Chapter 13

Selection and Feedback

Chapter Objectives

After reading this chapter, you'll be able to do the following:

- Create applications that allow the user to select a region of the screen or pick an object drawn on the screen
- Use the OpenGL feedback mode to obtain the results of rendering calculations

Some graphics applications simply draw static images of two- and three-dimensional objects. Other applications allow the user to identify objects on the screen and then to move, modify, delete, or otherwise manipulate those objects. OpenGL is designed to support exactly such interactive applications. Since objects drawn on the screen typically undergo multiple rotations, translations, and perspective transformations, it can be difficult for you to determine which object a user is selecting in a three-dimensional scene. To help you, OpenGL provides a selection mechanism that automatically tells you which objects are drawn inside a specified region of the window. You can use this mechanism together with a special utility routine to determine which object within the region the user is specifying, or *picking*, with the cursor.

Selection is actually a mode of operation for OpenGL; feedback is another such mode. In feedback mode, you use your graphics hardware and OpenGL to perform the usual rendering calculations. Instead of using the calculated results to draw an image on the screen, however, OpenGL returns (or feeds back) the drawing information to you. For example, if you want to draw three-dimensional objects on a plotter rather than the screen, you would draw the items in feedback mode, collect the drawing instructions, and then convert them to commands the plotter can understand.

In both selection and feedback modes, drawing information is returned to the application rather than being sent to the framebuffer, as it is in rendering mode. Thus, the screen remains frozen—no drawing occurs—while OpenGL is in selection or feedback mode. In these modes, the contents of the color, depth, stencil, and accumulation buffers are not affected. This chapter explains each of these modes in its own section:

- "Selection" discusses how to use selection mode and related routines to allow a user of your application to pick an object drawn on the screen.
- "Feedback" describes how to obtain information about what would be drawn on the screen and how that information is formatted.

Selection

Typically, when you're planning to use OpenGL's selection mechanism, you first draw your scene into the framebuffer, and then you enter selection mode and redraw the scene. However, once you're in selection mode, the contents of the framebuffer don't change until you exit selection mode. When you exit selection mode, OpenGL returns a list of the primitives that intersect the viewing volume. (Remember that the viewing volume is

defined by the current modelview and projection matrices and any additional clipping planes, as explained in Chapter 3.) Each primitive that intersects the viewing volume causes a selection *hit*. The list of primitives is actually returned as an array of integer-valued *names* and related data—the *hit records*—that correspond to the current contents of the *name stack*. You construct the name stack by loading names onto it as you issue primitive drawing commands while in selection mode. Thus, when the list of names is returned, you can use it to determine which primitives might have been selected on the screen by the user.

In addition to this selection mechanism, OpenGL provides a utility routine designed to simplify selection in some cases by restricting drawing to a small region of the viewport. Typically, you use this routine to determine which objects are drawn near the cursor, so that you can identify which object the user is picking. (You can also delimit a selection region by specifying additional clipping planes. Remember that these planes act in world space, not in screen space.) Since picking is a special case of selection, selection is described first in this chapter, and then picking.

The Basic Steps

To use the selection mechanism, you need to perform the following steps:

1. Specify the array to be used for the returned hit records with **glSelectBuffer()**.
2. Enter selection mode by specifying GL_SELECT with **glRenderMode()**.
3. Initialize the name stack using **glInitNames()** and **glPushName()**.
4. Define the viewing volume you want to use for selection. Usually this is different from the viewing volume you originally used to draw the scene, so you probably want to save and then restore the current transformation state with **glPushMatrix()** and **glPopMatrix()**.
5. Alternately issue primitive drawing commands and commands to manipulate the name stack so that each primitive of interest is assigned an appropriate name.
6. Exit selection mode and process the returned selection data (the hit records).

