Chapter 12

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 OpenGL's 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. If you want to draw three−dimensional objects on a plotter rather than the screen, for example, 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. 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. Once you're in selection mode, however, the contents of the framebuffer don't change until you exit selection mode. When you exit, OpenGL returns a list of the primitives that would have intersected the viewing volume (remember that the viewing volume is defined by the current modelview and projection matrices and any clipping planes you've specified, as explained in **"Additional Clipping Planes."**) 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; see **"Additional Clipping Planes"** for more information about how to do this. 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 used to draw the scene originally, 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 has an appropriate name assigned.
6. Exit selection mode and process the returned selection data (the hit records).

The following paragraphs describe *glSelectBuffer()* and *glRenderMode()* . In the next section, the commands to manipulate 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 one of 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 entering selection mode, *glSelectBuffer()* must be called to specify the selection array. Similarly, before entering feedback mode, *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 previous section, the name stack forms the basis for the selection information that's returned to you. To create the name stack, first you initialize it with *glInitNames()* , which simply clears the stack, and then you add integer names to it as you issue 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 on the top of the stack with a different one (*glLoadName()*). **Example 12−1** shows what your name−stack manipulation code might look like with these commands.

Example 12−1 Creating a Name Stack

```
glInitNames();
glPushName(−1);
```

```
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 causes 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 sixty−four 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 on 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 that's 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 primitives, valid coordinates produced by *glRasterPos()* can cause a selection hit.

In the case of polygons, no hit occurs if the polygon would have been culled.

Each hit record consists of four items, in order:

- 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 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 fits 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 had been set, the return value is −1.

A Selection Example

In **Example 12−2**, four triangles (green, red, and two yellow ones) are drawn in selection mode, and the corresponding hit records are processed. The first triangle generates a hit, the second one doesn't, and the third and fourth ones together generate a single hit. Routines are defined to draw a triangle (*drawTriangle()*) and to draw a wireframe box representing the viewing volume (*drawViewVolume()*). The *processHits()* routine prints out the selection array. Finally, *selectObjects()* draws the triangles in selection mode to generate the hit records.

Example 12−2 A Selection Example: select.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

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, 0.01, 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 %u;", *ptr); ptr++;
        printf (" z2 is %u\n", *ptr); 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, viewport[4];

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

    glInitNames();
    glPushName(-1);

    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 myinit (void)
{
    glDepthFunc(GL_LEQUAL);
    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)
{
```

```
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 200, 200);
    auxInitWindow (argv[0]);
    myinit ();
    auxMainLoop(display);
}
```

Picking

As an extension of the process described in the previous 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 like regular selection mode is, 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 a special projection matrix onto the current matrix. This routine should be called prior to multiplying a projection matrix onto the stack.

Another, completely different way to perform picking is described in **"Object Selection Using the Back Buffer."** 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

The net result of the matrix created by *gluPickMatrix()* is to transform the clipping region into the unit cube $-1 \leq (x, y, z) \leq 1$ (or $-w \leq (wx, wy, wz) \leq w$). The picking matrix effectively performs an orthogonal transformation that maps a subregion of this unit cube 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 12–3 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 color of all squares under the cursor position changes.

