

Object Selection

Prerequisites

An understanding of the rendering process, an understanding of event handling, and a knowledge of list management to handle hit lists for events

Introduction

OpenGL has many facilities for responding to mouse events — clicks on the screen — but most of them are complex and require the programmer to do significant work to identify the objects that lie between the front and back clipping planes along the line between the points in those planes that correspond to the click. However, OpenGL makes it possible for you to get the same information with much less pain with a built-in selection operation.

The built-in selection approach calls for you to render your scene twice. In the first rendering, you work in the same mode you are used to: you simply draw the scene in `GL_RENDER` mode. When you get a mouse event, you change to `GL_SELECT` mode and re-draw the scene with each item of interest given a unique name. When the scene is rendered, nothing is changed in the frame buffer but the pixels that would be rendered are identified. When any named object is found that would meet the mouse point, that object's name is added to a name stack for the point. This name stack holds the names of all the items in a hierarchy of named items that were hit. This will create a list of hit records, one for each item that meets the mouse click point, and you can then process this list to identify the item nearest the eye that was hit, and you can proceed to do whatever work you need to do with this information.

The concept of “item of interest” is more complex than is immediately apparent. It can include a single object, a set of objects, or even a hierarchy of objects. Think creatively about your problem and you may be surprised just how powerful this kind of selection can be.

Definitions

The first concept we must deal with for object selection is the notion of a *selection buffer*. This is an array of unsigned integers (`GLuint`) that will hold the array of *hit records* for a mouse click. In turn, a hit record contains several items as illustrated in Figure 15.1. These include the number of items that were on the *name stack*, the nearest (`zmin`) and farthest (`zmax`) distances to objects on the stack, and the list of names on the name stack for the selection. The distances are integers because they are taken from the Z-buffer, where you may recall that distances are stored as integers in order to make comparisons more effective. The name stack contains the names of all the objects in a hierarchy of named objects that were selected with the mouse click.

The distance to objects is given in terms of the viewing projection environment, in which the nearest points have the smallest non-negative values because this environment has the eye at the origin and distances increase as points move away from the eye. Typical processing will examine each selection record to find the record with the smallest value of `zmin` and will work with the names in that hit record to carry out the work needed by that hit. This work is fairly typical of the handling of any list of variable-length records, proceeding by accumulating the starting points of the individual records (starting with 0 and proceeding by adding the values of `(nitems+3)` from the individual records), with the `zmin` values being offset by 1 from this base and the list of names being offset by 3. This is not daunting, but it does require some care.

In OpenGL, choosing an object by a direct intersection of the object with the pixel identified by the mouse is called *selection*, while choosing an object by clicking near the object is called *picking*. In

order to do picking, then, you must identify points near, but not necessarily exactly on, the point where the mouse clicks. This is discussed toward the end of this note.

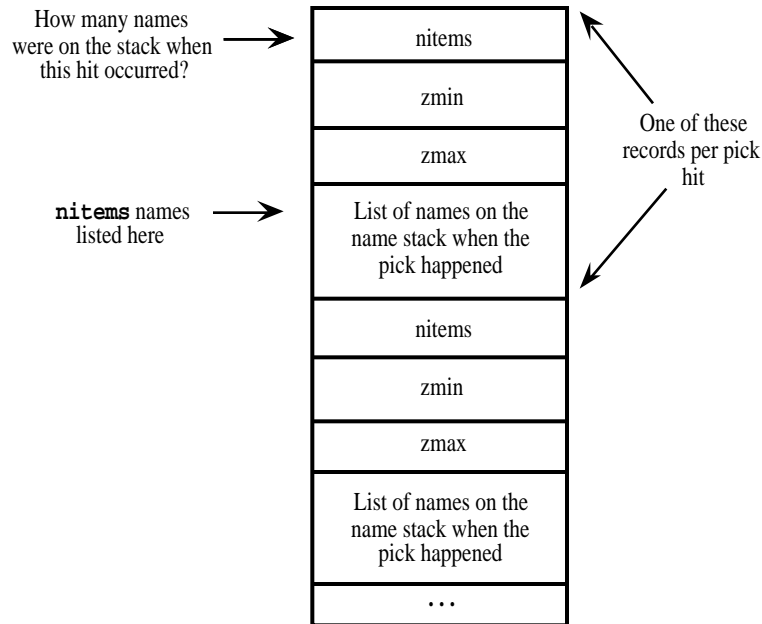


Figure 15.1: the structure of the selection buffer

Making selection work

The selection or picking process is fairly straightforward. The function `glRenderMode(mode)` allows you to draw in either of two modes: render mode (`GL_RENDER`) invokes the graphics pipeline and produces pixels in the frame buffer, and select mode (`GL_SELECT`) calculates the pixels that would be drawn if the graphics pipeline were to be invoked, and tests the pixels against the pixels that were identified by the mouse click. As illustrated in the example below, the mouse function can be defined to change the drawing mode to `GL_SELECT` and to post a redisplay operation. The display function can then draw the scene in select mode with selection object names defined with `glutLoadName(int)` to determine what name will be put into the selection buffer if the object includes the selected pixel, noting that the mode can be checked to decide what is to be drawn and/or how it is to be drawn, and then the selection buffer can be examined to identify what was hit so the appropriate processing can be done. After the selection buffer is processed, the scene can be displayed again in render mode to present the effect of the selection.

