Chapter 14: 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

In the earlier chapter on interaction, we saw many ways that a graphics API would support user inputs into your images through menus, keystrokes, and mouse functions. If you wanted to identify one of the objects in the scene to act on, you could use those tools to identify an object by name, but you could not identify an object simply by clicking on it. In this chapter we will show you how you can make that kind of selection, or to use the term that’s most common in graphics, we will show you how to pick an object in a scene. This kind of object picking permits the user to interact with a scene in a much more direct way than is possible with the kind of external events, such as menu selections, mouse clicks, or key presses, that we saw in the earlier chapter on event handling.

With object picking we can get the kind of direct manipulation that we are familiar with from graphical user interfaces, where the user selects a graphical object and then applies operations to it. Conceptually, picking allows your user to identify a particular object with the cursor and to choose it by clicking the mouse button when the cursor is on the object. The program must be able to identify what was selected, and then must have the ability to apply whatever action the user chooses to that particular selected object.

To understand how picking works, let’s start with the mouse click. When you get a mouse event, the event callback gets four pieces of information: the button that was clicked, the state of that button, and the integer coordinates of the point in the window where the event happened. In order to find out what object might be indicated by the click, we convert the window coordinates to 2D eye coordinates but we must then reverse the projection and go back from 2D eye space to 3D eye space. However, a single point in 2D eye space becomes a line in 3D eye space, as shown in Figure 14.1. Our problem then becomes how to identify what objects meet this line segment and which of those objects was chosen by the user.

![Figure 14.1: the line in the view volume that corresponds to a point in 2D eye space](image)

It is possible to make this computation directly from the geometry using collision detection logic. That is, for each object in the scene, we calculate whether or not the line and object intersect. When we have done all these calculations, we can say which objects lie under the point that was chosen and where each of those intersections takes place in 3D eye space. We may then choose the
closest, which is be the one that the user would see at the place the mouse click was made, or we may choose any other intersection that our logic demands. The problem with this is that it is very computation intensive and requires us to be able to go back to eye space, and that makes it difficult to implement.

Another way to identify which objects lie on the line is to invert this logic completely. Instead of focusing on the objects that lie under the line of the pick, we see that any object that we might have chosen will use the pixel that was chosen when it was rendered. So if we can keep track of that pixel and save information on any object that includes it, we can identify all the objects that lie under the pick point. Because the rendering process keeps track of depth values, we can also get information on the depth of the object in the view volume when the pixel is used. Note that we do not talk about what kind of information is saved, so there are some interesting opportunities to think about what might be done here.

We cannot say which of these techniques might be used by any particular graphics application, and we cannot say which might be used by a particular graphics API. But OpenGL uses the second technique, and in the rest of this chapter we discuss how that is done.

**Picking in OpenGL**

OpenGL has several ways to identify objects that correspond to mouse events, and we will discuss two of them in this chapter. One of these involves drawing invisibly and keeping track of objects that include a given pixel or small region around a pixel; this is what we will call the standard selection approach. The other involves drawing with synthetic colors that are unique to each object and looking at the color of the selected pixel in the color buffer to identify the nearest object at the picked point. We will discuss the standard selection approach first.

The standard selection approach calls for the mouse event to request that you render your scene invisibly so a record may be made of all the objects that are selected. This approach introduces the concept of the **render mode** for drawing. In your standard rendering, you draw the scene in GL_RENDER mode, which is the default drawing mode. In the mouse event callback that is executed after the mouse event, you change your rendering to GL_SELECT mode and re-draw the scene with each item of interest given a unique name. When the scene is rendered in GL_SELECT mode, nothing is actually changed in the frame buffer but the pixels that would be rendered are identified. When any named object is found that would include the pixel selected by the mouse, that object's name is added to a selection buffer data structure, actually a stack of unsigned integers, that is maintained for that name. This selection buffer holds information on all the items in a hierarchy of named items that were hit. When the rendering of the scene in GL_SELECT mode is finished, a list of hit records is produced, with one entry for each name of an object whose rendering included the mouse click point, and the number of such records is returned when the system is returned to GL_RENDER mode. The structure of these hit records is described below. You can then process this list to identify the items that were hit, including the distance from the eye where the hit occurred, and you can proceed to do whatever work you need 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 14.2. These include the number
of items that were on the *name stack*, the nearest (\(z_{\text{min}}\)) and farthest (\(z_{\text{max}}\)) 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 \(z_{\text{min}}\) 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 \((n_{\text{items}}+3)\) from the individual records), with the \(z_{\text{min}}\) 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.

