);}\]
\[\text{glVertex2f(xmm, ymm);}\]
\[\text{glEnd();}\]
\[\text{glFlush();}\]
\[\text{glLogicOp(GL\_XOR);}\]

We can keep track of vertices by adding a counter, as in the paint program, to store which endpoint is expected next.

This example is not quite a rubber-band line, which is drawn continuously as the user moves the mouse. We can get rubber-band line by using the same operations in conjunction with a motion callback. Thus, when the user first depresses a designated mouse button, the first location is saved through the mouse callback. As the mouse moves with the key depressed, line segments are drawn from the first point to a second point determined through the motion callback, each time first redrawing the previous line segment to erase it. When the mouse button is released, a final line segment is drawn in copy mode from the mouse callback. Here is the code for a rubber-band rectangle

```c
void mouse(int btn, int state, int x, int y)
{
    if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    {
        xm = x/500.;
        ym = (500-y)/500.;
        glColor3f(0.0, 0.0, 1.0);
        glLogicOp(GL\_XOR);
        first = 0;
    }
    if(btn==GLUT_LEFT && state == GLUT_UP)
    {
        glRectf(xm, ym, xmm, ymm);
        glFlush();
        glColor3f(0.0, 1.0, 0.0);
        glLogicOp(GL\_COPY);
        xmm = x/500.0;
        ymm = (500-y)/500.0;
        glLogicOp(GL\_COPY);
        glRectf(xm, ym, xmm, ymm);
        glFlush();
    }
}
```
void move(int x, int y)
{
    if( first == 1)
    {
        glRectf(xm, ym, xmm, ymm);
        glFlush();
    }
    xmm = x/500.0;
    ymm = (500-y)/500.0;
    glRectf(xm, ym, xmm, ymm);
    glFlush();
    first = 1;
}

3.11.2 XOR and Color

If you run our previous example, you should notice some odd color combinations where the temporary lines are drawn over pixels that have already been colored. In the example, we used blue for the temporary color as we did the rubberbanding and green for the final color. Suppose that we are using a system that stores color components to eight bits, so that blue is stored as the 24-bit RGB values (00000000, 00000000, 11111111). Suppose that the screen is cleared to white or (11111111, 11111111, 11111111). Then when we draw the blue line using XOR mode, we will see a line in the RGB color (11111111, 11111111, 00000000) or yellow, rather than blue, because the XOR operation is applied bitwise. If the line crosses a red object, it will be colored magenta inside the object. These possibly annoying visual effects are a consequence of our use of the XOR write mode and should make it clear why a final drawing using copy mode is required.

Another common use of the XOR mode is for drawing a cursor on the display. Cursors can be small rectangles, small crosses, or other shapes. Often in CAD programs we use crosshairs, two perpendicular lines that can extend to rulers on the edge of the display. If these objects are drawn in XOR mode, then we can move them around the screen without distorting anything else on the display. However, if there are objects on the display the color of the objects under the cursor will be altered as the cursor moves across them.

3.11.3 Cursors and Overlay Planes

Rubberbanding and cursors can place a significant burden on graphics systems as they require the display to be updated constantly. Although the use of XOR mode appears to simplify the process, XOR requires the system to read the present destination pixels before computing the new destination pixels. These operations are slower than simply copying source pixels to the frame buffer.
An alternative is to provide hardware support for these types of interaction by providing extra bits in the color buffers by adding what are called overlay planes. Thus, a typical color buffer as in Figure 3.26 may have 8 bits for each of red, green, and blue and one red, one green, and one blue overlay plane.

The contents of the overlay planes are independent of what is in the color buffer. In addition, when the color buffer and overlay planes are displayed, the contents of the overlay planes act as if their values were copied into the color buffer. In other words, if there is a bit set in the red overlay plane and the corresponding bits in the blue and green overlay planes are not set, the corresponding point in the window will be red, regardless of what is in the matching 24 bits of the RGB-color buffer.

Overlay planes are a hardware feature and are not present in all systems. GLUT provides support for overlay planes in color index mode.

### 3.12 Summary and Notes

In this chapter, we have touched on a number of topics related to interactive computer graphics. These interactive aspects make the field of computer graphics exciting and fun. Although our API, OpenGL, is independent of any operating or window system, we recognize that any program must have at least minimal interaction with the rest of the computer system. We handled simple interactions by using a simple toolkit, GLUT, whose API provides the necessary additional functionality, without being dependent on a particular operating or window system.

