Chapter 11: Events, Event Handling, and Object Selection

Graphics programming goes much farther than creating a fixed image or animation based on a principle or a set of data. More and more we see the value of creating applications that allow the user to interact with a graphical presentation, that allow the user to control the way an image is created, or that use interaction with a graphical model of a process to control the process itself. These interactive images can be used in entertainment, in education, in engineering, or in the sciences. These are called interactive computer graphics applications, and the ability to interact with information through an image is critically important to the success of this field.

The set of capabilities for interaction is called a user interface, and there is a considerable science of creating and evaluating interfaces for different kinds of interactive processes. However, the emphasis in this chapter is on graphical interaction, not on user interfaces. Certainly many user interfaces use graphical presentations that give information to the user, take graphical actions, and interpret the results for program control, but we simply view these as applications of our graphics. Late in the chapter we introduce the MUI (Micro User Interface) system that allows you to add a primitive interface to your OpenGL programs, and we believe very strongly that you should try to understand the nature of the communication about images that can be supported by an external user interface, but a genuine discussion of user interfaces is much too deep for us to undertake here. In general, we subscribe to the view that computer interaction should be designed by persons who are specially trained in human factors, interface design, and evaluation, and not by computer scientists, but computer scientists will implement the design. This chapter describes how you can implement such actions.

Interactive programming in computer graphics generally takes advantage of the event-handling capabilities of modern systems, so we must understand something of what events are and how to use them in order to write interactive graphics programs. Events are fairly abstract and come in several varieties, so we will need to go into some details as we develop this idea below. But modern graphics APIs handle events pretty cleanly, and you will find that once you are used to the idea, it is not particularly difficult to write event-driven programs. You should realize that some basic APIs do not include event handling on their own, so you may need to use an extension to the API for this.

When you have finished working through this chapter, you should be able to understand the range of interactive capabilities offered by a standard graphics API and should be able to implement an interactive graphics program using appropriate kinds of event-driven tools.

Definitions

An event is a transition in the control state of a computer system. Events can come from many sources and can cause any of a number of actions to take place as the system responds to the transition. In general, we will treat an event as an abstraction, a concept that we use to design interactive applications, that provides a concrete piece of data to the computer system. An event record is a formal record of some system activity, often an activity from a device such as a keyboard or mouse. An event record is a data structure that contains information that identifies the event and any data corresponding to the event. This is not a user-accessible data structure, but its values are returned to the system and application by appropriate system functions. A keyboard event record contains the identity of the key that was pressed and the location of the cursor when it was pressed, for example; a mouse event record contains the mouse key that was pressed, if any, and the cursor’s location on the screen when the event took place. Event records are stored in the event queue, which is managed by the operating system; this keeps track of the sequence in which events happen and serves as a resource to processes that deal with events. When an event occurs, its event record is inserted in the event queue and we say that the event is posted to the queue. The operating system manages the event queue and as each event gets to the front of the queue and a
process requests an event record, the operating system passes the record to the process that should handle it. In general, events that involve a screen location get passed to whatever program owns that location, so if the event happens outside a program’s window, that program will not get the event.

Let’s consider a straightforward example of a user action that causes an event in an application, and think about what is actually done to get and manage that event. This will vary from system to system, so we will consider two alternatives. In the first, the application includes a system utility that polls for events and, when one occurs in the system, identifies the event and calls an appropriate event handler. In the second, the application initializes an event loop and provides event handlers that act like callbacks to respond to events as the system gets them. A graphics API might use either model, but we will focus on the event loop and callback model.

Programs that use events for control—and most interactive programs do this—manage that control through functions that are called event handlers. While these can gain access to the event queue in a number of ways, most APIs use functions called callbacks to handle events. Associating a callback function with an event is called registering the callback for the event. When the system passes an event record to the program, the program determines what kind of event it is and if any callback function has been registered for the event, passes control to that function. In fact, most interactive programs contain initialization and action functions, callback functions, and a main event loop. The main event loop invokes an event handler whose function is to get an event, determine the callback needed to handle the event, and pass control to that function. When that function finishes its operation, control is returned to the event handler.

What happens in the main event loop is straightforward—the program gives up direct control of the flow of execution and places it in the hands of the user. From this point throughout the remainder of the execution of the program, the user will cause events to which the program will respond through the callbacks that you have created. We will see many examples of this approach in this, and later, chapters.

A callback is a function that is executed when a particular event is recognized by the program. This recognition happens when the event handler takes an event off the event queue and the program has expressed an interest in the event. The key to being able to use a certain event in a program, then, is to express an interest in the event and to indicate what function is to be executed when the event happens—the function that has been registered for the event.

Some examples of events

Events are often categorized in a set of classes, depending on the kind of action that causes the event. Below we describe one possible way of classifying events that gives you the flavor of this concept.

keypress events, such as keyDown, keyUp, keyStillDown, ... Note that there may be two different kinds of keypress events: those that use the regular keyboard and those that use the so-called “special keys” such as the function keys or the cursor control keys. There may be different event handlers for these different kinds of keypresses. You should be careful when you use special keys, because different computers may have different special keys, and those that are the same may be laid out in different ways.

mouse events, such as leftButtonDown, leftButtonUp, leftButtonStillDown, ... Note that different “species” of mice have different numbers of buttons, so for some kinds of mice some of these events are collapsed.
menu events, such as selection of an item from a pop-up or pull-down menu or submenu, that are based on menu choices.

window events, such as moving or resizing a window, that are based on standard window manipulations.

system events, such as idle and timer, that are generated by the system based on the state of the event queue or the system clock, respectively.

software events, which are posted by programs themselves in order to get a specific kind of processing to occur next.

These events are very detailed, and many of them are not used in the APIs or API extensions commonly found with graphics. However, all could be used by going deeply enough into the system on which programs are being developed.

Note that event-driven actions are fundamentally different from actions that are driven by polling—that is, by querying a device or some other part of the system on some schedule and basing system activity on the results. There are certainly systems that operate by polling various kinds of input and interaction devices, but these are outside our current approach.

The vocabulary of interaction

Interaction can take many forms, but in computer graphics interaction is generally visual and it is part of the general area of visual communication. We discuss that aspect of interaction in the visual communication chapter of this book. Here we want to focus on the relation of interaction to events caused by computing devices, both hardware and software.

When users are working with your application, they are focusing on the content of their work, not on how you designed the application. They want to be able to communicate with the program and their data in ways that feel natural to them, and it is the task of the interface designer to create an interface that feels very natural and that doesn’t interfere with their work. Interface design is the subject of a different course from computer graphics, but it is useful to have a little understanding of the vocabulary of interaction.

We have been focusing on how to program interaction with the kind of devices that are commonly found in current computers: keyboards or mice. These devices have distinctly different kinds of behaviors in users’ minds. When you get away from text operations, keyboards give discrete input that can be interpreted in different ways depending on the keys that are pressed. They are basically devices that make abstract selections, with the ability select actions as well as objects. The keyboard input that navigates through simple text games is an example of action selection. The mouse buttons are also selection devices, although they are primarily used to select graphical objects on the screen, including control buttons as well as displayed objects. The keyboard and mouse buttons both are discrete devices, providing only a finite number of well-defined actions.

The mouse itself has a different kind of meaning. It provides a more continuous input, and can be used to control continuous motion on the screen. This can be the motion of a selected object as it is moved into a desired position, or it can be an input that will cause motion in an object. The motion that the mouse controls can be of various kinds as well—it can be a linear motion, such as moving the eye point across a scene, or it can be a rotational motion, such as moving an object by changing the angles defining the object in spherical coordinates.
When you plan the interaction for your application, then, you should decide whether a user will see the interaction as a discrete selection or as a continuous control, and then you should implement the interaction with the keyboard or mouse, as determined by the user’s expected vocabulary.

There are other kinds of devices that are not ordinarily found on most computers, but that may well be more common in the future. If we consider the motion of the mouse, we see that we have only four degrees of freedom represented by changes in $X$ and $Y$, both positive and negative. This can let us control things like the direction we are looking, either up or down, left or right, but it cannot at the same time let us move forward and backward. For that we need two more degrees of freedom, and there are devices that will allow you to move left and right, up and down, and forward and back; these 6DF (six degree of freedom) devices are sometimes used for controlling virtual environments and could readily be adapted to a graphics API and to graphics applications.

Object selection

Earlier in this chapter on interaction, we saw many ways that a graphics API can support user inputs into your applications through menus, keystrokes, and mouse functions that can manipulate your images. If you wanted to identify a particular object in the scene to act on, you could use those tools to identify an object by name, but you could not identify an object simply by clicking on it. In this section we will show you how you can make that kind of selection, or to use the term that’s most common in graphics, we will show you how to *pick* an object in a scene. This kind of object picking permits the user to interact with a scene in a much more direct way than is possible with the kind of external events, such as menu selections, mouse clicks, or key presses, that we saw in the earlier chapter on event handling.

With object picking we can get the kind of direct manipulation that we are familiar with from graphical user interfaces, where the user selects a graphical object and then applies operations to it. Conceptually, picking allows your user to identify a particular object with the cursor and to choose it by clicking the mouse button when the cursor is on the object. The program must be able to identify what was selected, and then must have the ability to apply whatever action the user chooses to that particular selected object.