The following two paragraphs describe **glSelectBuffer()** and **glRenderMode()**. In the next subsection, the commands for manipulating the name stack are described.

void **glSelectBuffer**(GLsizei *size*, GLuint **buffer*);

Specifies the array to be used for the returned selection data. The *buffer* argument is a pointer to an array of unsigned integers into which the data is put, and *size* indicates the maximum number of values that can be stored in the array. You need to call **glSelectBuffer()** before entering selection mode.

GLint **glRenderMode**(GLenum *mode*);

Controls whether the application is in rendering, selection, or feedback mode. The *mode* argument can be GL_RENDER (the default), GL_SELECT, or GL_FEEDBACK. The application remains in a given mode until **glRenderMode()** is called again with a different argument. Before selection mode is entered, **glSelectBuffer()** must be called to specify the selection array. Similarly, before feedback mode is entered, **glFeedbackBuffer()** must be called to specify the feedback array. The return value for **glRenderMode()** has meaning if the current render mode (that is, not the *mode* parameter) is either GL_SELECT or GL_FEEDBACK. The return value is the number of selection hits or the number of values placed in the feedback array when either mode is exited; a negative value means that the selection or feedback array has overflowed. You can use GL_RENDER_MODE with **glGetIntegerv()** to obtain the current mode.

Creating the Name Stack

As mentioned in the preceding subsection, the name stack forms the basis for the selection information that's returned to you. To create the name stack, first initialize it with **glInitNames()**, which simply clears the stack, and then add integer names to it while issuing corresponding drawing commands. As you might expect, the commands to manipulate the stack allow you to push a name onto it (**glPushName()**), pop a name off of it (**glPopName()**), and replace the name at the top of the stack with a different one (**glLoadName()**). Example 13-1 shows what your name-stack manipulation code might look like with these commands.

Example 13-1 Creating a Name Stack

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

glPushMatrix();   /* save the current transformation state */

    /* create your desired viewing volume here */

    glLoadName(1);
    drawSomeObject();
    glLoadName(2);
    drawAnotherObject();
    glLoadName(3);
    drawYetAnotherObject();
    drawJustOneMoreObject();

glPopMatrix();   /* restore the previous transformation state*/
```

In this example, the first two objects to be drawn have their own names, and the third and fourth objects share a single name. With this setup, if either or both of the third and fourth objects cause a selection hit, only one hit record is returned to you. You can have multiple objects share the same name if you don't need to differentiate between them when processing the hit records.

void **glInitNames**(void);

Clears the name stack so that it's empty.

void **glPushName**(GLuint *name*);

Pushes *name* onto the name stack. Pushing a name beyond the capacity of the stack generates the error GL_STACK_OVERFLOW. The name stack's depth can vary among different OpenGL implementations, but it must be able to contain at least 64 names. You can use the parameter GL_NAME_STACK_DEPTH with **glGetIntegerv()** to obtain the depth of the name stack.

void **glPopName**(void);

Pops one name off the top of the name stack. Popping an empty stack generates the error GL_STACK_UNDERFLOW.

void **glLoadName**(GLuint *name*);

Replaces the value at the top of the name stack with *name*. If the stack is empty, which it is right after **glInitNames()** is called, **glLoadName()** generates the error GL_INVALID_OPERATION. To avoid this, if the stack is initially empty, call **glPushName()** at least once to put something on the name stack before calling **glLoadName()**.

Calls to **glPushName()**, **glPopName()**, and **glLoadName()** are ignored if you're not in selection mode. You might find that it simplifies your code to use these calls throughout your drawing code, and then use the same drawing code for both selection and normal rendering modes.

The Hit Record

In selection mode, a primitive that intersects the viewing volume causes a selection hit. Whenever a name-stack manipulation command is executed or **glRenderMode()** is called, OpenGL writes a hit record into the selection array if there's been a hit since the last time the stack was manipulated or **glRenderMode()** was called. With this process, objects that share the same name—for example, an object composed of more than one primitive—don't generate multiple hit records. Also, hit records aren't guaranteed to be written into the array until **glRenderMode()** is called.

Note: In addition to geometric primitives, valid raster position coordinates produced by **glRasterPos() or glWindowPos()** cause a selection hit. Also, during selection mode, if culling is enabled and a polygon is processed and culled, then no hit occurs.

Each hit record consists of the following four items, in the order shown:

- The number of names on the name stack when the hit occurred
- Both the minimum and maximum window-coordinate *z*-values of all vertices of the primitives that have intersected the viewing volume since the last recorded hit. These two values, which lie in the range [0, 1], are each multiplied by $2^{32} - 1$ and rounded to the nearest unsigned integer.
- The contents of the name stack at the time of the hit, with the bottommost element first

When you enter selection mode, OpenGL initializes a pointer to the beginning of the selection array. Each time a hit record is written into the array, the

pointer is updated accordingly. If writing a hit record would cause the number of values in the array to exceed the *size* argument specified with **glSelectBuffer()**, OpenGL writes as much of the record as will fit in the array and sets an overflow flag. When you exit selection mode with **glRenderMode()**, this command returns the number of hit records that were written (including a partial record if there was one), clears the name stack, resets the overflow flag, and resets the stack pointer. If the overflow flag has been set, the return value is –1.

A Selection Example

In Example 13-2, four triangles (a green, a red, and two yellow triangles, created by calling **drawTriangle()**) and a wireframe box representing the viewing volume (**drawViewVolume()**) are drawn on the screen. Then the triangles are rendered again (**selectObjects()**), but this time in selection mode. The corresponding hit records are processed in **processHits()**, and the selection array is printed out. The first triangle generates a hit, the second one doesn't, and the third and fourth ones together generate a single hit.

Example 13-2 Selection Example: select.c