![Diagram](image.png)

**Figure 14.2:** The structure of the selection buffer

In OpenGL, we have two ways to pick an object. We can generate the scene (partly or fully) and record the intersection of the object with the pixel identified by the mouse, or we can set up an additional projection with the pick matrix and clip everything that is not within a certain distance of the pixel we chose. The first of these methods is perhaps simpler because it does not involve any changes in the basic rendering, but the second is faster because it uses clipping to avoid rendering anything except those items very near the pixel. We will begin by discussing the simpler case, and describe the use of the pick matrix later.

Before we go on, however, we must be sure you understand one key point: you don’t do anything more to generate the selection buffer than to draw the scene in select mode. That’s it. The system does all the rest of the work for you, putting the active name (or all the names on the active name stack) into the selection buffer whenever the drawing process operates on the selected pixel or the object is not clipped away by the pick matrix. You simply generate the scene in select mode, get
the number of hits from the returned value of the \texttt{glRenderMode(...)} function when you call it to restore \texttt{GL\_RENDER} mode, and process the selection buffer to handle that many hits.

\textit{Making picking work}

The picking process is fairly straightforward. The function \texttt{glRenderMode(mode)} allows you to draw in either of two modes: render mode (\texttt{GL\_RENDER}) invokes the graphics pipeline and produces pixels in the frame buffer, and select mode (\texttt{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 \texttt{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 \texttt{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 often not the case; there are some techniques to make the select mode drawing work more quickly or more flexibly than render-mode drawing. These include:

\begin{itemize}
  \item if there are any objects that you don’t want the user to be able to select, do not draw these at all in select mode. Because they are not drawn, they are invisible to picking.
  \item if you want to allow a pick of a complex object, you need not do all the work of a full rendering of the object in select mode; you need only design an approximation of the object and draw that.
  \item you can even create invisible controls by allowing the user to pick things that are only drawn in select mode but not in render mode.
\end{itemize}

Think creatively and you can find that you can do interesting things with selection. In fact, you will sometimes find that you must use some of these techniques. For example, if you want the user to be able to select a wireframe object, you probably want to replace the wireframe version by a solid version in the select mode drawing, because a user will visualize the spaces in the wireframe as part of the object but OpenGL will not. Another important use is in selecting text, because you cannot pick raster characters in OpenGL. 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.

It’s worth a word on the notion of selection names. You cannot load a new name inside a \texttt{glBegin(mode)}-\texttt{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 \texttt{name stack}, using the \texttt{glPushName(int)} function so that while the original name is active, the new name is also active. For example, suppose we were dealing with automobiles, and suppose that we wanted someone to select parts for an automobile. We could permit the user to select parts at a number of levels; for example, to select an entire automobile, the body of the automobile, or simply one of the tires. In the code below, we create a heirarchy of selections for an automobile (“Jaguar”) and for various parts of the auto (“body”, “tire”, etc.) In this case, the names JAGUAR, BODY, FRONT\_LEFT\_TIRE, and FRONT\_RIGHT\_TIRE are symbolic names for integers that are defined elsewhere in the code.

\begin{verbatim}
  glLoadName( JAGUAR );
  glPushName( BODY );
  glCallList( JagBodyList );
\end{verbatim}
When a selection occurs, then, the selection buffer will include everything whose display involved
the pixel that was chosen, including the automobile as well as the lower-level part. For example, if
you selected the right front tire of the automobile, nitems would be 3 and your hit record would
include three names: the names FRONT_LEFT_TIRE, BODY, and JAGUAR. Your program then
would know that a hierarchy was selected that had these three parts and could choose (or allow the
user to choose) which selection or what other logic it needed to use.

There are a couple of things to watch out for in the name stack. The first is that the name stack is
empty when it is initialized, so you cannot simply load a name into the stack; this will generate an
error. Instead you must push some name onto the stack so that you can load a name to replace it.
The second thing to watch for with the name stack is that loading a name only replaces the top
name on the name stack. If you have finished a hierarchy and need to remove the entire hierarchy
from the name stack, you will need to pop the name stack until there is a single name left; you can
then load the new name and replace that single name. But these are straightforward to remember as
you use the name stack.

