Chapter 2

Two-Dimensional Programming in OpenGL

OpenGL is a three-dimensional system. From an application programmer's perspective, OpenGL primitives describe three-dimensional objects that exist in a three-dimensional world. But some objects reside in a plane, and these two-dimensional problems are a special case of three-dimensional problems. This special case is important because many applications are two dimensional. For these applications, it is easier to work directly in two dimensions, something that OpenGL allows. We start with these simpler problems as a straightforward way of getting started with OpenGL.

In this chapter, we start by dissecting a very simple program to understand the basics of an OpenGL program. Then we enhance the program, introducing additional OpenGL functions. Finally, we introduce the full set of basic OpenGL primitives.

2.1 A Simple Program

Program 2.1 draws a white rectangle on a black background. Although it makes heavy use of default values for many parameters, the program nonetheless illustrates the structure of most OpenGL programs.

The program consists of two functions: `main()` and `display()`. The `main()` function initializes OpenGL, and `display()` defines the graphical entity to be drawn. First, we examine `main()`. Although OpenGL contains no input or window commands, any user program must interact with the window system, and interactive programs have input from such devices as the mouse and the keyboard. However, the user interface to window systems is system dependent. A program for Windows 98/NT differs from one with the same functionality written

```
/*simple.c */

#include <GL/glut.h>

void display()

{

     glClear(GL_COLOR_BUFFER_BIT);

     glBegin(GL_POLYGON);
          glVertex2f(-0.5, -0.5);
          glVertex2f(-0.5, 0.5);
          glVertex2f(0.5, 0.5);
          glVertex2f(0.5, -0.5);
     glEnd();

     glFlush();

}


int main(int argc, char** argv)
{

     glutInit(&argc,argv);
     glutCreateWindow("simple");
     glutDisplayFunc(display);
     glutMainLoop();

}
```

Program 2.1 simple.c

for the X Window System under Linux. We can interface with these systems by using a minimum amount of "glue" contained in system-specific libraries: `GLX` for X Windows, `wgl` for Windows, `agl` for the Macintosh. There is an alternative path through the GLUT library.

2.2 GLUT

The OpenGL Utility Toolkit (GLUT) is a library of functions that are common to virtually all modern windowing systems. GLUT has been implemented on all the popular systems, so programs written using the GLUT API for windowing and

input can be compiled with the source code unchanged on all these systems. We use GLUT throughout this book.

```
void glutInit(int argc, char **argv)
```

Initializes GLUT and should be called before any OpenGL functions: `glutInit()` takes the arguments from `main()` and can use them in an implementation-dependent manner.

Starting with the first function in `main()`, `glutInit()`, we see that the name for all GLUT functions begins with the letters `glut`. Although we can pass in command line arguments from `main()`, their interpretation within GLUT is implementation dependent. We shall not use implementation-dependent arguments in our examples. The function `glutCreateWindow()` puts a window on the screen in a default position—at the upper-left corner—and at a default size—300×300 pixels. The argument allows us to put a title on the top border of the window.

```
int glutCreateWindow(*char title)
```

Creates a window on the screen with the title given by the argument. The function returns an integer that can be used to refer to the window in multiwindow situations.

Altering the defaults in GLUT is discussed later.

2.3 Event Loops and Callback Functions

Most interactive programs are based on the program's reacting to a variety of discrete **events**. Events include mouse events, such as moving the mouse or clicking a mouse button; keyboard events, such as pressing a key; and window events, such as the user resizing a window or the covering up of a window by another window. The programming paradigm used to work with them is to have events handled by the operating systems and placed in an **event queue**. Events are processed sequentially from the event queue. The application programmer writes a set of **callback functions** that define how the program should react to specific events. In GLUT, the most common events are recognized. The application program can define its own callback functions, rely on default callbacks for a few events, or do nothing, in which case events without callbacks are ignored.

Program 2.1 contains only a single callback, the **display callback**, which is invoked whenever OpenGL determines that the display has to be redrawn. One such time is when the window is first opened. Consequently, if we put our graphics in the display callback, we can be assured that they will be drawn at least once. Note that the form of the display function, a function with no arguments, is fixed by GLUT.

```
void glutDisplayFunc(void (*func) (void))
```

The function `func()` is called each time there is a display callback.

If we wish to pass values to the display callback function, we must use globals in our programs. After the callbacks have been defined, the program enters the event loop by executing the function `glutMainLoop()`. Once we have entered the loop, we cannot escape except through a callback or an outside intervention, such as pressing a "kill" key. Any code after this call will never be executed.

The form of GLUT callback functions is fixed. Consequently, global variables may be necessary to pass values between functions.

```
void glutMainLoop()
```

Causes the program to enter an event-processing loop. This statement should be the last one in the `main()` function.

2.4 Drawing a Rectangle

Now we have to define our display callback, which we have chosen to name `display`. First, note that we include the file `glut.h`, which is usually stored in a directory named `GL` wherever the standard include files are stored. This file contains the prototypes for the GLUT functions and the `#defines` for a variety of constants that are used in OpenGL programs. This file also contains the following lines to include similar definitions for the OpenGL and GLU functions and constants.