```
void drawTriangle(GLfloat x1, GLfloat y1, GLfloat x2,
    GLfloat y2, GLfloat x3, GLfloat y3, GLfloat z)
{
   glBegin(GL_TRIANGLES);
   glVertex3f(x1, y1, z);
   glVertex3f(x2, y2, z);
   glVertex3f(x3, y3, z);
   glEnd();
}

void drawViewVolume(GLfloat x1, GLfloat x2, GLfloat y1,
                    GLfloat y2, GLfloat z1, GLfloat z2)
{
   glColor3f(1.0, 1.0, 1.0);
   glBegin(GL_LINE_LOOP);
   glVertex3f(x1, y1, -z1);
   glVertex3f(x2, y1, -z1);
   glVertex3f(x2, y2, -z1);
   glVertex3f(x1, y2, -z1);
   glEnd();
```

```
    glBegin(GL_LINE_LOOP);
    glVertex3f(x1, y1, -z2);
    glVertex3f(x2, y1, -z2);
    glVertex3f(x2, y2, -z2);
    glVertex3f(x1, y2, -z2);
    glEnd();

    glBegin(GL_LINES);  /*  4 lines     */
    glVertex3f(x1, y1, -z1);
    glVertex3f(x1, y1, -z2);
    glVertex3f(x1, y2, -z1);
    glVertex3f(x1, y2, -z2);
    glVertex3f(x2, y1, -z1);
    glVertex3f(x2, y1, -z2);
    glVertex3f(x2, y2, -z1);
    glVertex3f(x2, y2, -z2);
    glEnd();
}

void drawScene(void)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(40.0, 4.0/3.0, 1.0, 100.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(7.5, 7.5, 12.5, 2.5, 2.5, -5.0, 0.0, 1.0, 0.0);
    glColor3f(0.0, 1.0, 0.0);   /*  green triangle      */
    drawTriangle(2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -5.0);
    glColor3f(1.0, 0.0, 0.0);   /*  red triangle        */
    drawTriangle(2.0, 7.0, 3.0, 7.0, 2.5, 8.0, -5.0);
    glColor3f(1.0, 1.0, 0.0);   /*  yellow triangles    */
    drawTriangle(2.0, 2.0, 3.0, 2.0, 2.5, 3.0, 0.0);
    drawTriangle(2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -10.0);
    drawViewVolume(0.0, 5.0, 0.0, 5.0, 0.0, 10.0);
}