To understand how picking works, let’s start with the mouse click. When you get a mouse event, the event callback gets four pieces of information: the button that was clicked, the state of that button, and the integer coordinates of the point in the window where the event happened. In order to find out what object might be indicated by the click, we convert the window coordinates to 2D eye coordinates but we must then reverse the projection and go back from 2D eye space to 3D eye space. However, a single point in 2D eye space becomes a line in 3D eye space, as shown in Figure 11.1. Our problem then becomes how to identify what objects meet this line segment and which of those objects was chosen by the user.

![Figure 11.1: the line in the view volume that corresponds to a point in 2D eye space](image-url)
It is possible to make this computation directly from the geometry using collision detection logic. That is, for each object in the scene, we calculate whether or not the line and object intersect. When we have done all these calculations, we can say which objects lie under the point that was chosen and where each of those intersections takes place in 3D eye space. We may then choose the closest, which is be the one that the user would see at the place the mouse click was made, or we may choose any other intersection that our logic demands. The problem with this is that it is very computation-intensive and requires us to be able to go back to eye space, and that makes it difficult to implement.

Another way to identify which objects lie on the line is to invert this logic completely. Instead of focusing on the objects that lie under the line of the pick, we see that any object that we might have chosen will use the pixel that was chosen when it was rendered. So if we can keep track of that pixel and save information on any object that includes it, we can identify all the objects that lie under the pick point. Because the rendering process keeps track of depth values, we can also get information on the depth of the object in the view volume when the pixel is used. Note that we do not talk about what kind of information is saved, so there are some interesting opportunities to think about what might be done here.

We cannot say which of these techniques might be used by any particular graphics application, and we cannot say which might be used by a particular graphics API. But OpenGL uses the second technique, and in the rest of this chapter we discuss how that is done.

Events and the scene graph

As we think about designing interaction for a scene, it can be very useful to use the scene graph to design our event handling. In the scene graph we have four kinds of nodes: group nodes, transformation nodes, geometry nodes, and appearance nodes. We can readily see interactions that affect many of these. We can change transformations to move objects in the space of a scene or to change the position or orientation of the view. We can change appearance to indicate what has been selected or simply to change the appearance by adding or removing particular rendering actions. We can change geometry to select an appropriate geometry for an object or to replace an object with another object or collection of objects. So we have a number of ways we can attach events and interactions to entities in the scene graph.

Perhaps one of the most important ways an event can affect a scene graph is selecting a particular geometric object or group of objects to have special treatment, whether in motion (transformations) or appearance or presentation. This kind of action requires triggering particular objects for action and so will need to have triggers in various scene graph nodes, with the trigger activated by the selection. This goes beyond the original modeling context of the scene graph, but can be added informally. You should incorporate events and interaction into your scene graph and use it to manage the actions of your interaction.

A word to the wise...

This section discusses the mechanics of interaction through event handling, but it does not cover the critical questions of how a user would naturally control an interactive application. There are many deep and subtle issues involved in designing the user interface for such an application, and this module does not begin to cover them. The extensive literature in user interfaces will help you get a start in this area, but a professional application needs a professional interface, one designed, tested, and evolved by persons who focus in this area. When thinking of a real application, heed the old cliché: Kids, don’t try this at home!

The examples below do their best to present user controls that are not impossibly clumsy, but they are designed much more to focus on the event and callback than on a clever or smooth way for a
user to work. When you write your own interactive projects, think carefully about how a user might perceive the task, not just about an approach that might be easiest for you to program.

**Events in OpenGL**

The OpenGL API generally uses the Graphics Library Utility Toolkit GLUT (or a similar extension) for event and window handling. GLUT defines a number of kinds of events and gives the programmer a means of associating a callback function with each event that the program will use. In OpenGL with the GLUT extension, this main event loop is quite explicit as a call to the function `glutMainLoop()` as the last action in the main program.

**Callback registering**

Below we will list some kinds of events and will then indicate the function that is used to register the callback for each event. Following that, we will give some code examples that register and use these events for some programming effects. This now includes only examples from OpenGL, but it should be extensible to other APIs fairly easily.

<table>
<thead>
<tr>
<th>Event</th>
<th>Callback Registration Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>idle</td>
<td><code>glutIdleFunc(functionname)</code></td>
</tr>
<tr>
<td></td>
<td>requires a callback function with template</td>
</tr>
<tr>
<td></td>
<td><code>void functionname(void)</code></td>
</tr>
<tr>
<td></td>
<td>as a parameter. This function is the event handler that determines</td>
</tr>
<tr>
<td></td>
<td>what is to be done at each idle cycle. Often this function will end</td>
</tr>
<tr>
<td></td>
<td>with a call to <code>glutPostRedisplay()</code> as described below. This function</td>
</tr>
<tr>
<td></td>
<td>is used to define what action the program is to take when there has</td>
</tr>
<tr>
<td></td>
<td>been no other event to be handled, and is often the function that</td>
</tr>
<tr>
<td></td>
<td>drives real-time animations.</td>
</tr>
<tr>
<td>display</td>
<td><code>glutDisplayFunc(functionname)</code></td>
</tr>
<tr>
<td></td>
<td>requires a callback function with template</td>
</tr>
<tr>
<td></td>
<td><code>void functionname(void)</code></td>
</tr>
<tr>
<td></td>
<td>as a parameter. This function is the event handler that generates a</td>
</tr>
<tr>
<td></td>
<td>new display whenever the display event is received. Note that the</td>
</tr>
<tr>
<td></td>
<td>display function is invoked by the event handler whenever a display</td>
</tr>
<tr>
<td></td>
<td>event is reached; this event is posted by the <code>glutPostRedisplay()</code></td>
</tr>
<tr>
<td></td>
<td>function and whenever a window is moved or reshaped.</td>
</tr>
<tr>
<td>reshape</td>
<td><code>glutReshapeFunc(functionname)</code></td>
</tr>
<tr>
<td></td>
<td>requires a callback function with template</td>
</tr>
<tr>
<td></td>
<td><code>void functionname(int, int)</code></td>
</tr>
<tr>
<td></td>
<td>as a parameter. This function manages any changes needed in the view</td>
</tr>
<tr>
<td></td>
<td>setup to accommodate the reshaped window, which may include a fresh</td>
</tr>
<tr>
<td></td>
<td>definition of the projection. The parameters of the reshape function</td>
</tr>
<tr>
<td></td>
<td>are the width and height of the window after it has been changed.</td>
</tr>
<tr>
<td>keyboard</td>
<td><code>glutKeyboardFunc(keybd)</code></td>
</tr>
<tr>
<td></td>
<td>requires a callback function with template</td>
</tr>
<tr>
<td></td>
<td><code>void functionname(unsigned char, int, int)</code></td>
</tr>
<tr>
<td></td>
<td>as a parameter. This parameter function is the event handler that</td>
</tr>
<tr>
<td></td>
<td>receives the character and the location of the cursor (int x, int y)</td>
</tr>
<tr>
<td></td>
<td>when a key is pressed. As is the case for all callbacks that involve</td>
</tr>
<tr>
<td></td>
<td>a screen location, the location on the screen will be converted to</td>
</tr>
<tr>
<td></td>
<td>coordinates relative to the window. Again, this function will often</td>
</tr>
<tr>
<td></td>
<td>end with a call to <code>glutPostRedisplay()</code> as described below. This</td>
</tr>
<tr>
<td></td>
<td>function is used to define what action the program is to take when</td>
</tr>
<tr>
<td></td>
<td>there has been no other event to be handled, and is often the function</td>
</tr>
<tr>
<td></td>
<td>that drives real-time animations.</td>
</tr>
</tbody>
</table>
glutPostRedisplay() to re-display the scene with the changes caused by the particular keyboard event.

**special**

GlutSpecialFunc(special)
requires a callback function with template

```c
void functionname(int key, int x, int y)
```
as a parameter. This event is generated when one of the “special keys” is pressed; these keys are the function keys, directional keys, and a few others. The first parameter is the key that was pressed; the second and third are the integer window coordinates of the cursor when the keypress occurred as described above. The usual approach is to use a special symbolic name for the key, and these are described in the discussion below. The only difference between the special and keyboard callbacks is that the events come from different kinds of keys.

**menu**

Int glutCreateMenu(functionname)
requires a callback function with template

```c
void functionname(int)
```
as a parameter. The integer value passed to the function is the integer assigned to the selected menu choice when the menu is opened and a choice is made; below we describe how menu entries are associated with these values.

The glutCreateMenu() function returns a value that identifies the menu for later operations that can change the menu choices. These operations are discussed later in this chapter when we describe how menus can be manipulated. The glutCreateMenu() function creates a menu that is brought up by a mouse button down event, specified by

```c
glutAttachMenu(event),
```
which attaches the current menu to an identified event, and the function

```c
glutAddMenuEntry(string, int)
```
identifies each of the choices in the menu and defines the value to be returned by each one. That is, when the user selects the menu item labeled with the string, the value is passed as the parameter to the menu callback function. The menu choices are identified before the menu itself is attached, as illustrated in the lines below:

```c
glutAddMenuEntry("text", VALUE);
...
```

```c
glutAttachMenu(GLUT_RIGHT_BUTTON)
```
The glutAttachMenu() function signifies the end of creating the menu.

