PEARSON NEW INTERNATIONAL EDITION

Computer Graphics with Open GL

Hearn Baker Carithers

Fourth Edition

ALWAYS LEARNING

PEARSON

Pearson New International Edition

Computer Graphics with Open GL
Hearn Baker Carithers
Fourth Edition

PEARSON

In addition, at any time, we can query the system to determine which window is the current display window:

```
currentWindowID = glutGetWindow ( );
```

A value of 0 is returned by this function if there are no display windows or if the current display window was destroyed.

Relocating and Resizing a GLUT Display Window

We can reset the screen location for the current display window with

```
glutPositionWindow (xNewTopLeft, yNewTopLeft);
```

where the coordinates specify the new position for the upper-left display-window corner, relative to the upper-left corner of the screen. Similarly, the following function resets the size of the current display window:

```
glutReshapeWindow (dwNewWidth, dwNewHeight);
```

With the following command, we can expand the current display window to fill the screen:

```
glutFullScreen ( );
```

The exact size of the display window after execution of this routine depends on the window-management system. A subsequent call to either `glutPositionWindow` or `glutReshapeWindow` will cancel the request for an expansion to full-screen size.

Whenever the size of a display window is changed, its aspect ratio may change and objects may be distorted from their original shapes. We can adjust for a change in display-window dimensions using the statement

```
glutReshapeFunc (winReshapeFcn);
```

This GLUT routine is activated when the size of a display window is changed, and the new width and height are passed to its argument: the function `winReshapeFcn`, in this example. Thus, `winReshapeFcn` is the "callback function" for the "reshape event." We can then use this callback function to change the parameters for the viewport so that the original aspect ratio of the scene is maintained. In addition, we could also reset the clipping-window boundaries, change the display-window color, adjust other viewing parameters, and perform any other tasks.

Managing Multiple GLUT Display Windows

The GLUT library also has a number of routines for manipulating a display window in various ways. These routines are particularly useful when we have multiple display windows on the screen and we want to rearrange them or locate a particular display window.

We use the following routine to convert the current display window to an icon in the form of a small picture or symbol representing the window:

```
glutIconifyWindow ( );
```

The label on this icon will be the same name that we assigned to the window, but we can change this with the following command:

```
glutSetIconTitle ("Icon Name");
```

We also can change the name of the display window with a similar command:

```
glutSetWindowTitle ("New Window Name");
```

With multiple display windows open on the screen, some windows may overlap or totally obscure other display windows. We can choose any display window to be in front of all other windows by first designating it as the current window, and then issuing the "pop-window" command:

```
glutSetWindow (windowID);
glutPopWindow ( );
```

In a similar way, we can "push" the current display window to the back so that it is behind all other display windows. This sequence of operations is

```
glutSetWindow (windowID);
glutPushWindow ( );
```

We can also take the current window off the screen with

```
glutHideWindow ( );
```

In addition, we can return a "hidden" display window, or one that has been converted to an icon, by designating it as the current display window and then invoking the function

```
glutShowWindow ( );
```

GLUT Subwindows

Within a selected display window, we can set up any number of second-level display windows, which are called *subwindows*. This provides a means for partitioning display windows into different display sections. We create a subwindow with the following function:

```
glutCreateSubWindow (windowID, xBottomLeft, yBottomLeft,
                        width, height);
```

Parameter `windowID` identifies the display window in which we want to set up the subwindow. With the remaining parameters, we specify the subwindow's size and the placement of its lower-left corner relative to the lower-left corner of the display window.

Subwindows are assigned a positive integer identifier in the same way that first-level display windows are numbered, and we can place a subwindow inside another subwindow. Also, each subwindow can be assigned an individual display mode and other parameters. We can even reshape, reposition, push, pop, hide, and show subwindows, just as we can with first-level display windows. But we cannot convert a GLUT subwindow to an icon.