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

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

```
   for (i = 0; i < hits; i++) { /*  for each hit  */
      names = *ptr;
      printf(" number of names for hit = %d\n", names); ptr++;
      printf("  z1 is %g;", (float) *ptr/0x7fffffff); ptr++;
      printf(" z2 is %g\n", (float) *ptr/0x7fffffff); ptr++;
      printf("   the name is ");
      for (j = 0; j < names; j++) {     /*  for each name */
         printf("%d ", *ptr); ptr++;
      }
      printf("\n");
   }
}

#define BUFSIZE 512

void selectObjects(void)
{
   GLuint selectBuf[BUFSIZE];
   GLint hits;

   glSelectBuffer(BUFSIZE, selectBuf);
   (void) glRenderMode(GL_SELECT);

   glInitNames();
   glPushName(0);

   glPushMatrix();
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(0.0, 5.0, 0.0, 5.0, 0.0, 10.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glLoadName(1);
   drawTriangle(2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -5.0);
   glLoadName(2);
   drawTriangle(2.0, 7.0, 3.0, 7.0, 2.5, 8.0, -5.0);
   glLoadName(3);
   drawTriangle(2.0, 2.0, 3.0, 2.0, 2.5, 3.0, 0.0);
   drawTriangle(2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -10.0);
   glPopMatrix();
   glFlush();

   hits = glRenderMode(GL_RENDER);
   processHits(hits, selectBuf);
}
```

```
void init(void)
{
   glEnable(GL_DEPTH_TEST);
   glShadeModel(GL_FLAT);
}

void display(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   drawScene();
   selectObjects();
   glFlush();
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(200, 200);
   glutInitWindowPosition(100, 100);
   glutCreateWindow(argv[0]);
   init();
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

Picking

As an extension of the process described in the preceding section, you can use selection mode to determine if objects are picked. To do this, you use a special picking matrix in conjunction with the projection matrix to restrict drawing to a small region of the viewport, typically near the cursor. Then you allow some form of input, such as clicking a mouse button, to initiate selection mode. With selection mode established and with the special picking matrix used, objects that are drawn near the cursor cause selection hits. Thus, during picking, you're typically determining which objects are drawn near the cursor.

Picking is set up almost exactly the same as regular selection mode with the following major differences:

- Picking is usually triggered by an input device. In the following code examples, pressing the left mouse button invokes a function that performs picking.
- You use the utility routine **gluPickMatrix()** to multiply the current projection matrix by a special picking matrix. This routine should be called prior to multiplying a standard projection matrix (such as **gluPerspective()** or **glOrtho()**). You'll probably want to save the contents of the projection matrix first, so the sequence of operations may look like this:

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
gluPickMatrix(...);
gluPerspective, glOrtho, gluOrtho2D, or glFrustum
   /* ... draw scene for picking ; perform picking ... */
glPopMatrix();
```

Another completely different way to perform picking is described in "Object Selection Using the Back Buffer" in Chapter 14. This technique uses color values to identify different components of an object.

---

void **gluPickMatrix**(GLdouble *x*, GLdouble *y*, GLdouble *width*, GLdouble *height*, GLint *viewport[4]*);

Creates a projection matrix that restricts drawing to a small region of the viewport and multiplies that matrix onto the current matrix stack. The center of the picking region is (*x*, *y*) in window coordinates, typically the cursor location. *width* and *height* define the size of the picking region in screen coordinates. (You can think of the width and height as the sensitivity of the picking device.) *viewport[]* indicates the current viewport boundaries, which can be obtained by calling

```
glGetIntegerv(GL_VIEWPORT, GLint *viewport);
```


Advanced

Advanced

The net result of the matrix created by **gluPickMatrix()** transforms the clipping region into the unit cube $-1 \le (x, y, z) \le 1$ (or $-w \le (wx, wy, wz) \le w$). The picking matrix effectively performs an orthogonal transformation that maps the clipping region to the unit cube. Since the transformation is arbitrary, you can make picking work for different sorts of regions—for example, for rotated rectangular portions of the window. In certain situations, you might find it easier to specify additional clipping planes to define the picking region.