In the outline above, it sounds as though the drawing in select mode will be the same as in render mode. But this is usually not the case; anything that you don't want the user to be able to select should not be drawn at all in select mode. Further, if you have a complex object that you want to make selectable, you may not want to do all the work of a full rendering in select mode; you need only design an approximation of the object and draw that. You can even select things that aren't drawn in render mode by drawing them in select mode. Think creatively and you can find that you can do interesting things with selection.

It's worth a word on the notion of selection names. You cannot load a new name inside a `glBegin(mode)/glEnd()` pair, so if you use any geometry compression in your object, it must all be within a single named object. You can, however, nest names with the *name stack*, using the `glPushName(int)` function so that while the original name is active, the new name is

also active. For example, the code below creates a hierarchy of selections for an automobile (“Jaguar”) and for various parts of the auto (“body”, “tire”, etc.)

```
glLoadName( JAGUAR );
glPushName( BODY );
    glCallList( JagBodyList );
glPopName();
glPushName( FRONT_LEFT_TIRE );
    glPushMatrix();
    glTranslatef( ??, ??, ?? );
    glCallList( TireList );
    glPopMatrix();
glPopName();
glPushName( FRONT_RIGHT_TIRE );
    glPushMatrix();
    glTranslatef( ??, ??, ?? );
    glCallList( TireList );
    glPopMatrix();
glPopName();
```

When a selection occurs, then, the selection buffer will include the automobile as well as the lower-level part, and you can choose which of the selections you want to use.

Picking

Picking is almost the same operation, logically, as selection, but we present it separately because it uses a different process and allows us to define a concept of “near” and to talk about a way to identify the objects near the selection point. In the picking process, you can define a very small window in the immediate neighborhood of the point where the mouse was clicked, and then you can identify everything that is drawn in that neighborhood. The result is returned in the same selection buffer and can be processed in the same way.

This is done by creating a transformation with the function `gluPickMatrix(...)` that is applied after the projection transformation (that is, defined before the projection; recall the relation between the sequence in which transformations are identified and the sequence in which they are applied). The full function call is

```
gluPickMatrix(GLdouble x, GLdouble y, GLdouble width, GLdouble height,
              GLint viewport[4])
```

where `x` and `y` are the coordinates of the point picked by the mouse, which is the center of the picking region; the width and height are the size of the picking region in pixels, sometimes called the pick tolerance; and the viewport is the vector of four integers returned by the function call `glGetIntegerv(GL_VIEWPORT, GLint *viewport)`.

The function of this pick matrix is to identify a small region centered at the point where the mouse was clicked and to select anything that is drawn in that region. This returns a standard selection buffer that can then be processed to identify the objects that were picked, as described above.

A code fragment to implement this picking is given below. This corresponds to the point in the code for `doSelect(...)` above labeled “set up the standard viewing model” and “standard perspective viewing”:

```
int viewport[4]; /* place to retrieve the viewport numbers */
...
dx = glGet( GLUT_WINDOW_WIDTH );
dy = glGet( GLUT_WINDOW_HEIGHT );
...
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
```

```

if( RenderMode == GL_SELECT ) {
    glGetIntegerv( GL_VIEWPORT, viewport );
    gluPickMatrix( (double)Xmouse, (double)(dy - Ymouse),
        PICK_TOL, PICK_TOL, viewport );
}
... the call to glOrtho(), glFrustum(), or gluPerspective() goes here

```

A selection example

The selection process is pretty well illustrated by some code by a student, Ben Eadington. This code sets up and renders a Bézier spline surface with a set of selectable control points. When an individual control point is selected, that point can be moved and the surface responds to the adjusted set of points. An image from this work is given in Figure 15.2, with one control point selected (shown as being a red cube instead of the default green).

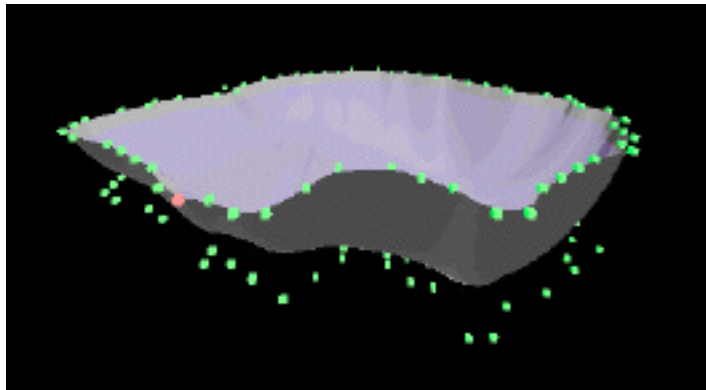


Figure 15.2: a surface with selectable control points and with one selected

Selected code fragments from this project are given below. Here all the data declarations and evaluator work are omitted, as are some standard parts of the functions that are presented, and just the important functions are given with the key points described in these notes. You will be directed to several specific points in the code to illustrate how selection works, described with interspersed text as the functions or code are presented.