Example 12–3 A Picking Example: picksquare.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"
```

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

/* Clear color value for every square on the board */
void myinit(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 ();
        }
    }
}

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 %u;", *ptr); ptr++;
        printf (" z2 is %u\n", *ptr); 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;
```

```
    board[ii][jj] = (board[ii][jj] + 1) % 3;
    }
}

#define BUFSIZE 512

void pickSquares(AUX_EVENTREC *event)
{
    GLuint selectBuf[BUFSIZE];
    GLint hits;
    GLint viewport[4];
    int x, y;

    x = event->data[AUX_MOUSEX];
    y = event->data[AUX_MOUSEY];
    glGetIntegerv (GL_VIEWPORT, viewport);

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

    glInitNames();
    glPushName(-1);

    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);
    glPopMatrix ();
    glFlush ();

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

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

void myReshape(GLsizei w, GLsizei 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)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
    auxInitPosition (0, 0, 100, 100);
    auxInitWindow (argv[0]);
    myinit ();
    auxMouseFunc (AUX_LEFTBUTTON, AUX_MOUSEDOWN, pickSquares);
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

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 tire of the third car in line. A different name can be used to identify each level of hierarchy: which car, which tire, 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 12–4 is a modification of **Example 3–4** that 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 12–4 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 12−5 uses the routines in **Example 12−4** to draw three different cars, numbered 1, 2, and 3.

Example 12−5 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.

empty	The pick was outside all cars
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

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 section.

Picking and Depth Values

Example 12−6 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 onto the stack before the 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 is 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 12−6 Picking with Depth Values: pickdepth.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

void myinit(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glDepthFunc(GL_LEQUAL);
    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);
        glColor3f (0.0, 1.0, 1.0);
        glVertex3i (3, 2, -1);
        glVertex3i (3, 8, -1);
        glVertex3i (8, 8, -1);
        glVertex3i (8, 2, -1);
        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 %u;", *ptr); ptr++;
        printf (" z2 is %u\n", *ptr); 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(AUX_EVENTREC *event)
{
    GLuint selectBuf[BUFSIZE];
    GLint hits;
    GLint viewport[4];
    int x, y;

    x = event->data[AUX_MOUSEX];
    y = event->data[AUX_MOUSEY];
    glGetIntegerv (GL_VIEWPORT, viewport);

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

    glInitNames();
    glPushName(-1);

    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, 0.0, 2.0);
        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 myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho (0.0, 8.0, 0.0, 8.0, 0.0, 2.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}


int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 100, 100);
    auxInitWindow (argv[0]);
    myinit ();
    auxMouseFunc (AUX_LEFTBUTTON, AUX_MOUSEDOWN, pickRects);
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

Try This

Try This

- Modify **Example 12−6** 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 it affects the *z* values that are returned in the selection array.

Hints for Writing a Program That Uses Selection

Most programs that allow a user to interactively edit some geometry provide a mechanism for the user to pick items or groups of items for editing. For two−dimensional drawing programs (for example, text editors, page−layout programs, circuit−design programs), it might be easier to do your own picking calculations instead of using the OpenGL picking mechanism. Often, it's easy to find bounding boxes for two−dimensional objects and to organize them in some hierarchical data structure to speed up searches. For example, OpenGL−style picking in a VLSI layout program that has millions of rectangles can be relatively slow. However, using simple bounding−box information when rectangles are typically aligned with the screen could make picking in such a program extremely fast. The code is probably simpler to write, too.

As another example, since only geometric objects cause hits, you might want to create your own method for picking text. Setting the current raster position is a geometric operation, but it effectively creates only a single pickable point at the current raster position, which is typically at the lower left−hand corner of the text. If your editor needs to manipulate individual characters within a text string, some other picking mechanism must be used. You could draw little rectangles around each character during picking mode, but it's almost certainly easier to handle text as a special case.

If you decide to use OpenGL picking, organize your program and its data structures so that it's easy to draw appropriate lists of objects in either selection or normal drawing mode. This way, when the user picks something, you can use the same data structures for the pick operation that you use to display the items on the screen. Also, consider whether you want to allow the user to select multiple objects. One way to do this is to store a bit for each item indicating whether it's selected, but with this method, you have to traverse your entire list of items to find the selected items. You might find it useful to maintain a list of pointers to selected items to speed up this search. It's probably a good idea to keep the selection bit for each item as well, since when you're drawing the entire picture, you might want to draw selected items differently (for example, in a different color or with a selection box around them). Finally, consider the selection user interface. You might want to allow the user to do the following:

- Select an item