Note that the Macintosh uses a slightly different menu attachment with the same parameters,

```c
glutAttachMenuName(event, string),
```
that attaches the menu to a name on the system menu bar. The Macintosh menu is activated by selecting the menu name from the menu bar, while the windows for Unix and Windows are popup windows that appear where the mouse is clicked and that do not have names attached.

Along with menus one can have sub-menus—items in a menu that cause a cascaded sub-menu to be displayed when it is selected. Sub-menus are created two ways; here we describe adding a sub-menu by using the function

```c
glutAddSubMenu(string, int)
```
where the string is the text displayed in the original menu and the integer is the identifier of the menu to cascade from that menu item. When the string item is chosen in the original menu, the submenu will be displayed. With this GLUT function, you can only add a sub-menu as the last item in a menu, so adding a sub-menu closes the creation of the main menu. However, later in this chapter we describe how you can add more submenus within a menu.

mouse glutMouseFunc(functionname)
requires a callback function with a template such as

```c
void functionname(int button, int state, int mouseX, int mouseY)
```
as a parameter, where button indicates which button was pressed (an integer typically made up of one bit per button, so that a three-button mouse can indicate any value from one to seven), the state of the mouse (symbolic values such as GLUT_DOWN to indicate what is happening with the mouse) — and both raising and releasing buttons causes events — and integer values xPos and yPos for the window-relative location of the cursor in the window when the event occurred.

The mouse event does not use this function if it includes a key that has been defined to trigger a menu.

mouse active motion glutMotionFunc(functionname)
requires a callback function with template

```c
void functionname(int, int)
```
as a parameter. The two integer parameters are the window-relative coordinates of the cursor in the window when the event occurred. This event occurs when the mouse is moved with one or more buttons pressed.

mouse passive motion glutPassiveMotionFunc(functionname)
requires a callback function with template

```c
void functionname(int, int)
```
as a parameter. The two integer parameters are the window-relative coordinates of the cursor in the window when the event occurred. This event occurs when the mouse is moved with no buttons pressed.

timer glutTimerFunc(msec, timer, value)
requires an integer parameter, here called msec, that is to be the number of milliseconds that pass before the callback is triggered; a callback function, here called timer, with a template such as

```c
void timer(int)
```
that takes an integer parameter; and an integer parameter, here called value, that is to be passed to the timer function when it is called.

Note that in any of these cases, the function NULL is an acceptable option for a callback function name. Thus you can create a template for your code that includes registrations for all the events your system can support, and simply register the NULL function for any event that you want to ignore.

Besides the kind of device events we generally think of, there are also software events such as the display event, created by a call to glutPostRedisplay(). There are also device events for devices that are probably not found around most undergraduate laboratories: the spaceball, a six-
degree-of-freedom device used in high-end applications, and the graphics tablet, a device familiar to the computer-aided design world and still valuable in many applications. If you want to know more about handling these devices, you should check the GLUT manual.

Some details

For most of these callbacks, the meaning of the parameters of the event callback is pretty clear. Most are either standard characters or integers such as window dimensions or cursor locations. However, for the special event, the callback must handle the special characters by symbolic names. Many of the names are straightforward, but some are not; the full table is:

- Function keys F1 through F12: GLUT_KEY_F1 through GLUT_KEY_F12
- Directional keys: GLUT_KEY_LEFT, GLUT_KEY_UP, GLUT_KEY_RIGHT, GLUT_KEY_DOWN
- Other special keys: GLUT_KEY_PAGE_UP (Page up)
  GLUT_KEY_PAGE_DOWN (Page down)
  GLUT_KEY_HOME (Home)
  GLUT_KEY_END (End)
  GLUT_KEY_INSERT (Insert)

So to use the special keys, use these symbolic names to process the keypress that was returned to the callback function.

More on menus

Earlier in the chapter we saw how we could create menus, add menu items, and specify the menu callback function. But menus are complex resources that can be managed with much more detail than this. Menus can be activated and deactivated, can be created and destroyed, and menu items can be added, deleted, or modified. The basic tools to do this are included in the GLUT toolkit and are described in this section.

You will have noticed that when you define a menu, the glutCreateMenu() function returns an integer value. This value is the menu number. While the menu you are creating is the active menu while you are creating it, if you have more than one menu you will have to refer to a particular menu by its number when you operate on it. In order to see what the active menu number is at any point, you can use the function

```c
int glutGetMenu(void)
```

that simply returns the menu number. If you need to change the active menu at any point, you can do so by using its number as the argument to the function

```c
void glutSetMenu(int menu)
```

This will make the menu whose number you choose into the active menu so the operations we describe below can be done to it. Note that both main menus and sub-menus have menu numbers, so it is important to keep track of them.

It is also possible to detach a menu from a button, in effect deactivating the menu. This is done by the function

```c
void glutDetachMenu(event)
```

which detaches the event from the current menu.

Menus can be dynamic. You can change the string and the returned value of any menu entry with the function

```c
void glutChangeToMenuEntry(int entry, char *name, int value)
```
where the name is the new string to be displayed and the new value is the value that the event handler is to return to the system when this item is chosen. The menu that will be changed is the active window, which can be set as described above.

While you may only create one sub-menu to a main menu with the `glutAddSubMenu()` function we described above, you may add sub-menus later by using the `void glutChangeToSubMenu(int entry, char *name, int menu)` function. Here the entry is the number in the current menu (the first item is numbered 1) that is to be changed into a submenu trigger, the name is the string that is to be displayed at that location, and menu is the number to be given to the new sub-menu. This will allow you to add sub-menus to any menu you like at any point you like.

Menus can also be destroyed as well as attached and detached. The function `void glutDestroyMenu(int menu)` destroys the menu whose identifier is passed as the parameter to the function.

These details can seem overwhelming until you have a reason to want to change menus as your program runs. When you have a specific need to make changes in your menus, you will probably find that the GLUT toolkit has enough tools to let you do the job.

**Code examples**

This section presents four examples. This first is a simple animation that uses an idle event callback and moves a cube around a circle, in and out of the circle's radius, and up and down. The user has no control over this motion. When you compile and run this piece of code, see if you can imagine the volume in 3-space inside which the cube moves.

The second example uses keyboard callbacks to move a cube up/down, left/right, and front/back by using a simple keypad on the keyboard. This uses keys within the standard keyboard instead of using special keys such as a numeric keypad or the cursor control keys. A numeric keypad is not used because some keyboards do not have them; the cursor control keys are not used because we need six directions, not just four.

The third example uses a mouse callback to pop up a menu and make a menu selection, in order to set the color of a cube. This is a somewhat trivial action, but it introduces the use of pop-up menus, which are a very standard and useful tool.

Finally, the fourth example uses a mouse callback with object selection to identify one of two cubes that are being displayed and to change the color of that cube. Again, this is not a difficult action, but it calls upon the entire selection buffer process that is the subject of another later module in this set. For now, we suggest that you focus on the event and callback concepts and postpone a full understanding of this example until you have read the material on selection.

**Idle event callback**

In this example, we assume we have a function named `cube()` that will draw a simple cube at the origin \((0, 0, 0)\). We want to move the cube around by changing its position with time, so we will let the idle event handler set the position of the cube and the display function draw the cube using the positions determined by the idle event handler. Much of the code for a complete program has been left out, but this illustrates the relation between the display function, the event handler, and the callback registration.
GLfloat cubex = 0.0, cubey = 0.0, cubez = 0.0, time = 0.0;

void display( void )
{
    glPushMatrix();
    glTranslatef( cubex, cubey, cubez );
    cube();
    glPopMatrix();
}

void animate(void)
{
    #define deltaTime 0.05
    // Position for the cube is set by modeling time-based behavior.
    // Try multiplying the time by different constants to see how
    // that behavior changes.
    time += deltaTime; if (time > 2.0*M_PI) time -= 2*0*M_PI;
    cubex = sin(time);
    cubey = cos(time);
    cubez = cos(time);
    glutPostRedisplay();
}

void main(int argc, char** argv)
{
    /* Standard GLUT initialization precedes the functions below*/
    ...
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutIdleFunc(animate);
    myinit();
    glutMainLoop();
}

Keyboard callback

Again we start with the familiar cube() function. This time we want to let the user move the
cube up/down, left/right, or backward/forward by means of simple keypresses. We will use two
virtual keypads:

 Q  W   I  O
 A  S  J  K
 Z  X N  M

with the top row controlling up/down, the middle row controlling left/right, and the bottom row
controlling backward/forward. So, for example, if the user presses either Q or I, the cube will
move up; pressing W or O will move it down. The other rows will work similarly.