In the first few lines you will see the declaration of the global selection buffer that will hold up to 200 values. This is quite large for the problem here, since there are no hierarchical models and no more than a very few control points could ever line up. The actual size would need to be no more than four `GLuint`s per control point selected, and probably no more than 10 maximum points would ever line up in this problem. Each individual problem will need a similar analysis.

```

// globals initialization section
#define MAXHITS 200 // number of GLuints in hit records
// data structures for selection process
GLuint selectBuf[MAXHITS];

```

The next point is the mouse callback. This simply catches a mouse-button-down event and calls the `DoSelect` function, listed and discussed below, to handle the mouse selection. When the hit is handled (including the possibility that there was no hit with the cursor position) the control is passed back to the regular processes with a redisplay.

```

// mouse callback for selection
void Mouse(int button, int state, int mouseX, int mouseY)
{

```

```

    if (state == GLUT_DOWN) { // find which object, if any was selected
        hit = DoSelect((GLint) mouseX, (GLint) mouseY);
    }
    glutPostRedisplay(); /* redraw display */
}

```

The control points may be drawn in either `GL_RENDER` or `GL_SELECT` mode, so this function must handle both cases. The only difference is that names must be loaded for each control point, and if any of the points had been hit previously, it must be identified so it can be drawn in red instead of in green. But there is nothing in this function that says what is or is not hit in another mouse click; this is handled in the `DoSelect` function below.

```

void drawpoints(GLenum mode)
{
    int i, j;
    int name=0;
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, green);
    // iterate through control point array
    for(i=0; i<GRIDSIZE; i++)
        for(j=0; j<GRIDSIZE; j++) {
            if (mode == GL_SELECT) {
                glLoadName(name); // assign a name to each point
                name++;           // increment name number
            }
            glPushMatrix();
            ... place point in right place with right scaling
            if(hit==i*16+j*16) { // selected point, need to draw it red
                glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, red);
                glutSolidCube(0.25);
                glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, green);
            }
            else glutSolidCube(0.25);
            glPopMatrix();
        }
}

```

The only real issue here is to decide what you do and do not need to draw in each of the two rendering modes. Note that the surface is only drawn if the program is in `GL_RENDER` mode; because nothing in the surface is itself selectable, the only thing that needs to be drawn in `GL_SELECT` mode is the control points.

```

void render(GLenum mode) {
    ... do appropriate transformations
    if (mode == GL_RENDER) { // don't render surface if mode is GL_SELECT
        surface(ctrlpts);
        ... some other operations that don't matter here
    }
    if(points) drawpoints(mode); // always render the control points
    ... pop the transform stack as needed and exit gracefully
}

```

This final function is the real meat of the problem. The display environment is set up (projection and viewing transformations), the `glRenderMode` function sets the rendering mode to `GL_SELECT` and the image is drawn in that mode, the number of hits is returned from the call to the `glRenderMode` function when it returns to `GL_RENDER` mode, the display environment is rebuilt for the next drawing, and the selection buffer is scanned to find the object with the smallest `zmin` value as the selected item. That value is then returned so that the `drawpoints` function

will know which control point to display in red and so other functions will know which control point to adjust.

```
GLint DoSelect(GLint x, GLint y)
{
    int i;
    GLint hits, temphit;
    GLuint zval;

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

    // set up the viewing model
    ... standard perspective viewing and viewing transformation setup

    render(GL_SELECT); // draw the scene for selection

    // find the number of hits recorded and reset mode of render
    hits = glRenderMode(GL_RENDER);
    // reset viewing model
    ... standard perspective viewing and viewing transformation setup
    // return the label of the object selected, if any
    if (hits <= 0) return -1;
    else {
        zval = selectBuf[1];
        temphit = selectBuf[3];
        for (i = 1; i < hits; i++) { // for each hit
            if (selectBuf[4*i+1] < zval) {
                zval = selectBuf[4*i+1];
                temphit = selectBuf[4*i+3];
            }
        }
    }
    return temphit;
}
```

A word to the wise...

This might be a good place to summarize the things we've seen in the discussions and code examples above:

- Define an array of unsigned integers to act as the selection buffer
- Design a mouse event callback that calls a function that does the following:
 - Sets `GL_SELECT` mode and draws selected parts of the image, having loaded names so these parts can be identified when the selection is made
 - when this rendering is completed, returns a selection buffer that can be processed
 - returns to `GL_RENDER` mode.

This design structure is straightforward to understand and can be easily implemented with a little care and planning.

Another point to recognize is that you cannot pick raster characters. For whatever reason, if you draw any raster characters in select mode, OpenGL will always think that the characters were picked no matter where you clicked. If you want to be able to pick a word that is drawn as raster characters, create a rectangle that occupies the space where the raster characters would be, and draw that rectangle in select mode.

Code examples

- `pool.c` — Ben Eadington's source for interactive control point manipulation