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

The fundamental entity for specifying geometric objects is the **vertex**, a location in space. Simple geometric objects, such as lines and polygons, can be specified through a collection of vertices. OpenGL allows us to work in two, three, or four dimensions through variants of the function `glVertex*()`.

```
void glVertex{234}{sifd}(TYPE xcoordinate, TYPE
    ycoordinate,....)
void glVertex{234}{sifd}v(TYPE *coordinates)
```

Specifies the location of a vertex in two, three, or four dimensions with the types short (`s`), int (`i`), float (`f`), or double (`d`). If `v` is present, `coordinates` is a pointer to an array of the type specified.

We shall use the notation `glVertex*()` to refer to all these variants. Thus, for example, `glVertex2f(x, y)` defines a vertex in two dimensions at the point (`x`, `y`), where `x` and `y` are floats, whereas `glVertex2fv(p)` specifies a vertex at the first two locations of an array of floats, that is, (`p[0]`, `p[1]`).

Because vertices can define a variety of objects, we must tell OpenGL what object a list of vertices defines and where the beginning and the end of the list occur. We make this specification through the functions `glBegin()` and `glEnd()`.

```
void glBegin(GLenum mode)
```

Specifies the beginning of an object of type `mode`. Modes include `GL_POINTS`, `GL_LINES`, and `GL_POLYGON`.

```
void glEnd()
```

Specifies the end of a list of vertices.

Don't forget to include the `glEnd()` after a list of vertices.

Using these three functions, we can define our rectangle as follows:

```
glBegin(GL_POLYGON);
    glVertex2f(-0.5, -0.5);
    glVertex2f(-0.5, 0.5);
    glVertex2f(0.5, 0.5);
    glVertex2f(0.5, -0.5);
glEnd();
```

Note that before we draw the rectangle, we clear the **color buffer**, where OpenGL puts the rendered image through the function `glClear()`.

```
void glClear(GLbitfield mask)
```

Clears all buffers whose bits are set in `mask`. The `mask` is formed by the logical `OR` of values defined in `gl.h`. `GL_COLOR_BUFFER_BIT` refers to the color buffer.

After we finish specifying our one object, we force the renderer to output the results by issuing a `glFlush()`, in case the implementation is buffering commands for efficiency.

```
void glFlush()
```

Forces OpenGL commands to execute.

Figure 2.1 shows the output from our program. Although the program obviously produces an image, we need to address a variety of questions.

- What do we do if we want the image to be a different size?
- What do we do if we want the image to appear in a different place on the screen?
- Why does the white rectangle occupy half the area in the window?

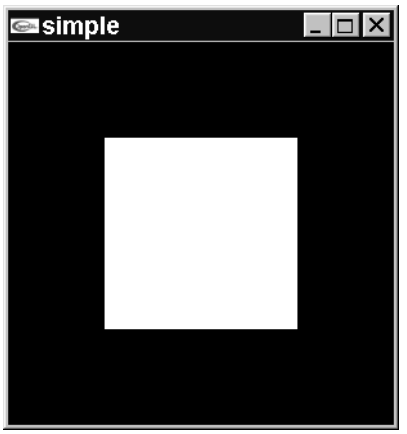

Figure 2.1 Output from `simple.c`

- Why is the background black and the rectangle white? How can we use other colors?
- Can we end the program other than by using the kill box provided by the window system?
- How do we define more complex objects?

2.5 Changing the GLUT Defaults

First, let's add a few GLUT functions that will give us a little finer control over the image that appears on our screen. The functions `glutInitDisplayMode()`, `glutInitWindowSize()`, and `glutInitWindowPosition()` allow us to define what type of window we want, its size, and its position. Generally, an implementation will support a variety of properties that can be associated with a window on the screen. An application program, through the function `glutInitDisplayMode()`, requests the type of window that it requires. The most common window properties to specify are what type of color we wish to use and whether we need double buffering. The defaults in GLUT are RGB color and single buffering, which can be specified explicitly by the function call

```
glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
```

```
void glutInitDisplayMode(unsigned int mode)
```

Requests a display with the properties in `mode`. The values of `mode` are combined by using the logical `OR` of options, such as color model (`GLUT_RGB`, `GLUT_INDEX`) and buffering of color buffers (`GLUT_SINGLE`, `GLUT_DOUBLE`).

The function `glutInitWindowSize()` specifies the size of the window on the screen, and `glutInitWindowPosition()` gives its initial position.

```
void glutInitWindowSize(int width, int height)
```

Specifies the initial height and width, in pixels, of the window on the screen.

```
void glutInitWindowPosition(int x, int y)
```

Specifies the top-left corner of the window, measured in pixels, from the top-left corner of the screen.

2.6 Color in OpenGL

OpenGL supports two colors models: **RGB,** or **RGBA, mode** and **color-index mode**. In RGB mode, each color is a triplet of red, green, and blue values. The eye blends these primary colors, forming the color that we see. If we use real numbers to specify colors, 0.0 is none of a primary, and 1.0 is the maximum amount of that primary. Thus, the RGB triplet (1.0, 0.0, 0.0) is a bright red, (0.5, 0.5, 0.0) is a dark yellow, (1.0, 1.0, 1.0) is white, and (0.0, 0.0, 0.0) is black. In RGBA mode, we use a fourth color component, A, or **alpha**, which is an **opacity**. An opacity of 1.0 means that the color is opaque and cannot be "seen through," whereas a value of 0.0 means that a color is transparent. We will not need opacity until much later. If we use an integer type to specify a color, the range is from 0 to the maximum value of the type. If, for example, we use unsigned bytes, the color values are from 0 to 255.