Example 13-3 illustrates simple picking. It also demonstrates how to use multiple names to identify different components of a primitive; in this case, the row and column of a selected object. A 3×3 grid of squares is drawn, with each square a different color. The *board*[3][3] array maintains the current amount of blue for each square. When the left mouse button is pressed, the **pickSquares()** routine is called to identify which squares were picked by the mouse. Two names identify each square in the grid—one identifies the row, and the other the column. Also, when the left mouse button is pressed, the colors of all squares under the cursor position change.

Example 13-3 Picking Example: picksquare.c

```
int board[3][3];   /*  amount of color for each square  */

/*  Clear color value for every square on the board   */
void init(void)
{
   int i, j;
   for (i = 0; i < 3; i++)
      for (j = 0; j < 3; j ++)
         board[i][j] = 0;
   glClearColor(0.0, 0.0, 0.0, 0.0);
}

void drawSquares(GLenum mode)
{
   GLuint i, j;
   for (i = 0; i < 3; i++) {
      if (mode == GL_SELECT)
         glLoadName(i);
      for (j = 0; j < 3; j ++) {
         if (mode == GL_SELECT)
            glPushName(j);
         glColor3f((GLfloat) i/3.0, (GLfloat) j/3.0,
                   (GLfloat) board[i][j]/3.0);
```

```
            glRecti(i, j, i+1, j+1);
            if (mode == GL_SELECT)
               glPopName();
         }
      }
}

/*  processHits prints out the contents of the
 *  selection array.
 */
void processHits(GLint hits, GLuint buffer[])
{
   unsigned int i, j;
   GLuint ii, jj, names, *ptr;

   printf("hits = %d\n", hits);
   ptr = (GLuint *) buffer;
   for (i = 0; i < hits; i++) { /*  for each hit  */
      names = *ptr;
      printf(" number of names for this hit = %d\n", names);
         ptr++;
      printf(" z1 is %g;", (float) *ptr/0x7fffffff); ptr++;
      printf(" z2 is %g\n", (float) *ptr/0x7fffffff); ptr++;
      printf("   names are ");
      for (j = 0; j < names; j++) { /*  for each name */
         printf("%d ", *ptr);
         if (j == 0)  /*  set row and column  */
            ii = *ptr;
         else if (j == 1)
            jj = *ptr;
         ptr++;
      }
      printf("\n");
      board[ii][jj] = (board[ii][jj] + 1) % 3;
   }
}

#define BUFSIZE 512

void pickSquares(int button, int state, int x, int y)
{
   GLuint selectBuf[BUFSIZE];
   GLint hits;
   GLint viewport[4];
```

```
   if (button != GLUT_LEFT_BUTTON || state != GLUT_DOWN)
      return;

   glGetIntegerv(GL_VIEWPORT, viewport);

   glSelectBuffer(BUFSIZE, selectBuf);
   (void) glRenderMode(GL_SELECT);

   glInitNames();
   glPushName(0);

   glMatrixMode(GL_PROJECTION);
   glPushMatrix();
   glLoadIdentity();
/*  create 5x5 pixel picking region near cursor location      */
   gluPickMatrix((GLdouble) x, (GLdouble) (viewport[3] - y),
                 5.0, 5.0, viewport);
   gluOrtho2D(0.0, 3.0, 0.0, 3.0);
   drawSquares(GL_SELECT);

   glMatrixMode(GL_PROJECTION);
   glPopMatrix();
   glFlush();

   hits = glRenderMode(GL_RENDER);
   processHits(hits, selectBuf);
   glutPostRedisplay();
}

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

void reshape(int w, int h)
{
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(0.0, 3.0, 0.0, 3.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
```

```
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(100, 100);
   glutInitWindowPosition(100, 100);
   glutCreateWindow(argv[0]);
   init();
   glutMouseFunc(pickSquares);
   glutReshapeFunc(reshape);
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

Picking with Multiple Names and a Hierarchical Model

Multiple names can also be used to choose parts of a hierarchical object in a scene. For example, if you were rendering an assembly line of automobiles, you might want the user to move the mouse to pick the third bolt on the left front wheel of the third car in line. A different name can be used to identify each level of hierarchy: which car, which wheel, and finally which bolt. As another example, one name can be used to describe a single molecule among other molecules, and additional names can differentiate individual atoms within that molecule.

Example 13-4 is a modification of Figure 3-4, which draws an automobile with four identical wheels, each of which has five identical bolts. Code has been added to manipulate the name stack with the object hierarchy.

Example 13-4 Creating Multiple Names