- Sweep−select a group of items (see the next paragraphs for a description of this behavior)
- Add an item to the selection
- Add a sweep selection to the current selections
- Delete an item from a selection
- Choose a single item from a group of overlapping items

A typical solution for a two−dimensional drawing program might work as follows.

1. All selection is done by pointing with the mouse cursor and using the left mouse button. In what follows, *cursor* means the cursor tied to the mouse, and *button* means the left mouse button.
2. Clicking on an item selects it and deselects all other currently selected items. If the cursor is on top of multiple items, the smallest is selected. (In three dimensions, many other strategies work to disambiguate a selection.)
3. Clicking down where there is no item, holding the button down while dragging the cursor, and then releasing the button selects all the items in a screen−aligned rectangle whose corners are determined by the cursor positions when the button went down and where it came up. This is called a *sweep selection*. All items not in the swept−out region are deselected. (You must decide whether an item is selected only if it's completely within the sweep region, or if any part of it falls within the region. The completely within strategy usually works best.)
4. If the Shift key is held down and the user clicks on an item that isn't currently selected, that item is added to the selected list. If the clicked−upon item is selected, it's deleted from the selection list.
5. If a sweep selection is performed with the Shift key pressed, the items swept out are added to the current selection.
6. In an extremely cluttered region, it's often hard to do a sweep selection. When the button goes down, the cursor might lie on top of some item, and normally, that item would be selected. You can make any operation a sweep selection, but a typical user interface interprets a button−down on an item plus a mouse motion as a select−plus−drag operation. To solve this problem, you can have an enforced sweep selection by holding down, say, the Alt key. With this, the following set of operations constitutes a sweep selection: Alt−button down, sweep, button up. Items under the cursor when the button goes down are ignored.
7. If the Shift key is held during this sweep selection, the items enclosed in the sweep region are added to the current selection.
8. Finally, if the user clicks on multiple items, select just one of them. If the cursor isn't moved (or maybe not moved more than a pixel), and the user clicks again in the same place, deselect the item originally selected, and select a different item under the cursor. Use repeated clicks at the same point to cycle through all the possibilities.

Different rules can apply in particular situations. In a text editor, you probably don't have to worry about characters on top of each other, and selections of multiple characters are always contiguous characters in the document. Thus, you need to mark only the first and last selected characters to identify the complete selection. With text, often the best way to handle selection is to identify the positions between characters rather than the characters themselves. This allows you to have an empty selection when the beginning and end of the selection are between the same pair of characters, and to put the cursor before the first character in the document or after the final one with no special−case code.

In three−dimensional editors, you might provide ways to rotate and zoom between selections, so sophisticated schemes for cycling through the possible selections might be unnecessary. On the other hand, selection in three dimensions is difficult because the cursor's position on the screen usually gives no indication of its depth.

Feedback

Feedback is similar to selection in that once you're in either mode, no pixels are produced and the screen is frozen. Instead of drawing occurring, information about primitives that would have been rendered is sent back to the application. The key difference between selection and feedback modes is what information is sent back. In selection mode, assigned names are returned to an array of integer values. In feedback mode, information about transformed primitives is sent back to an array of floating−point values. The values sent back to the feedback array consist of tokens that specify what type of primitive (point, line, polygon, image, or bitmap) has been processed and transformed, followed by vertex, color, or other data for that primitive. The values returned are fully transformed by lighting and viewing operations. Feedback mode is initiated by calling *glRenderMode()* with GL_FEEDBACK as the argument.

Here's how you enter and exit feedback mode:

1. Call *glFeedbackBuffer()* to specify the array to hold the feedback information. The arguments to this command describe what type of data and how much of it gets written into the array.
2. Call *glRenderMode()* with GL_FEEDBACK as the argument to enter feedback mode. (You can ignore the value returned by *glRenderMode()*.) After this point, until you exit feedback mode, primitives aren't rasterized to produce pixels, and the contents of the framebuffer don't change.
3. Draw your primitives. As you issue drawing commands, you can make several calls to *glPassThrough()* to insert markers into the returned feedback data to help you parse it more easily.
4. Exit feedback mode by calling *glRenderMode()*, with GL_RENDER as the argument if you want to return to normal drawing mode. The integer value returned by *glRenderMode()* is the number of values stored in the feedback array.
5. Parse the data in the feedback array.

void *glFeedbackBuffer* (GLsizei *size*, GLenum *type*, GLfloat * *buffer*);

Establishes a buffer for the feedback data: *buffer* is a pointer to an array where the data is stored. The *size* argument indicates the maximum number of values that can be stored in the array. The *type* argument describes the information fed back for each vertex in the feedback array; its possible values and their meaning are shown in **Table 12−1**. *glFeedbackBuffer()* must be called before feedback mode is entered. In the table, *k* is 1 in color−index mode and 4 in RGBA mode.

Type Argument	Coordinates	Color	Texture	Total Values
GL_2D	x, y	–	–	2
GL_3D	x, y, z	–	–	3
GL_3D_COLOR	x, y, z	k	–	3 + k
GL_3D_COLOR_TEXTURE	x, y, z	k	4	7 + k
GL_4D_COLOR_TEXTURE	x, y, z, w	k	4	8 + k

Table 12−1 Values for the Type Argument to glFeedbackBuffer()