Again, much of the code has been omitted, but the display function works just as it did in the
example above: the event handler sets global positioning variables and the display function
performs a translation as chosen by the user. Note that in this example, these translations operate
in the direction of faces of the cube, not in the directions relative to the window.
GLfloat cubex = 0.0;
GLfloat cubey = 0.0;
GLfloat cubez = 0.0;
GLfloat time  = 0.0;

void display( void ) {
  glPushMatrix();
  glTranslatef( cubex, cubey, cubez );
  cube();
  glPopMatrix();
}

void keyboard(unsigned char key, int x, int y) {
  ch = ' ';
  switch (key) {
    case 'q' : case 'Q' :
      ch = key; cubey -= 0.1; break;
    case 'w' : case 'W' :
      ch = key; cubey += 0.1; break;
    case 'a' : case 'A' :
      ch = key; cubex -= 0.1; break;
    case 's' : case 'S' :
      ch = key; cubex += 0.1; break;
    case 'z' : case 'Z' :
      ch = key; cubez -= 0.1; break;
    case 'x' : case 'X' :
      ch = key; cubez += 0.1; break;
  }
  glutPostRedisplay();
}

void main(int argc, char** argv) {
  /* Standard GLUT initialization */
  glutDisplayFunc(display);
  glutKeyboardFunc(keyboard);
  myinit();
  glutMainLoop();
}

The similar function, glutSpecialFunc(...), can be used in a very similar way to read input from the special keys (function keys, cursor control keys, ...) on the keyboard.

Menu callback

Again we start with the familiar cube() function, but this time we have no motion of the cube. Instead we define a menu that allows us to choose the color of the cube, and after we make our choice the new color is applied.
#define RED    1
#define GREEN  2
#define BLUE   3
#define WHITE  4
#define YELLOW 5

void cube(void) {
    ... 
    GLfloat color[4];
    // set the color based on the menu choice
    switch (colorName) {
        case RED: 
            color[0] = 1.0; color[1] = 0.0;
            color[2] = 0.0; color[3] = 1.0; break;
        case GREEN: 
            color[0] = 0.0; color[1] = 1.0;
            color[2] = 0.0; color[3] = 1.0; break;
        case BLUE: 
            color[0] = 0.0; color[1] = 0.0;
            color[2] = 1.0; color[3] = 1.0; break;
        case WHITE: 
            color[0] = 1.0; color[1] = 1.0;
            color[2] = 1.0; color[3] = 1.0; break;
        case YELLOW: 
            color[0] = 1.0; color[1] = 1.0;
            color[2] = 0.0; color[3] = 1.0; break;
    }
    // draw the cube 
    ...
}

void display(void) {
    cube();
}

void options_menu(int input) {
    colorName = input;
    glutPostRedisplay();
}

void main(int argc, char** argv) {
    ... 
    glutCreateMenu(options_menu); // create options menu
    glutAddMenuEntry("Red", RED); // 1 add menu entries
    glutAddMenuEntry("Green", GREEN); // 2
    glutAddMenuEntry("Blue", BLUE); // 3
    glutAddMenuEntry("White", WHITE); // 4
    glutAddMenuEntry("Yellow", YELLOW); // 5
    glutAttachMenu(GLUT_RIGHT_BUTTON, "Colors");
    myinit();
    glutMainLoop();
}
Mouse callbacks for mouse motion

This example shows the callbacks for the mouse click and mouse motion events. These events can be used for anything that uses the position of a moving mouse with button pressed as control. It is fairly common to see a graphics program that lets the user hold down the mouse and drag the cursor around in the window, and the program responds by moving or rotating the scene around the window. This is done by noting the position where the mouse button is first pressed, and then keeping track of the distance between the current mouse position and the original click position. The program that is the source for this code fragment uses the integer coordinates spinX and spinY to control spin, but the coordinates developed like this could be used for many purposes and the application code itself is omitted.

```c
void Mouse(int button, int state, int mouseX, int mouseY) {
    curX = mouseX;
    curY = mouseY;
}

void motion(int xPos, int yPos) {
    spinX = (GLfloat)xPos - curX;
    spinY = (GLfloat)yPos - curY;
    myX = curX;
    myY = curY;
    glutPostRedisplay();
}

int main(int argc, char** argv) {
    ...
    glutMouseFunc(mouse);
    glutMotionFunc(motion);

    myinit();
    glutMainLoop();
}
```

Mouse callback for object picking

This example is more complex than any of the previous examples because the use of a mouse event in object selection involves several complex steps. We begin here with a simple set of code, creating two cubes with the familiar `cube()` function, and we will select one with the mouse. When we select one of the cubes, the cubes will exchange colors.

In this code example, we start with a full `Mouse(...)` callback function, the `render(...)` function that registers the two cubes in the object name list, and the `DoSelect(...)` function that manages drawing the scene in GL_SELECT mode and identifying the object(s) selected by the position of the mouse when the event happened. We do not include the `main()` function that includes the mouse callback function registration, because this was included in the example above. A more general version of the `DoSelect()` function is presented later in this chapter.

```c
glutMouseFunc(Mouse);
...
```
void Mouse(int button, int state, int mouseX, int mouseY) {
    if (state == GLUT_DOWN) { /* find which object was selected */
        hit = DoSelect((GLint) mouseX, (GLint) mouseY);
    }
    glutPostRedisplay();
}

...

void render( GLenum mode ) {
    // Always draw the two cubes, even if we are in GL_SELECT mode,
    // because an object is selectable iff it is identified in the
    // name list and is drawn in GL_SELECT mode
    if (mode == GL_SELECT)
        glLoadName(0);
    glPushMatrix();
    glTranslatef( 1.0, 1.0, -2.0 );
    cube(cubeColor2);
    glPopMatrix();
    if (mode == GL_SELECT)
        glLoadName(1);
    glPushMatrix();
    glTranslatef(-1.0, -2.0, 1.0 );
    cube(cubeColor1);
    glPopMatrix();
    glFlush();
    glutSwapBuffers();
}

...

GLuint DoSelect(GLint x, GLint y) {
    GLint hits, temp;

    glSelectBuffer(MAXHITS, selectBuf);
    glRenderMode(GL_SELECT);
    glInitNames();
    glPushMatrix();
    // set up the viewing model
    glPushMatrix();
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    // set up the matrix that identifies the picked object(s), based
    // on the x and y values of the selection and the information
    // on the viewport
    gluPickMatrix(x, windH - y, 4, 4, vp);
    glClearColor(0.0, 0.0, 1.0, 0.0);
    gluPerspective(60.0,1.0,1.0,30.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    // eye point center of view up
    gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    render(GL_SELECT); // draw the scene for selection
    glPopMatrix();
}
// find the number of hits recorded and reset mode of render
hits = glRenderMode(GL_RENDER);
// reset viewing model into GL_MODELVIEW mode
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0, 1.0, 1.0, 30.0);
gluPerspective(GL_MODELVIEW);
glLoadIdentity();
// eye point     center of view     up
gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
// return the label of the object selected, if any
if (hits <= 0) {
    return -1;
}
// carry out the color changes to be caused by a selection
temp = cubeColor1; cubeColor1 = cubeColor2; cubeColor2 = temp;
return selectBuf[3];

Details on picking

OpenGL has several ways to identify objects that correspond to mouse events, and we will discuss two of them in this chapter. One of these involves drawing invisibly and keeping track of objects that include a given pixel or small region around a pixel; this is what we will call the standard selection approach. The other involves drawing with synthetic colors that are unique to each object and looking at the color of the selected pixel in the color buffer to identify the nearest object at the picked point. We will discuss the standard selection approach first.

The standard selection approach calls for the mouse event to request that you render your scene invisibly so a record may be made of all the objects that are selected. This approach introduces the concept of the render mode for drawing. In your standard rendering, you draw the scene in GL_RENDER mode, which is the default drawing mode. In the mouse event callback that is executed after the mouse event, you change your rendering to GL_SELECT mode and re-draw the scene with each item of interest given a unique name. When the scene is rendered in GL_SELECT mode, nothing is actually changed in the frame buffer but the pixels that would be rendered are identified. When any named object is found that would include the pixel selected by the mouse, that object’s name is added to a selection buffer data structure, actually a stack of unsigned integers, that is maintained for that name. This selection buffer holds information on all the items in a hierarchy of named items that were hit. When the rendering of the scene in GL_SELECT mode is finished, a list of hit records is produced, with one entry for each name of an object whose rendering included the mouse click point, and the number of such records is returned when the system is returned to GL_RENDER mode. The structure of these hit records is described below. You can then process this list to identify the items that were hit, including the distance from the eye where the hit occurred, and you can proceed to do whatever work you need with this information.

The concept of “item of interest” is more complex than is immediately apparent. It can include a single object, a set of objects, or even a hierarchy of objects. Think creatively about your problem and you may be surprised just how powerful this kind of selection can be.

Definitions

The first concept we must deal with for object selection is the notion of a selection buffer. This is an array of unsigned integers (GLuint selectBuf[SIZE]) that will hold the array of hit records for a mouse click. In turn, a hit record contains several items as illustrated in Figure 11.2. These include the number of items that were on the name stack, the nearest (zmin) and farthest
(z\text{max}) distances to objects on the stack, and the list of names on the name stack for the selection. The distances are integers because they are taken from the Z-buffer, where you may recall that distances are stored as integers in order to make comparisons more effective. The name stack contains the names of all the objects in a hierarchy of named objects that were selected with the mouse click.