In color-index mode, colors are specified as indices into a table of red, green, and blue values. In this mode, we form a table of the allowed colors, usually with 256 possible colors. This mode is not the common one used at present and also requires more detailed interaction with the windowing system than does RGB color. Hence, we shall always use RGB or RGBA color.

2.6.1 Setting Colors

In Program 2.1, we used the default values for our colors. The default color for clearing the screen was black and the default drawing color—the color that was used to fill the polygon—was white. These definitions can be changed by the functions `glColor*()` and `glClearColor()`.

```
void glColor3{b i f d ub us ui}(TYPE r, TYPE g, Type b)
void glColor3{b i f d ub us ui}v(TYPE *color)
void glColor4{b i f d ub us ui}(TYPE r, TYPE g, Type b,
    TYPE a)
void glColor4{b i f d ub us ui}(TYPE *color)
```

Specifies RGB and RGBA colors, using the standard types. If the `v` is present, the color is in an array pointed to by `color`.

```
void glClearColor(GLclampf r, GLclampf g, GLclampf b,
    GLclampf a)
```

Specifies the clear color (RGBA) used when clearing the color buffer.

2.6.2 Color and State

In OpenGL, our colors become part of the state. We can think of there being a *present drawing color,* which we set by `glColor*()`, and a *present clear color*, set by `glClearColor()`. These colors remain the same until we change them in the application program. Thus, colors are not attached to objects but rather to the internal state of OpenGL. The color used to render an object is the present color. In the code, it may appear that colors are associated with objects and their vertices. But in fact, OpenGL uses the present state to find the color at the time the program defines a vertex. Application programmers must be very careful about where in the code colors are changed. Later, we shall learn how to bind colors more closely to our objects.

With `glColor*()`, we can set either RGB or RGBA colors, using the standard C data types. OpenGL has only one internal form for the present color, which is in RGBA form. Using `glColor3*()` is the same as using RGBA color with the alpha value set to 1.0. The clear color specified by `glClearColor()` must be specified as an RGBA color, using floats in the range (0.0, 1.0) and values of type `GLclampf`.

Don't lose track of state changes, such as changing colors.

2.7 Coordinate System Differences between GLUT and OpenGL

OpenGL uses a variety of coordinates systems. Generally, users describe their geometry in world coordinates. For two-dimensional applications, this coordinate system has the positive x values increasing to the right and the positive y values increasing as we go up. Thus, if we put the origin on at the bottom-left corner of this page, all the locations the page would have positive x and y values.

Most windowing systems use a system in which the values of y increase as we go down. In such a system, if we want all x and y values to be positive, we put the origin in the top-left corner. In most windowing systems, the screen is displayed from top to bottom, and the counting of rows and columns starts from the top-left corner. Because it interacts with the window system, GLUT uses the second form, and we should think of the origin of the screen as being in its top-left corner and the locations of the pixels as numbered from (0, 0) going down and to the right. For such functions as `glutInitWindowPosition()`, there should be little difficulty. Later, when we use input from the mouse, we will have to work with values in both systems, which can cause some confusion.

For two-dimensional problems, the directions of positive increments in x and y in OpenGL are to the right and up. For input functions used in GLUT and windowing systems, positive increments usually are down and to the right.

2.8 Two-Dimensional Viewing

In Program 2.1, we used the default viewing conditions. In OpenGL, programs in two dimensions are a special case of three-dimensional programs. Two-dimensional objects have spatial coordinates of the form (x, y), but from an OpenGL perspective, these are three-dimensional (x, y, z) values, with z set to 0.[1] Consequently, two-dimensional viewing issues, such as which objects appear on the screen and at what size, are special cases of the same issues in three-dimensional viewing. However, because we are interested in getting started through two-dimensional programs, we can develop simple two-dimensional viewing independently.

The fundamental model we use in viewing is called the **synthetic-camera model**. It makes an analogy between a viewer—observer, photographer—forming a picture of a set of objects and what we do in the computer to produce an image. In two dimensions, we can define our objects by specifying or calculating a set of vertices, using some combination of `glVertex*()`, `glBegin()`, and `glEnd()` in our program. We can think of our code as describing objects on an infinite sheet of paper. The viewing step is specifying what part of that virtual sheet of paper is seen by our synthetic camera and thus appears on the screen. If we assume that the camera is aligned with the x and y axes, we need only specify a rectangular region through maximum and minimum values of x and y. We make this specification through the function `gluOrtho2D()`.

```
void gluOrtho2D(GLdouble left, GLdouble right, GLdouble
    bottom, GLdouble top)
```

Specifies a two-dimensional rectangular clipping region whose lower-left corner is at (`left, bottom`) and whose upper-right corner is at (`right, top`)

The prefix `glu` indicates that the function is the GLU library—because it is a special case of the three-dimensional function `glOrtho()`. The rectangle defined by `gluOrtho()` is called the **clipping window**. Objects that lie within this window are visible, whereas objects outside are not and are said to be clipped out.

