Chapter 10: Events and Event Handling for Computer Graphics

Introduction

Graphics programming can focus entirely on creating one single image based on a set of data, but more and more we are seeing the value of writing programs that allow the user to interact with the world through graphical presentations, or that allow the user to control the way an image is created. These are called interactive computer graphics programs, and the ability to interact with information through an image is critically important to the success of this field.

Our 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 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 and the chapter on selection describe how such implementation can be done.

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.

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.

<u>keypress events</u>, 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.

<u>mouse events</u>, 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.

<u>menu events</u>, such as selection of an item from a pop-up or pull-down menu or submenu, that are based on menu choices.

<u>window events</u>, 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

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.

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.

Event	Callback Registration Function

idle	`glutIdleFunc(functionname)`
	requires a callback function with template
	`void functionname(void)`
	as a parameter. This function is the event handler that determines what is to be done at each idle cycle. Often this function will end with a call to `glutPostRedisplay()` as described below. This function is used to define what action the program is to take when there has been no other event to be handled, and is often the function that drives real-time animations.

display	`glutDisplayFunc(functionname)`
	requires a callback function with template
	`void functionname(void)`
	as a parameter. This function is the event handler that generates a new display whenever the display event is received. Note that the display function is invoked by the event handler whenever a display event is reached; this event is posted by the `glutPostRedisplay()` function and whenever a window is moved or reshaped.

reshape	`glutReshapeFunc(functionname)`
	requires a callback function with template
	`void functionname(int, int)`
	as a parameter. This function manages any changes needed in the view setup to accomodate the reshaped window, which may include a fresh definition of the projection. The parameters of the `reshape` function are the width and height of the window after it has been changed.

keyboard	`glutKeyboardFunc(keybd)`
	requires a callback function with template
	`void functionname(unsigned char, int, int)`
	as a parameter. This parameter function is the event handler that receives the character and the location of the cursor `(int x, int y)` when a key

is pressed. As is the case for all callbacks that involve a screen location, the location on the screen will be converted to coordinates relative to the window. Again, this function will often end with a call to `glutPostRedisplay()` to re-display the scene with the changes caused by the particular keyboard event.

special

```
glutSpecialFunc(special)
```

requires a callback function with template

```
    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

```
    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

```
    glutAttachMenu(event),
```

which attaches the current menu to an identified event, and the function

```
    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:

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

```
    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

```
     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

```
     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

```
     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

```
     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 if 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

```
     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. 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 deviceused 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

```
    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

```
    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

```
    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

```
     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.

<u>Idle event callback</u>

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;
GLfloat cubey = 0.0;
GLfloat cubez = 0.0;
GLfloat 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);
    glutIdleFunc(animate);

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

<u>Keyboard callback</u>

Again we start with the familiar `cube()` funcntion. 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' :
        case 'i' : case 'I' :
           ch = key; cubey -= 0.1; break;
        case 'w' : case 'W' :
        case 'o' : case 'O' :
           ch = key; cubey += 0.1; break;
        case 'a' : case 'A' :
        case 'j' : case 'J' :
           ch = key; cubex -= 0.1; break;
        case 's' : case 'S' :
        case 'k' : case 'K' :
           ch = key; cubex += 0.1; break;
        case 'z' : case 'Z' :
        case 'n' : case 'N' :
           ch = key; cubez -= 0.1; break;
        case 'x' : case 'X' :
        case 'm' : case 'M' :
           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 callback for object selection

This example is more complex because it illustrates the use of a mouse event in object selection. This subject is covered in more detail in the later chapter on object selection, and the full code example for this example will also be included there. We will create 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 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. Finally, we include the statement in the `main()` function that registers the mouse callback function.

```
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();
}

...

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

   glSelectBuffer(MAXHITS, selectBuf);
   glRenderMode(GL_SELECT);
   glInitNames();
   glPushName(0);

// 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);
   glClear(GL_COLOR_BUFFER_BIT);
   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);
   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);
// return the label of the object selected, if any
   if (hits <= 0) {
         return -1;
   }
// carry out the color changes that will be the effect of a selection
   temp = cubeColor1; cubeColor1 = cubeColor2; cubeColor2 = temp;
   return selectBuf[3];
}

void main(int argc, char** argv)
{
   ...
   glutMouseFunc(Mouse);
   myinit();
   glutMainLoop();
}
```

Mouse callback for mouse motion

This example shows the callback for the motion event. This event 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. The program this code fragment is from uses the integer coordinates to control spin, but they could be used for many purposes and the application code itself is omitted.