We have been heavily influenced by the client–server perspective. Not only does it allow us to develop programs within a networked environment, but also it allows us to design programs that are portable yet can still take advantage of special features that might be available in the hardware. These concepts will be crucial in our discussion of object-oriented graphics and graphics for the World Wide Web in Chapter 9.
From the application programmer’s perspective, various characteristics of interactive graphics are shared by most systems. We see the graphics part of the system as a server, consisting of a raster display, a keyboard, and a pointing device. In almost all workstations, we have to work within a multiprocessing windowed environment. Most likely, many processes are executing concurrently with the execution of your graphics program. However, the window system allows us to write programs for a specific window that act as though this window is the display device of a single-user system.

The overhead of setting up a program to run in this environment is small. Each application program contains a set of function calls that is virtually the same in every program. The use of logical devices within the application program frees the programmer from worrying about the details of particular hardware.

Within the environment that we have described, event-mode input is the norm. Although the other forms are available—request mode is the normal method used for keyboard input—event-mode input gives us far more flexibility in the design of interactive programs.

Interactive computer graphics is a powerful tool with unlimited applications. At this point, you should be able to write fairly sophisticated interactive programs. Probably the most helpful exercise that you can do now is to write one. The exercises at the end of the chapter provide suggestions.

3.13 Suggested Readings

Sutherland’s Project Sketchpad is described in [Sut63].

Many of the conceptual foundations for the windows-icons-menus-pointing interfaces that we now consider routine were developed at the Xerox Palo Alto Research Center (PARC) during the 1970s; see [Sch87]. The mouse also was developed there [Eng68]. The familiar interfaces of today—such as the Macintosh Operating System, the X Window system, and Microsoft Windows—all have their basis in this work.

The volume by Foley and associates [Fol94] contains a thorough description of the development of user interfaces, with an emphasis on the graphical aspects. The books by Schneiderman [Sch97] and Nielsen [Nie94] provide an introduction to HCI.

The X Window system [Sch88] was developed at the Massachusetts Institute of Technology and is the de facto standard in the UNIX workstation community. Recently, the development of the LINUX version for PCs has allowed the X Window system to run on these platforms too.

The input and interaction modes that we discussed grew out of the standards that led to GKS [ANSI85] and PHIGS [ANSI88]. These standards were developed for both calligraphic and raster displays; thus, they do not take advantage of the possibilities available on raster-only systems (see [Pik84, Gol83]).

Although we have used the GLUT toolkit [Kil94b] exclusively, we can also interface directly with the X Window system, with various X Window toolkits.
Input and Interaction

[Kil94a, OSF89], and with Microsoft Windows. Also of interest is the use of scripting languages, such as tcl/tk [Ous94], to develop user interfaces that work with OpenGL. Additional details on GLUT are in the OpenGL Primer [Ang02]. There are also some other interface toolkits available for OpenGL. See the OpenGL website www.opengl.org.

Exercises

3.1 Explain problems that you face in defining a stroke font. Create a simple data structure that will allow you to define a set of stroke characters using only line segments.

3.2 Rewrite the Sierpinski-gasket program from Chapter 2 such that the left mouse button will start the generation of points on the screen, the right mouse button will halt the generation of new points, and the middle mouse button will terminate the program. Include a reshape callback.

3.3 Construct slidebars to allow users to define colors in the paint program. Your interface should let the user see a color before that color is used.

3.4 We can construct a virtual trackball from a mouse by mapping the mouse pad onto a ball. A major use of such a device is to allow the user to obtain velocities by using the mouse to “spin” the ball. Construct such a virtual device, and write a graphical application to demonstrate its use.

3.5 Alter the square-drawing program (Section 3.5) to incorporate menus like those described in Section 3.6.

3.6 Add an elapsed-time indicator in the paint program (Section 3.8) using a clock of your own design.

3.7 Creating simple games is a good way to become familiar with interactive graphics programming. Program the game of checkers. You can look at each square as an object that can be picked by the user. You can start with a program in which the user plays both sides.