1. In reality, OpenGL uses four dimensions. Three-dimensional space is a special case of four-dimensional space, but we do not have to worry about that yet.

2.9 Coordinate Systems and Transformations

So far, we have seen two coordinate systems in our functions. The first, called **object coordinates**, or **world coordinates**, is the application coordinate system that users use to write their programs. Each application program can decide what units it prefers and then specify values in these units in OpenGL functions such as `glVertex*()`. Thus, we can use microns for problems in very large-scale integration (VLSI) design or light years for astronomical problems. The second coordinate system, called **window coordinates**, or **screen coordinates**, uses units measured in pixels. The allowable range of window coordinates is determined by properties of the physical display and what part of that display is selected by the application program.

OpenGL automatically makes a coordinate transformation from object to window coordinates as part of the rendering process. The only information required is the size of the display window on the screen and how much of the object world the user wishes to display. The former is determined by `glutInitWindowSize()`—and possibly modified by later interactions—whereas the latter is set by `gluOrtho2D()`.

The required coordinate system transformations in OpenGL are determined by two matrices—the **model-view matrix** and the **projection matrix**—that are part of OpenGL's state. (We study these matrices in detail in Chapter 5.) However, we need to use a simple projection matrix in even the most basic programs. The function `gluOrtho2D()` is used to specify a projection matrix for two-dimensional applications. The typical sequence to set either of the matrices requires that we perform three steps.

1. Identify which matrix we wish to alter.
2. Set the matrix to an identity matrix.
3. Alter the identity matrix.

The second step is not required if we want to alter an existing matrix incrementally. Thus, if we want to set up a two-dimensional clipping window whose lower-left corner is at (–1.0, –1.0) and whose upper-right corner is at (1.0, 1.0), which are the default values we used in OpenGL, we execute the functions

```
glMatrixMode(GL_PROJECTION):
glLoadIdentity();
gluOrtho2D(-1.0, 1.0, -1.0, 1.0);
```

```
void glMatrixMode(GLenum mode)
```

Specifies which matrix will be affected by subsequent transformation functions. The `mode` is usually `GL_MODELVIEW` or `GL_PROJECTION`.

```
void glLoadIdentity()
```

Initializes the current matrix to an identity matrix.

Since these matrices are part of the OpenGL state, OpenGL will use their current values whenever a primitive is defined. These matrices can be changed virtually anywhere in an application program. For our simple example, without user interaction, we can set the matrices once as part of the initialization phase of the program. In Chapter 3, we will change the transformations in response to user events, such as the resizing of the screen window.

2.10 Second Version of a Simple Program

We can now incorporate all these changes into our program. The resulting program, Program 2.2, will behave the same as Program 2.1, but the structure of Program 2.2 is more general and characterizes more complex two-dimensional applications.

Program 2.2 illustrates the organization we use for almost all our programs in this book. Our programs consist of four major parts:

1. A `main()` function that initializes GLUT, puts a window on the screen, identifies the callback functions, and enters the main loop
2. An `init()` function that sets state variables to their initial values
3. A display callback, `display()`, that contains the code describing our objects
4. Other callbacks that deal with input and window events

Although other structures are possible, this organization has some advantages. The `main()` function is almost the same from program to program. Differences are usually related to which callbacks and menus are used in a particular application. Using `init()` allows us to place a lot of detailed state information and desired parameters in one place, separate from the geometry—which is in the display callback—and from the dynamics of animated and interactive programs, which usually are in the callbacks.

```
/* simple.c second version */
/* This program draws a white rectangle on a black background.*/


#include <GL/glut.h>        /* glut.h includes gl.h and glu.h*/


void display()

{
/* clear window */

    glClear(GL_COLOR_BUFFER_BIT);


/* draw unit square polygon */

    glBegin(GL_POLYGON);
        glVertex2f(-0.5, -0.5);
        glVertex2f(-0.5, 0.5);
        glVertex2f(0.5, 0.5)
        glVertex2f(0.5, -0.5);
    glEnd();

/* flush GL buffers */

    glFlush();

}


void init()
{

/* set clear color to black */

    glClearColor(0.0, 0.0, 0.0, 0.0);

/* set fill color to white */

    glColor3f(1.0, 1.0, 1.0);

/* set up standard orthogonal view with clipping */
/* box as cube of side 2 centered at origin */
/* This is default view and these statements could be removed */
```

Program 2.2 Second version of `simple.c` *Continued on next page.*

```
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-1.0, 1.0, -1.0, 1.0);

int main(int argc, char** argv)
{

/* Initialize mode and open a window in upper left corner of
/* screen */
/* Window title is name of program (arg[0]) */

    glutInit(&argc,argv)
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("simple");
    glutDisplayFunc(display);
    init();
    glutMainLoop();

}
```

Program 2.2 Second version of `simple.c` (*continued*)

2.11 Primitives and Attributes

Primitives are the atomic entities that we work with in our graphics system. In the one primitive we have seen so far, a polygon, is defined by a set of vertices. The polygon is a geometric primitive. That is, the polygon exists in a space, usually a two- or three-dimensional user-defined space, and can be imaged by our viewing process. Later, we shall see that OpenGL allows us to apply transformations to geometric primitives so that we can move them, resize them, and reorient them. OpenGL also has nongeometric primitives, such as pixels, that are dealt with in quite a different manner. Those primitives are introduced in Chapter 7.