```
draw_wheel_and_bolts()
{
    long i;

    draw_wheel_body();
    for (i = 0; i < 5; i++) {
        glPushMatrix();
            glRotate(72.0*i, 0.0, 0.0, 1.0);
            glTranslatef(3.0, 0.0, 0.0);
            glPushName(i);
                draw_bolt_body();
            glPopName();
        glPopMatrix();
    }
 }
```

```
draw_body_and_wheel_and_bolts()
{
    draw_car_body();
    glPushMatrix();
        glTranslate(40, 0, 20);  /* first wheel position*/
        glPushName(1);           /* name of wheel number 1 */
            draw_wheel_and_bolts();
        glPopName();
    glPopMatrix();
    glPushMatrix();
        glTranslate(40, 0, -20); /* second wheel position */
        glPushName(2);           /* name of wheel number 2 */
            draw_wheel_and_bolts();
        glPopName();
    glPopMatrix();

    /* draw last two wheels similarly */
 }
```

Example 13-5 uses the routines in Example 13-4 to draw three different cars, numbered 1, 2, and 3.

Example 13-5 Using Multiple Names

```
draw_three_cars()
{
    glInitNames();
    glPushMatrix();
        translate_to_first_car_position();
        glPushName(1);
            draw_body_and_wheel_and_bolts();
        glPopName();
    glPopMatrix();

    glPushMatrix();
        translate_to_second_car_position();
        glPushName(2);
            draw_body_and_wheel_and_bolts();
        glPopName();
    glPopMatrix();

    glPushMatrix();
        translate_to_third_car_position();
        glPushName(3);
            draw_body_and_wheel_and_bolts();
        glPopName();
    glPopMatrix();
}
```

Assuming that picking is performed, the following are some possible name-stack return values and their interpretations. In these examples, at most one hit record is returned; also, *d1* and *d2* are depth values.

2 *d1 d2* 2 1	Car 2, wheel 1
1 *d1 d2* 3	Car 3 body
3 *d1 d2* 1 1 0	Bolt 0 on wheel 1 on car 1
empty	The pick was outside all cars

The last interpretation assumes that the bolt and wheel don't occupy the same picking region. A user might well pick both the wheel and the bolt, yielding two hits. If you receive multiple hits, you have to decide which hit to process, perhaps by using the depth values to determine which picked object is closest to the viewpoint. The use of depth values is explored further in the next subsection.

Picking and Depth Values

Example 13-6 demonstrates how to use depth values when picking to determine which object is picked. This program draws three overlapping rectangles in normal rendering mode. When the left mouse button is pressed, the **pickRects()** routine is called. This routine returns the cursor position, enters selection mode, initializes the name stack, and multiplies the picking matrix with the current orthographic projection matrix. A selection hit occurs for each rectangle the cursor is over when the left mouse button is clicked. Finally, the contents of the selection buffer are examined to identify which named objects were within the picking region near the cursor.

The rectangles in this program are drawn at different depth, or *z*-, values. Since only one name is used to identify all three rectangles, only one hit can be recorded. However, if more than one rectangle is picked, that single hit has different minimum and maximum *z*-values.

Example 13-6 Picking with Depth Values: pickdepth.c