3.8 Write a program that allows a user to play a simple version of solitaire. First, design a simple set of cards, using only our basic primitives. Your program can be written in terms of picking rectangular objects.

3.9 Simulating a pool or billiards game presents interesting problems. Here, as in Exercise 2.18, you must compute trajectories and detect collisions. The interactive aspects include initiating movement of the balls through a graphical cue stick, ensuring that the display is smooth, and creating a two-person game.

3.10 Rather than using buttons or menus to select options in an interactive program, we can make selections based on where in the window the mouse is located. Use this mechanism in the paint program (Section 3.8).
3.11 The mapping from a point in world coordinates to one in screen coordinates is well defined. It is not invertible because we go from three dimensions to two dimensions. Suppose, however, that we are working with a two-dimensional application. Is the mapping invertible? What problem can arise if you use a two-dimensional mapping to return a position in world coordinates by a locator device?

3.12 How do the results of Exercise 3.11 apply to picking?

3.13 In a typical application program, the programmer must decide whether or not to use display lists. Consider at least two applications. For each, list at least two factors in favor and at least two factors against the use of display lists. Using the fonts provided by GLUT, test whether what you wrote down is correct in practice.

3.14 Write an interactive program that will allow you to guide a graphical rat through the maze that you generated in Exercise 2.7. You can use the left and right buttons to turn the rat, and the middle button to move him forward.

3.15 Add rubber-band lines and rectangles to the paint program.

3.16 Inexpensive joysticks, such as those used in toys and games, often lack encoders and contain only a pair of three-position switches. How might such devices function?

3.17 The orientation of an airplane is described by a coordinate system oriented as shown in Figure 3.27. The forward–backward motion of the joystick controls the up–down rotation with respect to the axis running along the length of the airplane, called the pitch. The right–left motion of the joystick controls the rotation about this axis, called the roll. Write a program that uses the mouse to control pitch and roll for the view seen by a pilot. You can do this exercise in two dimensions by considering a set of objects to be located far from the airplane, and then having the mouse control the two-dimensional viewing of these objects.

3.18 Consider a table with a two-dimensional sensing device located at the end of two linked arms, as shown in Figure 3.28. Suppose that the lengths of the two arms are fixed, and the arms are connected by simple (1-degree-of-freedom) pivot joints. Determine the relationship between the joint angles θ and φ and the position of the sensor.

3.19 Suppose that a CRT has a square face of 40 × 40 centimeters and is refreshed in a noninterlaced manner at a rate of 60 Hz. Ten percent of the time that the system takes to draw each scan line is used to return the CRT beam from the right edge to the left edge of the screen (the horizontal-retrace time), and 10 percent of the total drawing time is allocated for the beam to return from the lower-right corner of the screen to the upper-left corner after each refresh is complete (the vertical-retrace time). Assume that the resolution of the display is 1024 × 1024 pixels. Find a relationship between the time at which a lightpen detects the
beam and the lightpen’s position. Give the result using both centimeters and screen coordinates for the location on the screen.

3.20 Circuit-layout programs are variants of paint programs. Consider the design of logical circuits using the Boolean AND, OR, and NOT functions. Each of these functions is provided by one of the three types of integrated circuits (gates), the symbols for which are shown in Figure 3.29. Write a program that allows the user to design a logical circuit by selecting gates from a menu and positioning them on the screen. Consider methods for connecting the outputs of one gate to the inputs of others.

3.21 Extend Exercise 3.20 to allow the user to specify a sequence of input signals. Have the program display the resulting values at selected points in the circuit.

3.22 Extend Exercise 3.20 to have the user enter a logical expression. Have the program generate a logical diagram from that expression.

3.23 Use the methods of Exercise 3.20 to form flowcharts for programs or images of graphs that you have studied in a data-structures class.

3.24 Plotting packages offer a variety of methods for displaying data. Write an interactive plotting application for two-dimensional curves. Your application should allow the user to choose the mode (polyline display of the data, bar chart, or pie chart), colors, and line styles.

3.25 The required refresh rate for CRT displays of 50 to 85 Hz is based on the use of short-persistence phosphors that emit light for extremely short intervals when excited. Long-persistence phosphors are available. Why are long-persistence phosphors not used in most workstation displays? In what types of applications might such phosphors be useful?

Figure 3.29 Symbols for logical circuits.