```
void motion(int xPos, int yPos)
{
   spinX = (GLfloat)xPos;
   spinY = (GLfloat)yPos;
}

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

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

The MUI (Micro User Interface) facility

Prerequisites

An understanding of event-driven programming and some experience using the simple events and callbacks from the GLUT toolkit in OpenGL, and some review of interface capabilities in 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 thest 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:

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

An example of the latter function would be

```
     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

```
      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

```
      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

```
      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:

```
      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

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

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

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

to clear the string:

```
      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 `muiGetHSVal(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

```
      muiSetHSValue(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)
   muiGetVSValue(muiObject *obj, float value)
   muiSetVSValue(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 11.1 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

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

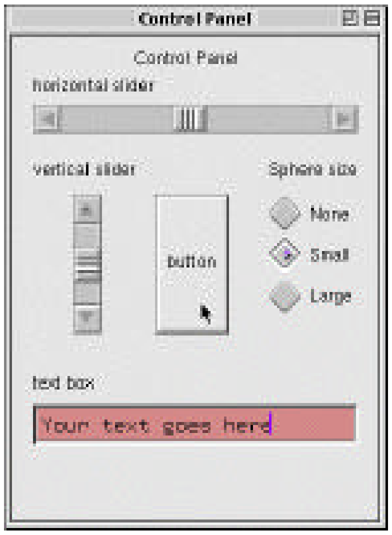

Figure 11.1: the set of MUI facilities on a single window

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

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

where a callback function has the structure

```
     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.

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

```
     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.

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.

```
// 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 11.2 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.

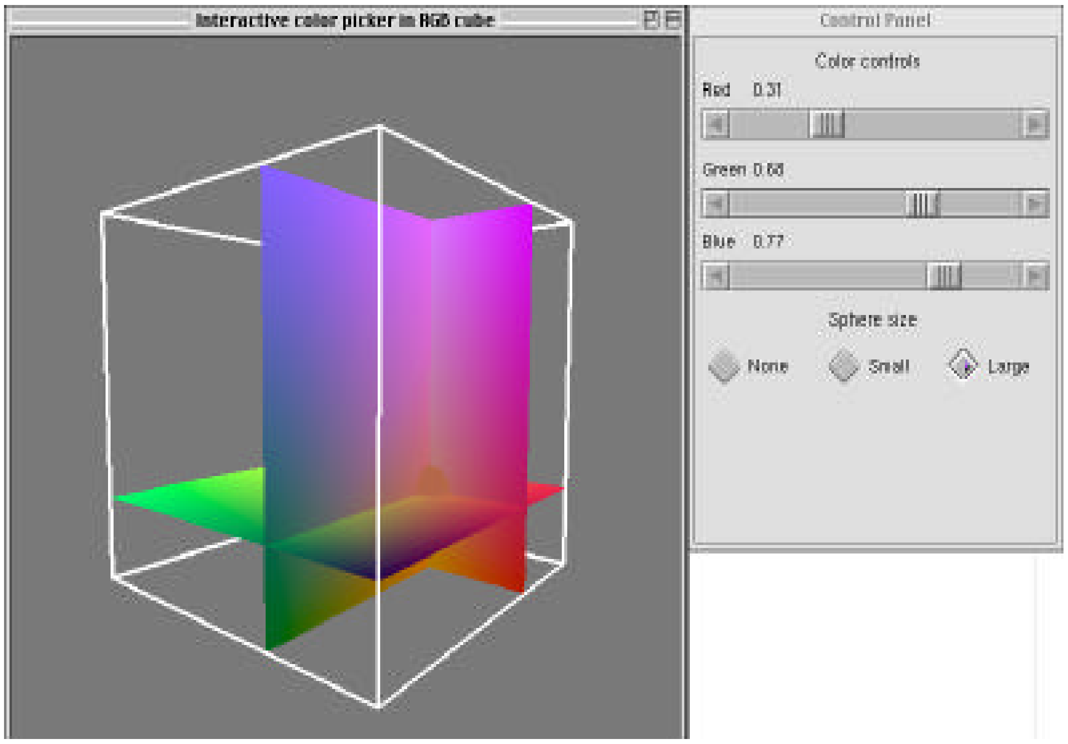

Figure 11.2: the color selector in context, with both the display and control windows shown

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 redisplays) 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 redisplays, 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.