```
void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glEnable(GL_DEPTH_TEST);
   glShadeModel(GL_FLAT);
   glDepthRange(0.0, 1.0);  /* The default z mapping */
}
```

```
void drawRects(GLenum mode)
{
   if (mode == GL_SELECT)
      glLoadName(1);
   glBegin(GL_QUADS);
   glColor3f(1.0, 1.0, 0.0);
   glVertex3i(2, 0, 0);
   glVertex3i(2, 6, 0);
   glVertex3i(6, 6, 0);
   glVertex3i(6, 0, 0);
   glEnd();
   if (mode == GL_SELECT)
      glLoadName(2);
   glBegin(GL_QUADS);
   glColor3f(0.0, 1.0, 1.0);
   glVertex3i(3, 2, -1);
   glVertex3i(3, 8, -1);
   glVertex3i(8, 8, -1);
   glVertex3i(8, 2, -1);
   glEnd();
   if (mode == GL_SELECT)
      glLoadName(3);
   glBegin(GL_QUADS);
   glColor3f(1.0, 0.0, 1.0);
   glVertex3i(0, 2, -2);
   glVertex3i(0, 7, -2);
   glVertex3i(5, 7, -2);
   glVertex3i(5, 2, -2);
   glEnd();
}

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

   printf("hits = %d\n", hits);
   ptr = (GLuint *) buffer;
   for (i = 0; i < hits; i++) {  /* for each hit  */
      names = *ptr;
      printf(" number of names for hit = %d\n", names); ptr++;
      printf(" z1 is %g;", (float) *ptr/0x7fffffff); ptr++;
```

```
      printf(" z2 is %g\n", (float) *ptr/0x7fffffff); ptr++;
      printf("   the name is ");
      for (j = 0; j < names; j++) {  /* for each name */
         printf("%d ", *ptr); ptr++;
      }
      printf("\n");
   }
}

#define BUFSIZE 512

void pickRects(int button, int state, int x, int y)
{
   GLuint selectBuf[BUFSIZE];
   GLint hits;
   GLint viewport[4];

   if (button != GLUT_LEFT_BUTTON || state != GLUT_DOWN)
      return;
   glGetIntegerv(GL_VIEWPORT, viewport);

   glSelectBuffer(BUFSIZE, selectBuf);
   (void) glRenderMode(GL_SELECT);

   glInitNames();
   glPushName(0);

   glMatrixMode(GL_PROJECTION);
   glPushMatrix();
   glLoadIdentity();
/*  create 5x5 pixel picking region near cursor location */
   gluPickMatrix((GLdouble) x, (GLdouble) (viewport[3] - y),
                 5.0, 5.0, viewport);
   glOrtho(0.0, 8.0, 0.0, 8.0, -0.5, 2.5);
   drawRects(GL_SELECT);
   glPopMatrix();
   glFlush();

   hits = glRenderMode(GL_RENDER);
   processHits(hits, selectBuf);
}
```

```
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   drawRects(GL_RENDER);
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(0.0, 8.0, 0.0, 8.0, -0.5, 2.5);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

int main(int argc, char **argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(200, 200);
   glutInitWindowPosition(100, 100);
   glutCreateWindow(argv[0]);
   init();
   glutMouseFunc(pickRects);
   glutReshapeFunc(reshape);
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```


Try This

Try This

- Modify Example 13-6 to add additional calls to **glPushName()** so that multiple names are on the stack when the selection hit occurs. What will the contents of the selection buffer be?
- By default, **glDepthRange()** sets the mapping of the *z*-values to [0.0, 1.0]. Try modifying the **glDepthRange()** values and see how these modifications affect the *z*-values that are returned in the selection array.