Selecting a Display-Window Screen-Cursor Shape

We can use the following GLUT routine to request a shape for the screen cursor that is to be used with the current window:

```
glutSetCursor (shape);
```

The possible cursor shapes that we can select are an arrow pointing in a chosen direction, a bidirectional arrow, a rotating arrow, a crosshair, a wristwatch, a question mark, or even a skull and crossbones. For example, we can assign the symbolic constant GLUT_CURSOR_UP_DOWN to parameter shape to obtain an up-down arrow. A rotating arrow is chosen with GLUT_CURSOR_CYCLE, a wristwatch shape is selected with GLUT_CURSOR_WAIT, and a skull and crossbones is obtained with the constant GLUT_CURSOR_DESTROY. A cursor shape can be assigned to a display window to indicate a particular kind of application, such as an animation. However, the exact shapes that we can use are system dependent.

Viewing Graphics Objects in a GLUT Display Window

After we have created a display window and selected its position, size, color, and other characteristics, we indicate what is to be shown in that window. If more than one display window has been created, we first designate the one we want as the current display window. Then we invoke the following function to assign something to that window:

```
glutDisplayFunc (pictureDescrip);
```

The argument is a routine that describes what is to be displayed in the current window. This routine, called pictureDescrip for this example, is referred to as a *callback function* because it is the routine that is to be executed whenever GLUT determines that the display-window contents should be renewed. Routine pictureDescrip usually contains the OpenGL primitives and attributes that define a picture, although it could specify other constructs such as a menu display.

If we have set up multiple display windows, then we repeat this process for each of the display windows or subwindows. Also, we may need to call glutDisplayFunc after the glutPopWindow command if the display window has been damaged during the process of redisplaying the windows. In this case, the following function is used to indicate that the contents of the current display window should be renewed:

```
glutPostRedisplay ( );
```

This routine is also used when an additional object, such as a pop-up menu, is to be shown in a display window.

Executing the Application Program

When the program setup is complete and the display windows have been created and initialized, we need to issue the final GLUT command that signals execution of the program:

```
glutMainLoop ( );
```

At this time, display windows and their graphic contents are sent to the screen. The program also enters the **GLUT processing loop** that continually checks for new "events," such as interactive input from a mouse or a graphics tablet.

Other GLUT Functions

The GLUT library provides a wide variety of routines to handle processes that are system dependent and to add features to the basic OpenGL library. For example, this library contains functions for generating bitmap and outline characters, and it provides functions for loading values into a color table. In addition, some GLUT functions are available for displaying three-dimensional objects, either as solids or in a wireframe representation. These objects include a sphere, a torus, and the five regular polyhedra (cube, tetrahedron, octahedron, dodecahedron, and icosahedron).

Sometimes it is convenient to designate a function that is to be executed when there are no other events for the system to process. We can do that with

```
glutIdleFunc (function);
```

The parameter for this GLUT routine could reference a background function or a procedure to update parameters for an animation when no other processes are taking place.

We also have GLUT functions for obtaining and processing interactive input and for creating and managing menus. Individual routines are provided by GLUT for input devices such as a mouse, keyboard, graphics tablet, and spaceball.

Finally, we can use the following function to query the system about some of the current state parameters:

```
glutGet (stateParam);
```

This function returns an integer value corresponding to the symbolic constant we select for its argument. For example, we can obtain the x-coordinate position for the top-left corner of the current display window, relative to the top-left corner of the screen, with the constant `GLUT_WINDOW_X`; and we can retrieve the current display-window width or the screen width with `GLUT_WINDOW_WIDTH` or `GLUT_SCREEN_WIDTH`.