The distance to objects is given in terms of the viewing projection environment, in which the nearest points have the smallest non-negative values because this environment has the eye at the origin and distances increase as points move away from the eye. Typical processing will examine each selection record to find the record with the smallest value of z\text{min} and will work with the names in that hit record to carry out the work needed by that hit. This work is fairly typical of the handling of any list of variable-length records, proceeding by accumulating the starting points of the individual records (starting with 0 and proceeding by adding the values of (n\text{items}+3) from the individual records), with the z\text{min} values being offset by 1 from this base and the list of names being offset by 3. This is not particularly difficult, but it does require that the programmer exercise care because of the variable size of the record for a single hit.

In OpenGL, we have two ways to pick an object. We can generate the scene (partly or fully) and record the intersection of the object with the pixel identified by the mouse, or we can set up an additional projection with the pick matrix and clip everything that is not within a certain distance of the pixel we chose. The first of these methods is perhaps simpler because it does not involve any changes in the basic rendering, but the second is faster because it uses clipping to avoid rendering anything except those items very near the pixel. We will begin by discussing the simpler case, and describe the use of the pick matrix later.

Before we go on, however, we must be sure you understand one key point: you don’t do anything more to generate the selection buffer than to draw the scene in select mode. That’s it. The system does all the rest of the work for you, putting the active name (or all the names on the active name stack) into the selection buffer whenever the drawing process operates on the selected pixel or the object is not clipped away by the pick matrix. You simply generate the scene in select mode, get the number of hits from the returned value of the glRenderMode(\ldots) function when you call it to restore GL\_RENDER mode, and process the selection buffer to handle that many hits.
Making picking work

The picking process is fairly straightforward. The function `glRenderMode(mode)` allows you to draw in either of two modes: render mode (GL_RENDER) invokes the graphics pipeline and produces pixels in the frame buffer, and select mode (GL_SELECT) calculates the pixels that would be drawn if the graphics pipeline were to be invoked, and tests the pixels against the pixels that were identified by the mouse click. As illustrated in the example below, the mouse function can be defined to change the drawing mode to GL_SELECT and to post a redisplay operation. The display function can then draw the scene in select mode with selection object names defined with `glutLoadName(int)` to determine what name will be put into the selection buffer if the object includes the selected pixel, noting that the mode can be checked to decide what is to be drawn and/or how it is to be drawn, and then the selection buffer can be examined to identify what was hit so the appropriate processing can be done. After the selection buffer is processed, the scene can be displayed again in render mode to present the effect of the selection.

In the outline above, it sounds as though the drawing in select mode will be the same as in render mode. But this is often not the case; there are some techniques to make the select mode drawing work more quickly or more flexibly than render-mode drawing. These include:

• if there are any objects that you don’t want the user to be able to select, do not draw these at all in select mode. Because they are not drawn, they are invisible to picking.
• if you want to allow a pick of a complex object, you need not do all the work of a full rendering of the object in select mode; you need only design an approximation of the object and draw that.
• you can even create invisible controls by allowing the user to pick things that are only drawn in select mode but not in render mode.

Think creatively and you can find that you can do interesting things with selection. In fact, you will sometimes find that you must use some of these techniques. For example, if you want the user to be able to select a wireframe object, you probably want to replace the wireframe version by a solid version in the select mode drawing, because a user will visualize the spaces in the wireframe as part of the object but OpenGL will not. Another important use is in selecting text, because you cannot pick raster characters in OpenGL. For whatever reason, if you draw any raster characters in select mode, OpenGL will always think that the characters were picked no matter where you clicked. If you want to be able to pick a word that is drawn as raster characters, create a rectangle that occupies the space where the raster characters would be, and draw that rectangle in select mode.

It’s worth a bit of thinking about the concept of selection names. A name is simply an integer that is used to identify a point in the modeling process; these integers are often given symbolic names through `#define` statements. You cannot load a new name inside a region of code delimited by a `glBegin(mode)`–`glEnd()` pair, so if you use any geometry compression in your object, all of the compression must all be within a single named object. You can, however, nest names with the name stack, using the `glPushMatrix(int)` function so that while the original name is active, the new name is also active. For example, suppose we were dealing with automobiles, and suppose that we wanted someone to select parts for an automobile. We could permit the user to select parts at a number of levels; for example, to select an entire automobile, the body of the automobile, or simply one of the tires. In the sample code fragment below, we create a hierarchy of selections for an automobile (“Jaguar”) and for various parts of the auto (“body”, “tire”, etc.) In this case, the names JAGUAR, BODY, FRONT_LEFT_TIRE, and FRONT_RIGHT_TIRE are symbolic names for integers that are defined elsewhere in the code, as suggested above.

```c
    glLoadName( JAGUAR );
    glPushMatrix( BODY );
        glCallList( JagBodyList );
    glPopMatrix();
```

2/12/03
The use of the name stack in developing the objects to be selected is quite similar to the use of the transformation stack in modeling. Object names can be added to the information in the scene graph so that whenever a name is passed in going down a branch of the scene graph, the name is pushed onto the name stack, and whenever a name is passed coming back up a branch, the name stack is popped. This allows you to include the name stack as part of the modeling and make the process more systematic.

When a selection occurs, then, the selection buffer will include everything whose display involved the pixel that was chosen, including the automobile as well as the lower-level part. For example, if you selected the right front tire of the automobile, nitems would be 3 and your hit record would include three names: the names FRONT_LEFT_TIRE, BODY, and JAGUAR. Your program then would know that a hierarchy was selected that had these three parts and could choose (or allow the user to choose) which selection or what other logic it needed to use.

There are a couple of things to watch out for in the name stack. The first is that the name stack is empty when it is initialized, so you cannot simply load a name into the stack; this will generate an error. Instead you must push some name onto the stack so that you can load a name to replace it. The second thing to watch for with the name stack is that loading a name only replaces the top name on the name stack. If you have finished a hierarchy and need to remove the entire hierarchy from the name stack, you will need to pop the name stack until there is a single name left; you can then load the new name and replace that single name. But these are straightforward to remember as you use the name stack.

The pick matrix

Picking using the pick matrix is almost the same operation, logically, as picking using the selected pixel, but we present it separately because it uses a different process and allows us to define a concept of ‘near’ and to talk about a way to identify the objects near the selection point.

In the picking process, you can define a very small window in the immediate neighborhood of the point where the mouse was clicked, and then you can identify everything that is drawn in that neighborhood. The result is returned in the same selection buffer and can be processed in the same way. This is done by creating a transformation with the function gluPickMatrix (...) that is applied after the projection transformation (that is, defined before the projection; recall the relation between the sequence in which transformations are identified and the sequence in which they are applied). The full function call is

\[
gluPickMatrix(GLdouble x, GLdouble y, GLdouble width, GLdouble height, GLint viewport[4])
\]

where \( x \) and \( y \) are the coordinates of the point picked by the mouse, which is the center of the picking region; the width and height are the size of the picking region in pixels, sometimes called...
the pick tolerance; and the viewport is the vector of four integers returned by the function call 
`glGetIntegerv(GL_VIEWPORT, GLint *viewport)`.

The function of this pick matrix is to identify a small region centered at the point where the mouse 
was clicked and to select anything that is drawn in that region. The picking process returns a 
standard selection buffer that can then be processed to identify the objects that were picked, as 
described above.

A code fragment to implement this picking is given below. This corresponds to the point in the 
code for `doSelect(...)` above labeled “set up the standard viewing model” and “standard 
perspective viewing”:

```c
int viewport[4]; /* place to retrieve the viewport numbers */
...
dx = glutGet( GLUT_WINDOW_WIDTH );
dy = glutGet( GLUT_WINDOW_HEIGHT );
...
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
if( RenderMode == GL_SELECT ) {
    glGetIntegerv( GL_VIEWPORT, viewport );
    gluPickMatrix( (double)Xmouse, (double)(dy - Ymouse),
                   PICK_TOL, PICK_TOL, viewport );
}
... call glOrtho(), glFrustum(), or gluPerspective() here
```

Using the back color buffer to do picking

There is another approach you can use that avoids the selection buffer entirely by using some of the 
facilities of double-buffered drawing. In this approach, when you want to permit a selection, you 
continue to draw your scene in render mode but you draw your scene into the back buffer in a 
unique way. Here you can identify the objects you want to make selectable by giving each a 
unique color, you can use a proxy for an object so that you draw the alternate representation we 
discussed above, and you can omit any objects you don’t want to draw. When the mouse event 
occurs and the mouse callback gets the pixel location of the pick, you look in the back buffer to 
see what color is at that pixel location. That color can be used to identify the object drawn at that 
position, giving you your picked object. If you were drawing with depth test enabled, you will get 
the object that is closest to the eye at that point. If you were not, you will get the last object drawn 
to that point. But you can control which of these you want to happen. And after you have gotten 
the information you need from the back buffer, you simply don’t swap it with the front buffer but 
let the next (and probably normal) drawing operation replace the artificial-color image you have just 
created.