The three basic types of geometric primitives in OpenGL are points, line segments, and polygons. More sophisticated objects can be built out of these primitives, or we can use OpenGL curves and surfaces (Chapter 9). The basic primitives are all determined by vertices. Thus, they are specified as was the polygon in `simple.c`, but the type parameter in `glBegin()` varies.

Each primitive has **attributes**, properties that determine how it is displayed by OpenGL. For example, the polygon in our simple program was drawn in white. A line segment might be drawn in green and be thick or thin. In Chapter 6, we

introduce more sophisticated attributes, called material properties, that determine how light interacts with the primitive.

It is important to remember that although we conceptualize attributes as being associated with objects—a red line, a blue point—in fact, OpenGL regards attributes as part of its state. Thus, when we produced a white polygon in `simple.c`, the whiteness of the polygon was determined by the present color, which was part of the state, being white. In Chapter 3, we shall learn how the programmer can better bind attributes to objects.

2.11.1 Points

Points are the simplest primitive. The type in `glBegin()` is `GL_POINTS` (see Figure 2.2, a). Each vertex determines a point; points that are within the clipping window are displayed using the present point size attribute, which is set by `glPointSize()` and the present color.

```
void glPointSize(GLfloat size)
```

Sets the point size state variable. Size is measured in pixels on the screen, and the default is 1.0.

We can thus use the same four vertices we used for our polygons but display those vertices as points in four different colors with the code

```
glPointSize(2.0);
glBegin(GL_POINTS);
    glColor3f(1.0, 1.0, 1.0);
    glVertex2f(-0.5, -0.5);
    glColor3f(1.0, 0.0, 0.0);
    glVertex2f(-0.5, 0.5);
    glColor3f(0.0, 0.0, 1.0);
    glVertex2f(0.5, 0.5);
    glColor3f(0.0, 1.0, 0.0);
    glVertex2f(0.5, -0.5);
glEnd();
```

Note that `glPointSize()` is one of the functions that cannot go between a `glBegin()` and a `glEnd()`.

2.11.2 Lines

There are three choices for `type` (see Figure 2.2, b, c, and d) for line segments. We can use `GL_LINES`, `GL_LINE_STRIP`, and `GL_LINE_LOOP`, to define one or more line segments between a `glBegin()` and a `glEnd()`.

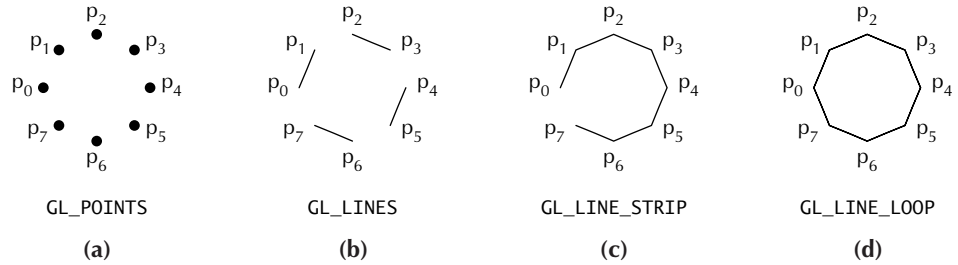

Figure 2.2 Point and line types

`GL_LINES`: Each successive pair of vertices between `glBegin()` and `glEnd()` defines a line segment. Thus, the following code defines two line segments: the first from (`-0,5, -0.5`) to (`-0.5, 0.5`) and the second from (`0.5, 0.5`) to (`0,5, -0.5`):

```
glBegin(GL_LINES);
    glVertex2f(-0.5, -0.5);
    glVertex2f(-0.5, 0.5);
    glVertex2f(0.5, 0.5);
    glVertex2f(0.5, -0.5);
glEnd();
```

`GL_LINE_STRIP`: The vertices define a sequence of line segments with the end point of one segment starting the next line segment. Thus, the following code defines three line segments: the first from (`-0.5, -0.5`) to (`-0.5, 0.5`), the second from (`-0.5, 0.5`) to (`0.5, 0.5`), and the third from (`0.5, 0,5`) to (`0.5, -0.5`):

```
glBegin(GL_LINE_STRIP);
    glVertex2f(-0.5, -0.5);
    glVertex2f(-0.5, 0.5);
    glVertex2f(0.5, 0.5);
    glVertex2f(0.5, -0.5);
glEnd();
```

`GL_LINE_LOOP`: The vertices connect the line segments as in `GL_LINE_STRIP`, but in addition, the last vertex is connected to the first. Thus, the following code defines a square:

```
glBegin(GL_LINE_LOOP);
    glVertex2f(-0.5, -0.5);
    glVertex2f(-0.5, 0.5);
    glVertex2f(0.5, 0.5);
    glVertex2f(0.5, -0.5);
glEnd();
```

The attributes for line segments are the color, the line thickness, and a pattern, called the **stipple pattern**, that allows us to create dashed and dotted lines.

```
void glLineWidth(GLfloat width)
```

Sets the `width` in pixels for the display of lines. The default is 1.0.

```
void glLineStipple(GLint factor, Glushort pattern)
```

Defines a 16-bit pattern for drawing lines. If a bit in `pattern` is 1, a pixel on the line is drawn. If the bit is 0, the pixel is not drawn. Successive groups of 1s and 0s in `pattern` are repeated `factor` times for values of `factor` between 1 and 256. The stipple pattern is repeated as necessary to draw the line. The bits are used starting with the lowest-order bits.