OpenGL Two-Dimensional Viewing Program Example

As a demonstration of the use of the OpenGL viewport function, we use a split-screen effect to show two views of a triangle in the xy plane with its centroid at the world-coordinate origin. First, a viewport is defined in the left half of the display window, and the original triangle is displayed there in blue. Using the same clipping window, we then define another viewport for the right half of the display window, and the fill color is changed to red. The triangle is then rotated about its centroid and displayed in the second viewport.

```
#include <GL/glut.h>

class wcPt2D {
   public:
      GLfloat x, y;
};
```

```
void init (void)
{
   /*  Set color of display window to white.  */
   glClearColor (1.0, 1.0, 1.0, 0.0);

   /*  Set parameters for world-coordinate clipping window.  */
   glMatrixMode (GL_PROJECTION);
   gluOrtho2D (-100.0, 100.0, -100.0, 100.0);

   /*  Set mode for constructing geometric transformation matrix.  */
   glMatrixMode (GL_MODELVIEW);
}

void triangle (wcPt2D *verts)
{
   GLint k;

   glBegin (GL_TRIANGLES);
      for (k = 0; k < 3; k++)
         glVertex2f (verts [k].x, verts [k].y);
   glEnd ( );
}

void displayFcn (void)
{
   /*  Define initial position for triangle.  */
   wcPt2D verts [3] = { {-50.0, -25.0}, {50.0, -25.0}, {0.0, 50.0} };

   glClear (GL_COLOR_BUFFER_BIT);   //  Clear display window.

   glColor3f (0.0, 0.0, 1.0);       //  Set fill color to blue.
   glViewport (0, 0, 300, 300);     //  Set left viewport.
   triangle (verts);                //  Display triangle.

   /*  Rotate triangle and display in right half of display window.  */
   glColor3f (1.0, 0.0, 0.0);         //  Set fill color to red.
   glViewport (300, 0, 300, 300);     //  Set right viewport.
   glRotatef (90.0, 0.0, 0.0, 1.0);   //  Rotate about z axis.
   triangle (verts);           //  Display red rotated triangle.

   glFlush ( );
}
void main (int argc, char ** argv)
{
   glutInit (&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowPosition (50, 50);
   glutInitWindowSize (600, 300);
   glutCreateWindow ("Split-Screen Example");

   init ( );
   glutDisplayFunc (displayFcn);

   glutMainLoop ( );
}
```

5 Clipping Algorithms

Generally, any procedure that eliminates those portions of a picture that are either inside or outside a specified region of space is referred to as a **clipping algorithm** or simply **clipping.** Usually a clipping region is a rectangle in standard position, although we could use any shape for a clipping application.

The most common application of clipping is in the viewing pipeline, where clipping is applied to extract a designated portion of a scene (either two-dimensional or three-dimensional) for display on an output device. Clipping methods are also used to antialias object boundaries, to construct objects using solid-modeling methods, to manage a multiwindow environment, and to allow parts of a picture to be moved, copied, or erased in drawing and painting programs.

Clipping algorithms are applied in two-dimensional viewing procedures to identify those parts of a picture that are within the clipping window. Everything outside the clipping window is then eliminated from the scene description that is transferred to the output device for display. An efficient implementation of clipping in the viewing pipeline is to apply the algorithms to the normalized boundaries of the clipping window. This reduces calculations, because all geometric and viewing transformation matrices can be concatenated and applied to a scene description before clipping is carried out. The clipped scene can then be transferred to screen coordinates for final processing.

In the following sections, we explore two-dimensional algorithms for

- Point clipping
- Line clipping (straight-line segments)
- Fill-area clipping (polygons)
- Curve clipping
- Text clipping

Point, line, and polygon clipping are standard components of graphics packages. But similar methods can be applied to other objects, particularly conics, such as circles, ellipses, and spheres, in addition to spline curves and surfaces. Usually, however, objects with nonlinear boundaries are approximated with straight-line segments or polygon surfaces to reduce computations.

Unless otherwise stated, we assume that the clipping region is a rectangular window in standard position, with boundary edges at coordinate positions $xw_{\min}$, $xw_{\max}$, $yw_{\min}$, and $yw_{\max}$. These boundary edges typically correspond to a normalized square, in which the x and y values range either from 0 to 1 or from -1 to 1.

6 Two-Dimensional Point Clipping

For a clipping rectangle in standard position, we save a two-dimensional point $\mathbf{P} = (x, y)$ for display if the following inequalities are satisfied:

$$\begin{aligned} xw_{\min} \le x \le xw_{\max} \\ yw_{\min} \le y \le yw_{\max} \end{aligned} \tag{12}$$

If any of these four inequalities is not satisfied, the point is clipped (not saved for display).

Although point clipping is applied less often than line or polygon clipping, it is useful in various situations, particularly when pictures are modeled with particle systems. For example, point clipping can be applied to scenes involving