The mechanics of this are pretty straightforward. After the back buffer has been filled with the 
artificial image, select the back buffer to be read with the `glReadBuffer(GL_BACK)` function 
(although the back buffer is the default buffer for reading in double-buffered mode). Then use the 
`glReadPixels(...)` function to read a 1x1 array of color pixels (that is, the value of the color 
at a single point) at the position of the selection, and use whatever logic you created in determining 
the color of each object to identify the object you have just selected. This is a straightforward 
technique but it may require some thought to make a reasonable set of color identifications if you 
have a large number of objects.
A selection example

The selection process is well illustrated by an interactive shape selection program by a student, Ben Eadington. This code sets up and renders a Bézier spline surface with a set of selectable control points. When an individual control point is selected, that point can be moved with a keyboard event callback function and the surface responds to the adjusted set of points. A sample image from this work is given in Figure 11.3, with one control point selected (at the left and toward the front, shown as a red cube instead of the default green color).

Selected code fragments from this project are given below. Here all the data declarations and evaluator work are omitted, as are some standard parts of the functions that are presented, and just the important functions are given with the key points described in these notes. You will be directed to several specific points in the code to illustrate how selection works, described with interspersed text as the functions or code are presented.

![Image of a surface with selectable control points and with one selected](image.png)

Figure 11.3: a surface with selectable control points and with one selected

In the first few lines you will see the declaration of the global selection buffer that will hold up to 200 values. This is quite large for this particular problem, since there are no hierarchical models and no more than a very few control points could ever line up. The actual size needed would probably be no more than four GLuints per control point selected, and probably no more than 10 maximum points would ever line up in this problem. Hence the selection buffer could probably be defined to hold only 40 or 50 values. Each individual problem will need a similar analysis.

```c
// globals initialization section
#define MAXHITS 200 // number of GLuints in hit records
// data structures for selection process
GLuint selectBuf[MAXHITS];
```

The next point is the mouse callback. This simply catches a mouse-button-down event and calls the DoSelect function, listed and discussed below, to handle the mouse selection. When the hit is handled (including the possibility that there was no hit with the cursor position) the control is passed back to the regular processes with a redisplay.

```c
// mouse callback for selection
void Mouse(int button, int state, int mouseX, int mouseY)
{
    if (state == GLUT_DOWN) { // find which object was selected
        hit = DoSelect((GLint) mouseX, (GLint) mouseY);
    }
    glutPostRedisplay(); /* redraw display */
}
```
The control points may be drawn in either GL_RENDER or GL_SELECT mode, so this function must handle both cases. The only difference is that names must be loaded for each control point, and if any of the points had been hit previously, it must be identified so it can be drawn in red instead of in green. But there is nothing in this function that says what is or is not hit in another mouse click; this is handled in the DoSelect function below.

```c
void drawpoints(GLenum mode)
{
    int i, j;
    int name=0;
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, green);
    // iterate through control point array
    for(i=0; i<GRIDSIZE; i++)
        for(j=0; j<GRIDSIZE; j++) {
            if (mode == GL_SELECT) {
                glLoadName(name); // assign a name to each point
                name++;    // increment name number
            }
            glPushMatrix();
            ... place point in right place with right scaling
            if(hit==i*16+j%16) { // selected point, need to draw it red
                glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, red);
                glutSolidCube(0.25);
                glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, green);
            }
            else glutSolidCube(0.25);
            glPopMatrix();
        }
}
```

The only real issue here is to decide what you do and do not need to draw in each of the two rendering modes. Note that the surface is only drawn if the program is in GL_RENDER mode; because nothing in the surface is itself selectable, the only thing that needs to be drawn in GL_SELECT mode is the control points.

```c
void render(GLenum mode) {
    ... do appropriate transformations
    if (mode == GL_RENDER) { // don't render surface if mode is GL_SELECT
        surface(ctrlpts);
        ... some other operations that don't matter here
    }
    if(points) drawpoints(mode); // always render the control points
    ... pop the transform stack as needed and exit gracefully
}
```

This final function is the real meat of the problem. The display environment is set up (projection and viewing transformations), the glRenderMode function sets the rendering mode to GL_SELECT and the image is drawn in that mode, the number of hits is returned from the call to the glRenderMode function when it returns to GL_RENDER mode, the display environment is rebuilt for the next drawing, and the selection buffer is scanned to find the object with the smallest zmin value as the selected item. That value is then returned so that the drawpoints function will know which control point to display in red and so other functions will know which control point to adjust.
A summary of picking

This might be a good place to summarize the things we’ve seen about the standard picking process in the discussions and code examples above:

- Define an array of unsigned integers to act as the selection buffer
- Design a mouse event callback that calls a function that does the following:
  - Sets GL_SELECT mode and draws selected parts of the image, having loaded names so these parts can be identified when the selection is made
  - when this rendering is completed, returns a selection buffer that can be processed
  - returns to GL_RENDER mode.
- Be careful to manage your name stack as you draw objects in GL_SELECT mode so that you have exactly the things on the stack you need to identify the objects you want the user to be able to pick.

This design structure is straightforward to understand and can be easily implemented with a little care and planning.
The MUI (Micro User Interface) facility

Graphics APIs may include some basic interaction capabilities, as we saw earlier in this chapter, but they rarely offer a broad set of tools to build full user interfaces. Such work might require you to build a set of graphics objects such as buttons, sliders, dials, or the like, and to write your own functions to provide program control by interacting with these objects. This can be hard work and that work might have to be repeated for each program that needed an interface.

Instead of taking this approach, many people have built standardized interface toolkits for graphics APIs. These are functions such as those noted above that allow you to add interface capabilities to your programs simply by including appropriate headers and linking with appropriate libraries. In this section we introduce a very simple interface facility that is commonly available with OpenGL so you can get some experience with programming an interface. As we said at the first of this chapter, we do not believe that this will make you an effective interface designer, but it may help you understand what is involved with implementing an interface for a program using a standard toolkit.

To understand this section and be able to take advantage of an interaction toolkit for a graphics API, you should have an understanding of event-driven programming and some experience using the simple events and callbacks from the GLUT toolkit in OpenGL. You should also review the interface capabilities of a number of standard applications.

Introduction

There are many kinds of interface tools that we are used to seeing in applications but that we cannot readily code in OpenGL, even with the GLUT toolkit. Some of these are provided by the MUI facility that is a universal extension of GLUT for OpenGL. With MUI you can use sliders, buttons, text boxes, and other tools that may be more natural for many applications than the standard GLUT capabilities. Of course, you may choose to write your own tools as well, but you may choose to use your time on the problem at hand instead of writing an interface, so the MUI tools may be just what you want.

MUI has a good bit of the look and feel of the X-Motif interface, so do not expect applications you write with this to look like they are from either the Windows or Macintosh world. Instead, focus on the functionality you need your application to have, and find a way to get this functionality from the MUI tools. The visible representation of these tools are called widgets, just as they are in the X Window System, so you will see this term throughout these notes.

This chapter is built on Steve Baker’s “A Brief MUI User Guide,” and it shares similar properties: it is based on a small number of examples and some modest experimental work. It is intended as a guide, not as a manual, though it is hoped that it will contribute to the literature on this useful tool.

Definitions

The capabilities of MUI include pulldown menus, buttons, radio buttons, text labels, text boxes, and vertical and horizontal sliders. We will outline how each of these work below and will include some general code to show how each is invoked.

The main thing you must realize in working with MUI is that MUI takes over the event handling from GLUT, so you cannot mix MUI and GLUT event-handling capabilities in the same window. This means that you will have to create separate windows for your MUI controls and for your display, which can feel somewhat clumsy. This is a tradeoff you must make when you design your application — are you willing to create a different kind of interface than you might expect in a
traditional application in order to use the extra MUI functionality? Only you can say. But before you can make that choice, you need to know what each of the MUI facilities can do.

Menu bars: A MUI menu bar is essentially a GLUT menu that is bound to a MUI object and then that object is added to a UIlist. Assuming you have defined an array of GLUT menus named myMenus[...], you can use the function to create a new pulldown menu and then use the function to add new menus to the pulldown menu list:

```c
muiObject *muiNewPulldown();
muiAddPulldownEntry(muiObject *obj, char *title, int glut_menu,
                   int is_help);
```

An example of the latter function would be

```c
myMenubar = muiNewPulldown();
muiAddPulldownEntry(myMenubar, "File", myMenu, 0);
```

where the is_help value would be 1 for the last menu in the menu bar, because traditionally the help menu is the rightmost menu in a menu bar.

According to Baker [Bak], there is apparently a problem with the pulldown menus when the GLUT window is moved or resized. The reader is cautioned to be careful in handling windows when the MUI facility is being used.

Buttons: a button is presented as a rectangular region which, when pressed, sets a value or carries out a particular operation. Whenever the cursor is in the region, the button is highlighted to show that it is then selectable. A button is created by the function

```c
muiNewButton(int xmin, int xmax, int ymin, int ymax)
```

that has a muiObject * return value. The parameters define the rectangle for the button and are defined in window (pixel) coordinates, with (0, 0) at the lower left corner of the window. In general, any layout in the MUI window will be based on such coordinates.

Radio buttons: radio buttons are similar to standard buttons, but they come in only two fixed sizes (either a standard size or a mini size). The buttons can be designed so that more than one can be pressed (to allow a user to select any subset of a set of options) or they can be linked so that when one is pressed, all the others are un-pressed (to allow a user to select only one of a set of options). Like regular buttons, they are highlighted when the cursor is scrolled over them.