For example, the following commands set the drawing color to yellow, define lines as two pixels wide, and define a dashed stipple pattern in which groups of six pixels are not colored and the following six pixels are rendered in yellow:

```
glColor3f(1.0, 1.0, 0.0);
glLineWidth(2.0);
glLineStipple(3, 0xcccc);
```

2.11.3 Enabling OpenGL Features

Stippling is one of many OpenGL features that have to be enabled specifically. The renderer has many capabilities, such as lighting, hidden-surface removal, and texture mapping, although generally each feature will slow down the rendering processing. A user program can turn on—enable—or turn off—disable—each of these features individually with the application program. Some features, such as lighting, may be required in one part of a program but not in others.

```
void glEnable(GLenum feature)
void glDisable(GLenum feature)
```

Turns the OpenGL option `feature` on or off.

Line stippling is enabled by

```
glEnable(GL_LINE_STIPPLE);
```

Don't forget to enable features you want to use. Setting the parameters is not sufficient if the feature has not been enabled.

2.11.4 Filled Primitives

The polygon primitive with which we started is one example of a filled primitive, that is, a primitive with an interior that can be filled with a color or a pattern. Figure 2.3 shows the six filled primitives with the type parameters.

`GL_POLYGON`: Defines a polygon by a sequence of `glVertex*()` calls between and `glBegin()` and `glEnd()`.

`GL_QUADS`: Successive groups of four vertices define quadrilaterals.

`GL_TRIANGLES`: Treats each successive group of three vertices between a `glBegin()` and a `glEnd()` as a triangular polygon. Extra vertices are ignored.

`GL_TRIANGLE_STRIP`: The first three vertices after a `glBegin()` define the first triangle. Each subsequent vertex is used with the previous two to define the next triangle. Thus, after the first polygon is defined, the others require only a single `glVertex*()` call.

`GL_QUAD_STRIP`: The first four vertices define a quadrilateral. Each subsequent pair of vertices is used with the previous pair of vertices to define the next quadrilateral.

`GL_TRIANGLE_FAN`: The first three vertices define the first triangle. Each subsequent vertex is used with the first vertex and the previous vertex to define the next triangle.

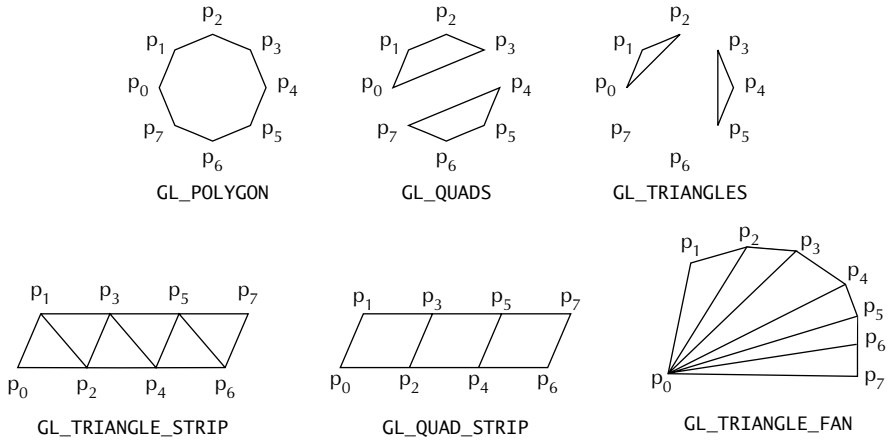

Figure 2.3 Filled types

These multiple polygon types have two advantages. First, a particular OpenGL implementation may have special software and hardware to render triangles or quadrilaterals more quickly than general polygons. Second, many CAD applications generate triangles or quadrilaterals with shared edges. Strip primitives allow us to define these primitives with far fewer OpenGL functions calls than if we had to treat each as a separate polygon. In Chapter 4, we introduce vertex arrays as another way that we can reduce the number of function calls required to define complex objects that share vertices.

2.11.5 Rectangles

OpenGL provides a function, `glRect*()`, for drawing two-dimensional filled rectangles aligned with the axes.

```
void glRect{sifd}(TYPE x1, y1, x2, y2)
void glRect{sifd}v(TYPE *v1, TYPE *v2)
```

Specifies a two-dimensional rectangle, using the standard data types by the `x` and `y` values of the corners or by pointers to arrays with these values.

