Two types of events are associated with the pointing device, which is conventionally assumed to be a mouse but could be a trackpad or a data tablet. A **move event** is generated when the mouse is moved with one of the buttons depressed. If the mouse is moved without a button being held down, this event is called a **passive move event**. After a move event, the position of the mouse is made available to the application program. A **mouse event** occurs when one of the mouse buttons is either depressed or released. When a button is depressed, the action generates a mouse down event. When it is released, a mouse up event is generated. 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

```
glutMouseFunc(myMouse);
```

Mouse Functions

The mouse callback must have the form

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

and is provided 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 depression of the left mouse button to terminate the program. The required callback is the 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 a depression 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.

We will now develop an example that incorporates many of the aspects of CAD programs and adds some interactivity. Along the way, we will introduce some additional callbacks. We start by developing a simple program that will display a single triangle whose vertices are entered interactively using the pointing device. We will use the same shaders so most of the code will be similar to our previous examples.

We specify a global array to hold the three two-dimensional vertices

```
point2 points[3];
```

2.11.6 Window Management

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

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

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

```
glutSetWindow(id);
```

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

2.12 MENUS

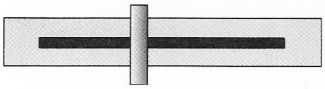

FIGURE 2.46 Slidebar.

We could use our graphics primitives and our mouse callbacks to construct various graphical input devices. For example, we could construct a slidebar (Figure 2.46) using filled rectangles for the device, text for any labels, and the mouse to get the position. However, much of the code would be tedious to develop, especially if we tried to create visually appealing and effective graphical devices (widgets). Most window systems provide a toolkit that contains a set of widgets, but because our philosophy is not to restrict our discussion to any particular window system, we shall not discuss the specifics of such widget sets. Fortunately, GLUT provides one additional feature, **pop-up menus**, that we can use with the mouse to create sophisticated interactive applications.

Using menus involves taking a few simple steps. We must specify the actions corresponding to each entry in the menu. We must link the menu to a particular mouse button. Finally, we must register a callback function for each menu. We can demonstrate simple menus with the example of a pop-up menu that has three entries. The first selection allows us to exit our program. The second and third start and stop the rotation. The function calls to set up the menu and to link it to the right mouse button should be placed in our `main` function. They are

```
glutCreateMenu(demo_menu);
glutAddMenuEntry("quit", 1);
glutAddMenuEntry("start rotation", 2);
glutAddMenuEntry("stop rotation", 3);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

The function `glutCreateMenu` registers the callback function `demo_menu`.

The second argument in each entry's definition is the identifier passed to the callback when the entry is selected. Hence, our callback function is

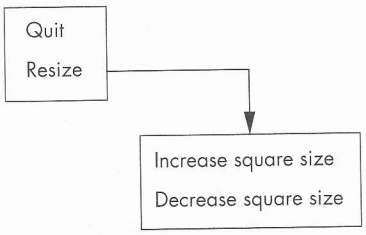

FIGURE 2.47 Structure of hierarchical menus.

```
void demo_menu(int id)
{
     switch(id)
     {
        case 1:
          exit(0);
          break;

        case 2:
          glutIdleFunc(idle);
          break;

        case 3:
          glutIdleFunc(NULL);
          break;
     }
     glutPostRedisplay();
}
```

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

GLUT also supports hierarchical menus, as shown in Figure 2.47. For example, suppose that we want the main menu that we create to have only two entries. The first entry still causes the program to terminate, but now the second causes a submenu to pop up. The submenu contains the two entries for turning the rotation on and off. The following code for the menu (which is in `main`) should be clear:

```
sub_menu = glutCreateMenu(rotation_menu);
glutAddMenuEntry("start rotation", 2);
glutAddMenuEntry("stop rotation", 3);
glutCreateMenu(top_menu);
glutAddMenuEntry("Quit", 1);
glutAddSubMenu("start/stop rotation", sub_menu);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

Writing the callback functions, `rotation_menu` and `top_menu`, should be a simple exercise.