You create radio buttons with the functions

```c
muiObject *muiNewRadioButton(int xmin, int ymin)
muiObject *muiNewTinyRadioButton(int xmin, int ymin)
```

where the xmin and ymin are the window coordinates of the lower left corner of the button. The buttons are linked with the function

```c
void muiLinkButtons(button1, button2)
```

where button1 and button2 are the names of the button objects; to link more buttons, call the function with overlapping pairs of button names as shown in the example below. In order to clear all the buttons in a group, call the function below with any of the buttons as a parameter:

```c
void muiClearRadio(muiObject *button)
```

Text boxes: a text box is a facility to allow a user to enter text to the program. The text can then be used in any way the application wishes. The text box has some limitations; for example, you cannot enter a string longer than the text box’s length. However, it also gives your user the ability to enter text and use backspace or delete to correct errors. A text box is created with the function

```c
muiObject *muiNewTextbox(xmin, xmax, ymin)
```

whose parameters are window coordinates, and there are functions to set the string:

```c
muiSetTBString(obj, string)
```

to clear the string:

```c
muiClearTBString(obj)
```

and to get the value of the string:
char *muiGetTBString (muiObject *obj).

Horizontal sliders: in general, sliders are widgets that return a single value when they are used. The value is between zero and one, and you must manipulate that value into whatever range your application needs. A slider is created by the function

```
muiNewHSlider(int xmin,int ymin,int xmax,int scenter,int shalf)
```

where xmin and ymin are the screen coordinates of the lower left corner of the slider, xmax is the screen coordinate of the right-hand side of the slider, scenter is the screen coordinate of the center of the slider's middle bar, and shalf is the half-size of the middle bar itself. In the event callback for the slider, the function muiGetHSVVal (muiObject *obj) is used to return the value (as a float) from the slider to be used in the application. In order to reverse the process — to make the slider represent a particular value, use the function

```
muiSetHSVValue(muiObject *obj, float value)
```

Vertical sliders: vertical sliders have the same functionality as horizontal sliders, but they are aligned vertically in the control window instead of horizontally. They are managed by functions that are almost identical to those of horizontal sliders:

```
muiNewVSlider(int xmin,int ymin,int ymax,int scenter,int shalf)
muiGetVSVValue(muiObject *obj, float value)
muiSetVSVValue(muiObject *obj, float value)
```

Text labels: a text label is a piece of text on the MUI control window. This allows the program to communicate with the user, and can be either a fixed or variable string. To set a fixed string, use

```
muiNewLabel(int xmin, int ymin, string)
```

with xmin and ymin setting the lower left corner of the space where the string will be displayed. To define a variable string, you give the string a muiObject name via the variation

```
muiObject *muiNewLabel(int xmin, int ymin, string)
```

to attach a name to the label, and use the muiChangeLabel(muiObject *, string) function to change the value of the string in the label.

Using the MUI functionality

Before you can use any of MUI's capabilities, you must initialize the MUI system with the function muiInit(), probably called from the main() function as described in the sample code below.

MUI widgets are managed in UI lists. You create a UI list with the muiNewUIList (int) function, giving it an integer name with the parameter, and add widgets to it as you wish with the function muiAddToUIList(listid, object). You may create multiple lists and can choose which list will be active, allowing you to make your interface context sensitive. However, UI lists are essentially static, not dynamic, because you cannot remove items from a list or delete a list.

All MUI capabilities can be made visible or invisible, active or inactive, or enabled or disabled. This adds some flexibility to your program by letting you customize the interface based on a particular context in the program. The functions for this are:

```
void muiSetVisible(muiObject *obj, int state);
void muiSetActive(muiObject *obj, int state);
void muiSetEnable(muiObject *obj, int state);
int muiGetVisible(muiObject *obj);
int muiGetActive(muiObject *obj);
int muiGetEnable(muiObject *obj);
```

Figure 11.4 shows most of the MUI capabilities: labels, horizontal and vertical sliders, regular and radio buttons (one radio button is selected and the button is highlighted by the cursor as shown),
and a text box. Some text has been written into the text box. This gives you an idea of what the standard MUI widgets look like, but because the MUI source is available, you have the opportunity to customize the widgets if you want, though this is beyond the scope of this discussion. Layout is facilitated by the ability to get the size of a MUI object with the function

```c
void muiGetObjectSize(muiObject *obj, int *xmin, int *ymin, int *xmax, int *ymax);
```

MUI object callbacks are optional (you would probably not want to register a callback for a fixed text string, for example, but you would with an active item such as a button). In order to register a callback, you must name the object when it is created and must link that object to its callback function with

```c
void muiSetCallback(muiObject *obj, callbackFn)
```

where a callback function has the structure

```c
void callbackFn(muiObject *obj, enum muiReturnValue)
```

Note that this callback function need not be unique to the object; in the example below we define a single callback function that is registered for three different sliders and another to handle three different radio buttons, because the action we need from each is the same; when we need to know which object handled the event, this information is available to us as the first parameter of the callback.

![Figure 11.4: the set of MUI facilities on a single window](image)

If you want to work with the callback return value, the declaration of the `muiReturnValue` is:

```c
enum muiReturnValue {  
    MUI_NO_ACTION,  
    MUI_SLIDER_MOVE,  
    MUI_SLIDER_RETURN,  
    MUI_SLIDER_SCROLLDOWN,  
    MUI_SLIDER_SCROLLUP,  
    MUI_SLIDER_THUMB,  
    MUI_BUTTON_PRESS,  
    MUI_TEXTBOX_RETURN,  
    MUI_TEXTLIST_RETURN,  
    MUI_TEXTLIST_RETURN_CONFIRM
};
```

so you can look at these values explicitly. For the example below, the button press is assumed because it is the only return value associated with a button, and the slider is queried for its value instead of handling the actual MUI action.

**Some examples**

Let’s consider a simple application and see how we can create the controls for it using the MUI facility. The application is color choice, commonly handled with three sliders (for R/G/B) or four sliders (for R/G/B/A) depending on the need of the user. This application typically provides a way to display the color that is chosen in a region large enough to reduce the interference of nearby colors in perceiving the chosen color. The application we have in mind is a variant on this that not only shows the color but also shows the three fixed-component planes in the RGB cube and draws a sphere of the selected color (with lighting) in the cube.

2/12/03
The design of this application is built on an example in the Science Examples chapter that shows three cross-sections of a real function of three variables. In order to determine the position of the cross sections, we use a control built on MUI sliders. We also add radio buttons to allow the user to define the size of the sphere at the intersection of the cross-section slices.

Selected code for this application includes declarations of muiObjects, callback functions for sliders and buttons, and the code in the main program that defines the MUI objects for the program, links them to their callback functions, and adds them to the single MUI list we identify. The main issue is that MUI callbacks, like the GLUT callbacks we met earlier, have few parameters and do most of their work by modifying global variables that are used in the other modeling and rendering operations.

```c
// selected declarations of muiObjects and window identifiers
muiObject *Rslider, *Gslider, *Bslider;
muiObject *Rlabel, *Glabel, *Blabel;
muiObject *noSphereB, *smallSphereB, *largeSphereB;
int muiWin, glWin;

// callbacks for buttons and sliders
void readButton(muiObject *obj, enum muiReturnValue rv) {
    if ( obj == noSphereB )
        sphereControl = 0;
    if ( obj == smallSphereB )
        sphereControl = 1;
    if ( obj == largeSphereB )
        sphereControl = 2;
    glutSetWindow( glWin );
    glutPostRedisplay();
}

void readSliders(muiObject *obj, enum muiReturnValue rv) {
    char rs[32], gs[32], bs[32];
    glutPostRedisplay();
    rr = muiGetHSVal(Rslider);
    gg = muiGetHSVal(Gslider);
    bb = muiGetHSVal(Bslider);
    sprintf(rs,"%6.2f",rr);
    muiChangeLabel(Rlabel, rs);
    sprintf(gs,"%6.2f",gg);
    muiChangeLabel(Glabel, gs);
    sprintf(bs,"%6.2f",bb);
    muiChangeLabel(Blabel, bs);
    DX = -4.0 + rr*8.0;
    DY = -4.0 + gg*8.0;
    DZ = -4.0 + bb*8.0;
    glutSetWindow(glWin);
    glutPostRedisplay();
}

void main(int argc, char** argv){
    char rs[32], gs[32], bs[32];
    // Create MUI control window and its callbacks
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGBA);
```
glutInitWindowSize(270,350);
    glutInitWindowPosition(600,70);
    muiWin = glutCreateWindow("Control Panel");
    glutSetWindow(muiWin);
    muiInit();
    muiNewUIList(1);
    muiSetActiveUIList(1);

    // Define color control sliders
    muiNewLabel(90, 330, "Color controls");

    muiNewLabel(5, 310, "Red");
    sprintf(rs, "%6.2f", rr);
    Rlabel = muiNewLabel(35, 310, rs);
    Rslider = muiNewHSlider(5, 280, 265, 130, 10);
    muiSetCallback(Rslider, readSliders);

    muiNewLabel(5, 255, "Green");
    sprintf(gs, "%6.2f", gg);
    Glabel = muiNewLabel(35, 255, gs);
    Gslider = muiNewHSlider(5, 225, 265, 130, 10);
    muiSetCallback(Gslider, readSliders);

    muiNewLabel(5, 205, "Blue");
    sprintf(bs, "%6.2f", bb);
    Blabel = muiNewLabel(35, 205, bs);
    Bslider = muiNewHSlider(5, 175, 265, 130, 10);
    muiSetCallback(Bslider, readSliders);