The pick matrix

Picking using the pick matrix is almost the same operation, logically, as picking using the selected
pixel, 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

```c
    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

```c
    glGetInteger(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. The picking process 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”:

```c
int viewport[4]; /* place to retrieve the viewport numbers */
...  
dx = glutGet( GLUT_WINDOW_WIDTH );
dy = glutGet( GLUT_WINDOW_HEIGHT );
...
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
if( RenderMode == GL_SELECT ) {
  gluGetIntegerv( GL_VIEWPORT, viewport );
  gluPickMatrix( (double)Xmouse, (double)(dy - Ymouse),
  PICK_TOL, PICK_TOL, viewport );
}
... call glOrtho(), glFrustum(), or gluPerspective() here
```

*Using the back color buffer to do picking*

There is another approach you can use that avoids the selection buffer entirely by using some of the facilities of double-buffered drawing. In this approach, when you want to permit a selection, you continue to draw your scene in render mode but you draw your scene into the back buffer in a unique way. Here you can identify the objects you want to make selectable by giving each a unique color, you can use a proxy for an object so that you draw the alternate representation we discussed above, and you can omit any objects you don’t want to draw. When the mouse event happens and the mouse callback gets the pixel location of the pick, you look in the back buffer to see what color is at that pixel location. That color can be used to identify the object drawn at that position, giving you your picked object. If you were drawing with depth test enabled, you will get the object that is closest to the eye at that point. If you were not, you will get the last object drawn to that point. But you can control which of these you want to happen. And after you have gotten the information you need from the back buffer, you simply don’t swap it with the front buffer but let the next (and probably normal) drawing operation replace the artificial-color image you have just created.

The mechanics of this are pretty straightforward. After the back buffer has been filled with the artificial image, select the back buffer to be read with the `glReadBuffer(GL_BACK)` function (although the back buffer is the default buffer for reading in double-buffered mode). Then use the `glReadPixels(...)` function to read a 1x1 array of color pixels (that is, the value of the color at a single point) at the position of the selection, and use whatever logic you created in determining the color of each object to identify the object you have just selected. This is a straightforward technique but it may require some thought to make a reasonable set of color identifications if you have a large number of objects.

*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 14.3, with one control point selected (shown as being a red cube instead of the default green color).
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 GLuints 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 redraw.

```
// mouse callback for selection
void Mouse(int button, int state, int mouseX, int mouseY)  
{  
    if (state == GLUT_DOWN) { // find which object 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

... 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);

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();
    glPushMatrix();

    // set up the viewing model

... standard perspective viewing and viewing transformation setup

render(GL_SELET);  // 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 about the standard picking process
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.
• Be careful to manage your name stack as you draw objects in GL_SELECT mode so that
  you have exactly the things on the stack you need to identify the objects you want the user
  to be able to pick.
This design structure is straightforward to understand and can be easily implemented with a little

Questions

1. Imagine, or lay out, some collection of objects in a space; an example might be the carousel
   model you created in Chapter 2. Suppose you only want part of these objects to be selectable
   (for example, the posts in the carousel, but not the carousel animals). Describe how you could
   define your selection process to make this happen.

2. Discuss the differences between picking and selection in terms of efficiency and of ease of
   identifying objects of various sizes. What advantages or disadvantages does each approach
   have?

3. Both selection and pick processes require you to analyze the selection buffer to identify which
   objects are closest, farthest, or have another relationship to the eye point. Why does the back
   buffer approach find the object nearest the eye without any further work? Is there any way that
   the back buffer could find any other object?
Exercises

4. Examine the nature of hit records by modifying any program including selection to include code to dump the selection buffer byte-by-byte into a file, and examining that file by a simple file dump utility such as Unix’s `od`. Identify all the components of the selection buffer within this byte array and see how these components are really arranged.

5. Do the previous exercise when several objects are grouped in one name; when objects are arranged in a hierarchy. These should give more complex lists of the names on the name stack when the selection is made; break these down to understand how grouping and hierarchy work.

Experiments

6. Recall from Chapter 1 on projection that you calculated the parametric equation of a line segment in the viewing frustum that represents the points in 3D eye space that project to a single screen point. Define a number of simple sphere and polygon primitives that lie in the visible part of a space and choose a point on the front viewing plane. Calculate the intersection of the resulting line with each of the primitives and explore the way you could tell which is nearest the eye.

7. Experiment with the use of the back buffer for picking by setting your select-mode rendering to draw different objects in different colors and identify what object is nearest the eye at any given screen point.