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

/* similar code for ears and mouth */

glEndList();
```

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

We will return to display lists when we discuss hierarchical modeling in Chapter 10. Because most OpenGL code can be encapsulated between a `glNewList` and `glEndList`, you should be able to convert most code to use display lists with little effort.

3.6 PROGRAMMING EVENT-DRIVEN INPUT

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

3.6.1 Using the Pointing Device

We start by altering the `main` function in the gasket program from Chapter 2. In the original version, we used functions in the GLUT library to put a window on the screen and then entered the event loop by executing the function `glutMainLoop`. In that chapter, we entered the loop but did nothing. We could not even terminate the program, except through an external system-dependent mechanism, such as pressing control-c. Our first example will remedy this omission by using the pointing device to terminate a program. We accomplish this task by having the program execute a standard C termination function `exit` when a particular mouse button is pressed.

We discuss only those events recognized by GLUT. Standard window systems such as the X Window System or Microsoft Windows recognize many more events, which differ among systems. However, the GLUT library recognizes a small set of events that is common to most window systems and is sufficient for developing basic interactive graphics programs. Because GLUT has been implemented for the major window systems, we can use our simple applications on multiple systems by recompiling the application.

Two types of events are associated with the pointing device, which is conventionally assumed to be a mouse but could be a trackball or a data tablet. A **move event** is generated when the mouse is moved with one of the buttons pressed. If the mouse is moved without a button being held down, this event is called a **passive move event.** After a move event, the position of the mouse—its measure—is made available to the application program. A **mouse event** occurs when one of the mouse buttons is either pressed or released. A button being held down does not generate a mouse event until the button is released. The information returned includes the button that generated the event, the state of the button after the event (up or down), and the position of the cursor tracking the mouse in window coordinates (with the origin in the upper-left corner of the window). We register the mouse callback function, usually in the `main` function, by means of the GLUT function as follows:

```
glutMouseFunc(myMouse);
```

The mouse callback must have the form

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

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

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

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

Our next example illustrates the benefits of the program structure that we introduced in Chapter 2. We write a program to draw a small box at each location on the screen where the mouse cursor is located at the time that the left button is pressed. We will use the middle button to terminate the program.[5]

5. We are assuming our system has a three-button mouse. Systems with one- or two-button mice can achieve the same behavior by use of the meta keys (*control, alt, shift*) *in combination* with the mouse buttons that are available.

First, we look at the main program, which is much the same as our previous examples.[6]

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

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

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

The mouse callbacks are again in the function `myMouse`.

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

Because only the primitives are generated in `drawSquare`, the desired attributes must have been set elsewhere, such as in our initialization function `myInit`.

We need three global variables. The size of the window may change dynamically, and its present size should be available, both to the reshape callback and to the drawing function `drawSquare`. If we want to change the size of the squares we draw, we may find it beneficial to make the square-size parameter global as well. Our initialization routine selects a clipping window that is the same size as the window

6. We use naming conventions for callbacks similar to those in the *OpenGL Programming Guide* [Ope07].

created in `main` and specifies a viewport to correspond to the entire window. This window is cleared to black. Note that we could omit the setting of the window and viewport here because we are merely setting them to the default values. However, it is illustrative to compare this code with what we do in the reshape callback in Section 3.6.2.

```
/* globals */

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

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

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

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

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

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

Note that we can make the example a little more interesting by setting a new color *in* `drawSquare` each time it is called. After we insert the necessary `include` statements, we have a program that works, as long as the window size remains unchanged.

3.6.2 Window Events

Most window systems allow a user to resize the window interactively, usually by using the mouse to drag a corner of the window to a new location. This event is an example of a **window event**. If such an event occurs, the user program can decide what to do.[7] If the window size changes, we have to consider three questions:

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

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

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

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

    /* adjust clipping box */

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

    /* adjust viewport and clear */
```

7. Unlike most other callbacks, there is a default reshape callback that simply changes the viewport to the new window size, an action that might not be what the user desires.

```
    glViewport(0,0,w,h);

    /* save new window size in global variables */

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

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

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

```
glutMotionFunc(drawSquare);
```

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

Note that the reshape callback generates a display callback. Our simple display callback clears the window so that any squares that are on the display are lost. If we want to retain the squares and redraw them on the resized window, we must create a mechanism to store their descriptions so they can be redrawn on the resized window. We return to this issue in Section 3.10.

3.6.3 Keyboard Events

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

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

```
glutKeyboardFunc(myKey);
```

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

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

GLUT includes a function `glutGetModifiers` that enables the user to define actions using the meta keys, such as the Control and Alt keys. These special keys can be important when we are using one- or two-button mice because we can then define the same functionality as having left, right, and middle buttons as we have assumed in this chapter. More information about these functions is in the Suggested Readings section at the end of the chapter.

3.6.4 The Display and Idle Callbacks

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

```
glutDisplayFunc(myDisplay);
```

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

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

```
glutPostRedisplay();
```

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

The **idle callback** is invoked when there are no other events. Its default is the null function pointer. A typical use of the idle callback is to continue to generate graphical primitives through a display function while nothing else is happening.