    // define radio buttons
    muiNewLabel(100, 150, "Sphere size");
    noSphereB = muiNewRadioButton(10, 110);
    smallSphereB = muiNewRadioButton(100, 110);
    largeSphereB = muiNewRadioButton(190, 110);
    muiLinkButtons(noSphereB, smallSphereB);
    muiLinkButtons(smallSphereB, largeSphereB);
    muiLoadButton(noSphereB, "None");
    muiLoadButton(smallSphereB, "Small");
    muiLoadButton(largeSphereB, "Large");
    muiSetCallback(noSphereB, readButton);
    muiSetCallback(smallSphereB, readButton);
    muiSetCallback(largeSphereB, readButton);
    muiClearRadio(noSphereB);

    // add sliders and radio buttons to UI list 1
    muiAddToUIList(1, Rslider);
    muiAddToUIList(1, Gslider);
    muiAddToUIList(1, Bslider);
    muiAddToUIList(1, noSphereB);
    muiAddToUIList(1, smallSphereB);
    muiAddToUIList(1, largeSphereB);

    // Create display window and its callbacks
    ...
}

The presentation and communication for this application are shown in Figure 11.5 below. As the
sliders set the R, G, and B values for the color, the numerical values are shown above the sliders
and the three planes of constant R, G, and B are shown in the RGB cube. At the intersection of
the three planes is drawn a sphere of the selected color in the size indicated by the radio buttons. The RGB cube itself can be rotated by the usual keyboard controls so the user can compare the selected color with nearby colors in those planes, but you have the usual issues of active windows: you must make the display window active to rotate the cube, but you must make the control window active to use the controls.

![Interactive color picker in RGB cube](image)

Figure 11.5: the color selector in context, with both the display and control windows shown

**Installing MUI for Windows systems**

MUI comes with the GLUT release, so if you have GLUT on your system you probably also have MUI. But if you do not have GLUT, when you download and uncompress the GLUT release you will have several header files (in the include/mui directory) and a couple of libraries: libmui.a for Unix and mui.lib for Windows. Install these in the usual places; for Windows, install mui.lib in the

\<drive>:\Program Files\Microsoft Visual Studio\VC98\Lib\directory. Place the header files also in the usual place; for Windows use

\<drive>:\Program Files\Microsoft Visual Studio\VC98\include\Then simply add mui.lib to your project files and you should be able to use MUI successfully.

**A word to the wise...**

The MUI control window has behaviors that are outside the programmer’s control, so you must be aware of some of these in order to avoid some surprises. The primary behavior to watch for is that many of the MUI elements include a stream of events (and their associated redispays) whenever the cursor is within the element’s region of the window. If your application was not careful to insulate itself against changes caused by redispays, you may suddenly find the application window showing changes when you are not aware of requesting them or of creating any events at all. So if you use MUI, you should be particularly conscious of the structure of your application.
on redisplay and ensure that (for example) you clear any global variable that causes changes in
your display before you leave the display function.

**Summary**

This chapter has outlined the standard kinds of events dealt with in a graphics system and the kinds
of callbacks they use, and has given several examples of programming with these events. It has
also discussed a user interface toolkit, both for its own value and as an example of the kind of
interaction toolkits that might be found with a graphics API. With these tools, you should be able
to extend your graphics programming abilities with a broad set of interaction tools that can be used
whenever your users should be able to interact with your images.

**Questions**

1. In the chapter we alluded to 4DF and 6DF controls (DF = degree of freedom). However, we
also have 2DF controls; discuss how the slider is a 2DF control and describe how you could
create a 6DF control from three sliders (or other 2DF controls). Why is this or is this not a
good way to create a 6DF control? Could you use mouse motion for 4DF and some keyboard
control for the other 2DF?

2. When you use a mouse motion control for a rotation, it is important that the direction of the
mouse be the direction of motion of the image. You can achieve image motion in two ways —
by moving the eye point around a fixed image, or by moving the image with a fixed eye point.
If we take increasing x- or y-coordinate values from the mouse to mean that the object moves to
the right or up, which of the two motions above will achieve that goal? Why?

3. Imagine, or lay out, some collection of objects in a space; an example might be the carousel
model you created in Chapter 2. Suppose you only want part of these objects to be selectable
(for example, the posts in the carousel, but not the carousel animals). Describe how you could
define your selection process to make this happen.

4. Sometimes a graphical object is made up of things that are difficult to select, such as line
segments. How would you lay out a “selection scene” that provides objects that represent the
difficult original objects so you could select them more easily?

5. Discuss the differences between picking and selection in terms of efficiency and of ease of
identifying objects of various sizes. What advantages or disadvantages does each approach
have?

6. Both selection and pick processes require you to analyze the selection buffer to identify which
objects are closest, farthest, or have another relationship to the eye point. Why does the back
buffer approach find the object nearest the eye without any further work? Is there any way that
the back buffer could find any other object?

**Exercises**

7. In many applications it is useful to be able to navigate around a scene by moving your eyepoint
in a 2D space within your scene and directing your view towards any point in the scene.
Devise a way to do this by using a diamond of keys on the keyboard controlling motion left,
right, forward, and back, and by using mouse motion to rotate the direction of view around the
current eyepoint. By a diamond of keys we are referring to a set of keys such as S-E-D-X (for
the left hand) or J-I-K-M (for the right hand).
8. Using examples from the science chapter or from your own work, identify a problem that uses animation, and build an idle callback that handles the necessary parameter changes to provide that animation for the image.

9. Using examples from the science chapter or from your own work, identify a problem that uses selection from a set of options, and build a menu interface that allows the user to make the necessary selection(s) for the problem.

10. Using examples from the science chapter or from your own work, identify a problem that uses selection from a set of options, and build a keyboard interface that allows the user to make the necessary selection(s) for the problem. Be careful that the keyboard options are understandable to the user.

11. Using examples from the science chapter or from your own work, identify a problem that uses selection from a set of options, and build a MUI button interface that allows the user to make the necessary selection(s) for the problem.

12. Using examples from the science chapter or from your own work, identify a problem that involves selecting a particular graphical object to be manipulated, and build a mouse selection operation that allows the user to select the graphical object for the problem.

13. Using examples from the science chapter or from your own work, identify a problem that needs a parameter or other value to be provided by the user, and build a MUI interface that allows the user to enter the value from either a text input window or a slider.

14. Examine the nature of hit records by modifying any program including selection to include code to dump the selection buffer byte-by-byte into a file, and examining that file by a simple file dump utility such as Unix’s od. Identify all the components of the selection buffer within this byte array and see how these components are really arranged.

15. Do the previous exercise when several objects are grouped in one name; when objects are arranged in a hierarchy. These should give more complex lists of the names on the name stack when the selection is made; break these down to understand how grouping and hierarchy work.

Experiments

16. (Class project) While a full user test of a program interface is well beyond the scope of this book, you can get an idea of how useful program controls are by getting a few of your friends to use your program and tell you about it. These friends can, in fact, be classmates. So choose a problem involving exploring a principle or set of data that is oriented to user interaction, and each person in the class should design and implement the interactions for the problem. Make everyone’s program available to the class, and each class member should run each program and write a short evaluation of the interaction. Gather the evaluations and find the program that works best, and discuss why this program’s interactions work.

17. Recall from Chapter 1 on projection that you calculated the parametric equation of a line segment in the viewing frustum that represents the points in 3D eye space that project to a single screen point. Define a number of simple sphere and polygon primitives that lie in the visible part of a space and choose a point on the front viewing plane. Calculate the intersection of the resulting line with each of the primitives and explore the way you could tell which is nearest the eye.
18. Experiment with the use of the back buffer for picking by setting your select-mode rendering to draw different objects in different colors and identify what object is nearest the eye at any given screen point.

19. As an experiment in different kinds of interaction, create a balance beam with an unknown weight at one end, and have the user add weight to the other end in order to balance the beam. Use different kinds of interaction to add the weight, and consider which kind of interaction is most effective based on either the time it takes to accomplish the task or the ease the user reports in accomplishing the task. Some possible interaction techniques are (a) use a dial or slider to adjust the weight to achieve the balance, (b) use picking to select standard weights to achieve the balance (think of a set of weights in a physics experiment), or (c) use the keyboard to increase or decrease the weight by unit amounts to achieve the balance.

20. (The scene graph) In order to be systematic about including event handling in the scene graph, you might want to include an “event node” that documents the event control over an aspect of the scene graph, such as transformations, appearance, or even geometry. Create such a new kind of node and modify the scene graph for an interactive graphics program by adding these nodes. It may be difficult to generate the code for the event handling from the scene graph, but see if you can do that.

Projects

21. (The small house) Build an interactive walkthrough of your small house by implementing the navigation scheme in exercise 7 above and showing the views of the house from successive eyepoints as they are selected.

22. (A scene graph parser) Add a name node to the scene graph, to be used in the same places as the transformation node. Add pushing and popping the name stack to the set of operations that are handled by the parser.