2.11.6 Polygon Stipple

All the filled types are treated as polygons by the rendering process and thus have the same attributes. The simplest way to display a polygon is to fill it with a solid color. As we saw in `simple.c`, we can obtain a solidly colored polygon by using `glColor*()`. We can also fill the polygon with a stipple pattern by enabling polygon stipple by

```
glEnable(GL_POLYGON_STIPPLE);
```

We then set the pattern by `glPolygonStipple()`.

```
void glPolygonStipple(const Glubyte *mask)
```

Sets the stipple pattern for polygons. The `mask` is a 32×32 pattern of bits.

The pattern is used as in line stipple but is two dimensional and is aligned with the window. Thus, if we rotate the polygon by changing its vertices and redrawing it, the stipple pattern will not be rotated.

2.12 Polygon Types

The unfilled primitives, such as lines and line loops, pose no difficulties whether the vertices are defined in two or in three dimensions. Such is not the case for filled primitives, however. Consider a polygon whose edges cross as in Figure 2.4. It is somewhat arbitrary which points we consider to be inside the polygon and which outside. Unless polygons are **simple polygons**—polygons whose edges do not cross—two different OpenGL implementations may render them differently. OpenGL does not check whether a polygon is simple; that is left to the application program.

Two-dimensional polygons have an additional feature: All the vertices lie in the same plane and thus for nonsimple polygons, an interior is well defined. In three dimensions, a set of more than three vertices need not lie in the same plane. Once more, different OpenGL implementations might render such polygons differently. If this situation is a potential problem, it must be dealt with within the application program.

A third issue is that even when all vertices lie in the same plane, rendering a complex polygon with many vertices can present problems for the implementation. **Convex objects** are ones for which if we connect any two points in the object, the entire line segment connecting these points lies inside the object (Figure 2.5). Convex polygons are much easier to render. Because triangles are always convex and every triangle is planar, graphics systems usually work best with triangles.

In OpenGL, a polygon can be displayed in three different ways: filled, by its edges, or just as a set of points (the vertices). In addition, because our two-dimensional polygons are really three-dimensional polygons that are restricted to the plane $z = 0$, they have two faces: a front face and a back face. OpenGL can render either or both faces.

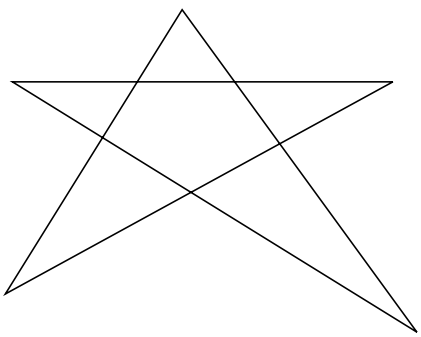

Figure 2.4 Nonsimple polygon

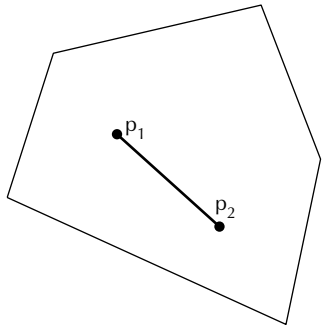

Figure 2.5 Convex polygon

A **front face** is one in which the order of the vertices is counterclockwise when we view the polygon. A **back face** is one in which the vertices are specified in a clockwise order. These definitions make sense for convex polygons. Defining front and back for nonconvex polygons may have difficulties, but OpenGL does not promise anything for such polygons, anyway.

The function `glPolygonMode()` lets us tell OpenGL how to render the faces.

```
void glPolygonMode(GLenum face, GLenum mode)
```

Specifies how the faces (`GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`) are to be rendered (`GL_POINT`, `GL_LINE`, or `GL_FILL`). The default is to fill both faces.

We can also not render either or both faces by culling the front- or back-facing polygon using `glCullFace()`.

```
void glCullFace(GLenum mode)
```

Causes the faces specified by `mode` (`GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`) to be ignored during rendering.

Culling must be enabled by

```
glEnable(GL_CULL_FACE);
```

OpenGL allows us to change the definition of front and back facing through the function `glFrontFace()`.

```
void glFrontFace(GLenum mode)
```

Allows the specification of either the counterclockwise (`GL_CCW`) or clockwise (`GL_CW`) direction for defining a front face.

Suppose that we want to display a polygon both by filling it and by displaying its edges. For example, we might want to display a polygon with yellow edges and filled in red. In OpenGL, the edges of a polygon are part of the inside of the polygon, so we cannot show both the edges and the inside with a single rendering. Instead, we can render the polygon twice: first with the polygon mode set to `GL_LINE` and the color set to yellow and then with mode set to `GL_FILL` and the color set to red, as in the code

```
glPolygonMode(GL_FILL);
glColor3fv(yellow);
square();
glPolygonMode(GL_LINE);
glColor3fv(red);
square();
```

In this code, the colors are in arrays, and the polygon is defined in the function `square()`. But we have a potential problem here. The two renderings of the polygon are on top of each other and even though the edge is drawn after the fill, small numerical errors in the renderer can cause parts of the yellow edge to be hidden by the red fill.

To solve this problem, we can ask OpenGL to move the lines slightly forward by the function `glPolygonOffset()`. The offset is a linear combination of the two parameters weighted by two internal constants, so it is not simple to set in terms of units of the application. Nevertheless, if you cannot get the desired effects without using the offset, some trial and error with the parameters may give a better image.

```
void glPolygonOffset(GLfloat factor, GL float units)
```

Sets the offset for polygons. The offset can be positive or negative and can be enabled for any of the polygon modes.