The Feedback Array

In feedback mode, each primitive that would be rasterized (or each call to *glDrawPixels()* or *glCopyPixels()*, if the raster position is valid) generates a block of values that's copied into the feedback array. The number of values is determined by the *type* argument to *glFeedbackBuffer()*, as listed in **Table 12−1**. Use the appropriate value for the type of primitives you're drawing: GL_2D or GL_3D for unlit two− or three−dimensional primitives, GL_3D_COLOR for lit, three−dimensional primitives, and

GL_3D_COLOR_TEXTURE or GL_4D_COLOR_TEXTURE for lit, textured, three− or four−dimensional primitives.

Each block of feedback values begins with a code indicating the primitive type, followed by values that describe the primitive's vertices and associated data. Entries are also written for pixel rectangles. In addition, pass−through markers that you've explicitly created can be returned in the array; the next section explains these markers in more detail. **Table 12−2** shows the syntax for the feedback array; remember that the data associated with each returned vertex is as described in **Table 12−1**. Note that a polygon can have *n* vertices returned. Also, the *x, y, z* coordinates returned by feedback are window coordinates; if *w* is returned, it's in clip coordinates. For bitmaps and pixel rectangles, the coordinates returned are those of the current raster position. In the table, note that GL_LINE_RESET_TOKEN is returned only when the line stipple is reset for that line segment.

Primitive Type	Code	Associated Data
Point	GL_POINT_TOKEN	vertex
Line	GL_LINE_TOKEN or GL_LINE_RESET_TOKEN	vertex vertex
Polygon	GL_POLYGON_TOKEN	n vertex vertex ... vertex
Bitmap	GL_BITMAP_TOKEN	vertex
Pixel Rectangle	GL_DRAW_PIXEL_TOKEN or GL_COPY_PIXEL_TOKEN	vertex
Pass−through	GL_PASS_THROUGH_TOKE N	a floating−point number

Table 12−2 Feedback Array Syntax

Using Markers in Feedback Mode

Feedback occurs after transformations, lighting, polygon culling, and interpretation of polygons by *glPolygonMode()*. It might also occur after polygons with more than three edges are broken up into triangles (if your particular OpenGL implementation renders polygons by performing this decomposition). Thus, it might be hard for you to recognize the primitives you drew in the feedback data you receive. To help yourself parse the feedback data, call *glPassThrough()* as needed in your sequence of drawing commands to insert a marker. You might use the markers to separate the feedback values returned from different primitives, for example. This command causes GL_PASS_THROUGH_TOKEN to be written into the feedback array, followed by the floating−point value you pass in as an argument.

void *glPassThrough* (GLfloat *token*);

Inserts a marker into the stream of values written into the feedback array, if called in feedback mode. The marker consists of the code GL_PASS_THROUGH_TOKEN followed by a single floating−point value, *token*. This command has no effect when called outside of feedback mode. Calling *glPassThrough()* between *glBegin()* and *glEnd()* generates a GL_INVALID_OPERATION error.

A Feedback Example

Example 12−7 demonstrates the use of feedback mode. This program draws a lit, three−dimensional scene in normal rendering mode. Then, feedback mode is entered, and the scene is redrawn. Since the program draws lit, untextured, three−dimensional objects, the type of feedback data is GL_3D_COLOR. Since RGBA mode is used, each unclipped vertex generates seven values for the feedback buffer: *x, y, z, r, g, b,* and *a*.

In feedback mode, the program draws two lines as part of a line strip and then inserts a pass−through marker. Next, a point is drawn at (−100.0, −100.0, −100.0), which falls outside the orthographic viewing volume and thus doesn't put any values into the feedback array. Finally, another pass−through marker is inserted, and another point is drawn.

Example 12−7 Using Feedback Mode: feedback.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

void myinit(void)
{
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
}

void drawGeometry (GLenum mode)
{
    glBegin (GL_LINE_STRIP);
        glNormal3f (0.0, 0.0, 1.0);
        glVertex3f (30.0, 30.0, 0.0);
        glVertex3f (50.0, 60.0, 0.0);
        glVertex3f (70.0, 40.0, 0.0);
    glEnd ();
    if (mode == GL_FEEDBACK)
        glPassThrough (1.0);

    glBegin (GL_POINTS);
        glVertex3f (-100.0, -100.0, -100.0);
    glEnd ();
    if (mode == GL_FEEDBACK)
        glPassThrough (2.0);

    glBegin (GL_POINTS);
        glNormal3f (0.0, 0.0, 1.0);
        glVertex3f (50.0, 50.0, 0.0);
    glEnd ();
}

void print3DcolorVertex (GLint size, GLint *count,
        GLfloat *buffer)
{
    int i;

    printf ("  ");
    for (i = 0; i < 7; i++) {
        printf ("%4.2f ", buffer[size-(*count)]);
        *count = *count - 1;
    }
    printf ("\n");
}

void printBuffer(GLint size, GLfloat *buffer)
{
    GLint count;
    GLfloat token;

    count = size;
    while (count) {
```

```
        token = buffer[size-count]; count--;
        if (token == GL_PASS_THROUGH_TOKEN) {
            printf ("GL_PASS_THROUGH_TOKEN\n");
            printf ("  %4.2f\n", buffer[size-count]);
            count--;
        }
        else if (token == GL_POINT_TOKEN) {
            printf ("GL_POINT_TOKEN\n");
            print3DcolorVertex (size, &count, buffer);
        }
        else if (token == GL_LINE_TOKEN) {
            printf ("GL_LINE_TOKEN\n");
            print3DcolorVertex (size, &count, buffer);
            print3DcolorVertex (size, &count, buffer);
        }
        else if (token == GL_LINE_RESET_TOKEN) {
            printf ("GL_LINE_RESET_TOKEN\n");
            print3DcolorVertex (size, &count, buffer);
            print3DcolorVertex (size, &count, buffer);
        }
    }
}

void display(void)
{
    GLfloat feedBuffer[1024];
    GLint size;

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    glOrtho (0.0, 100.0, 0.0, 100.0, 0.0, 1.0);

    glClearColor (0.0, 0.0, 0.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT);
    drawGeometry (GL_RENDER);

    glFeedbackBuffer (1024, GL_3D_COLOR, feedBuffer);
    (void) glRenderMode (GL_FEEDBACK);
    drawGeometry (GL_FEEDBACK);

    size = glRenderMode (GL_RENDER);
    printBuffer (size, feedBuffer);
}


int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
    auxInitPosition (0, 0, 100, 100);
    auxInitWindow (argv[0]);
    myinit ();
    auxMainLoop(display);
}
```

Running this program generates the following output:

```
GL_LINE_RESET_TOKEN
 30.00 30.00 0.00 0.84 0.84 0.84 1.00
 50.00 60.00 0.00 0.84 0.84 0.84 1.00
GL_LINE_TOKEN
 50.00 60.00 0.00 0.84 0.84 0.84 1.00
 70.00 40.00 0.00 0.84 0.84 0.84 1.00
GL_PASS_THROUGH_TOKEN
 1.00
GL_PASS_THROUGH_TOKEN
 2.00
GL_POINT_TOKEN
 50.00 50.00 0.00 0.84 0.84 0.84 1.00
```

Thus, the line strip drawn with these commands results in two primitives:

```
glBegin(GL_LINE_STRIP);
    glNormal3f (0.0, 0.0, 1.0);
    glVertex3f (30.0, 30.0, 0.0);
    glVertex3f (50.0, 60.0, 0.0);
    glVertex3f (70.0, 40.0, 0.0);
glEnd();
```

The first primitive begins with GL_LINE_RESET_TOKEN, which indicates that the primitive is a line segment and that the line stipple is reset. The second primitive begins with GL_LINE_TOKEN, so it's also a line segment, but the line stipple isn't reset and hence continues from where the previous line segment left off. Each of the two vertices for these lines generates seven values for the feedback array. Note that the RGBA values for all four vertices in these two lines are (0.84, 0.84, 0.84, 1.0), which is a very light gray color with the maximum alpha value. These color values are a result of the interaction of the surface normal and lighting parameters.

Since no feedback data is generated between the first and second pass−through markers, you can deduce that any primitives drawn between the first two calls to *glPassThrough()* were clipped out of the viewing volume. Finally, the point at (50.0, 50.0, 0.0) is drawn, and its associated data is copied into the feedback array.

Try This

Try This

Make changes to **Example 12−7** and see how they affect the feedback values that are returned. For example, change the coordinate values of *glOrtho()* . Change the lighting variables, or eliminate lighting altogether and change the feedback type to GL_3D. Or add more primitives to see what other geometry (such as filled polygons) contributes to the feedback array.