Polygon offset can be enabled separately for any of the three polygon modes. For example, the following with a positive offset should work for our example:

```
glEnable(GL_POLYGON_OFFSET_LINES);
```

2.13 Color Interpolation

The color used to render a polygon is the present value of the color state variable. When we change colors between calls to `glVertex*()`, we change the state but conceptually are associating the new color with the next vertex. In OpenGL, we often refer to **vertex colors** when we use them in this manner.

Suppose that we have a line segment defined by

```
glBegin(GL_LINES);
    glColor3f(1.0, 0.0, 0.0);
    glVertex2f(1.0, 0.0);
    glColor3f(0.0, 0.0, 1.0);
    glVertex2f(0.0, 1.0);
glEnd();
```

The vertices are defined as red and blue, but in what color will OpenGL render the points between the vertices? The default is to use **smooth shading**, whereby OpenGL will interpolate the colors at the vertices to obtain the color of intermediate points. Thus, as we go along the line, we see the color starting as red and then passing through various shades of magenta before becoming blue.

For polygons, the same is true except that the interpolation formula must interpolate the vertex colors across the interior of the polygons. Usually, OpenGL renders polygons as a set of triangles, using a simple two-dimensional interpolation formula called **bilinear interpolation**. Interpolating vertex properties will arise in other contexts later, such as in texture mapping and using material properties.

OpenGL also allows us to use the color at the first vertex to determine the properties of the entire primitive. Thus, we could have a solid blue line, or we could obtain a green polygon, even if there were color changes between vertex definitions. This style is called **flat shading** and is set by setting the shading model by the function `glShadeModel()`.

```
void glShadeModel(GLenum mode)
```

Sets the shading model to smooth (`GL_SMOOTH`) or flat (`GL_FLAT`). Smooth shading is the default.

Suppose that you have an application that generates nonconvex, nonplanar polygons. What can you do? You could hope for the best, as OpenGL generally will produce something. Or, you could break up, or **tessellate**, your polygons into triangles within your application. But a problem arises if we wish to display only the

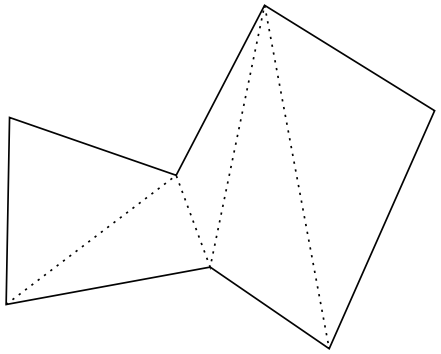

Figure 2.6 Tessellating a polygon

edges. Consider the polygon in Figure 2.6. When we break it up for display into five triangles, we won't have a problem if we fill the five polygons. However, if the polygon mode is set to display edges, we want only the edges corresponding to the original polygon to be displayed, not the new edge created by the tessellation.

```
void glEdgeFlag(GLboolean flag)
void glEdgeFlagv(GLboolean *flag)
```

Sets the edge flag (`GL_TRUE`, `GL_FALSE`) that determines if subsequent vertices are the start of edges that should be displayed if the polygon mode is `GL_LINE`.

We can decide which edges to display by using the function `glEdgeFlag*()`. If the flag is set to `GL_TRUE`, each vertex is considered the beginning of a line segment to be displayed. If the flag is set to `GL_FALSE`, the vertices do not start edges that are to be displayed.

We also have a third option. The GLU library provides a tessellator. The tessellator decides how to break up a general polygon and takes cares of such issues as creating faces that have a consistent facing and setting the edge flags. Use of the tessellator requires a large number of additional functions, and we will not discuss it further.

2.14 Text

Text is not one of the OpenGL primitives. This fact may seem a bit strange, given the importance of text in such applications as producing graphs. However, it is important to remember that OpenGL provides a minimal set of primitives that

provide building blocks for the application programmer. We could attempt to build a set of characters from the primitives that we have seen, but that might not be a very appealing task.

In general, text generation presents a few problems that must be confronted. Suppose that we want to create a **font**, a set of characters in a given size and style, such a 10-point Times Roman bold font. The two principal forms for generating such characters are **bitmap characters** and **stroke characters**. Bitmap characters are stored as rectangular patterns of bits in which if a bit is 1, it is displayed; otherwise, it is not displayed. In Chapter 7, we shall study how OpenGL allows the user to define and to output bitmaps. Such characters are very fast to generate, since each can be requested by a single OpenGL call. However, bitmap characters cannot be modified nicely by such operations as scaling. In addition, the patterns will appear differently on windows of different sizes. Stroke characters are generated by using the standard OpenGL primitives, such as lines, polygons, and curves and can be modified by the transformations that we will discuss in Chapter 5. However, stroke characters require more storage and are slower to generate. If you are willing to do the work, you can generate either type of characters. OpenGL supports both.

A simpler approach might be to use the fonts that are provided by most windowing systems. Programs that do so might not be portable to a different environment, but switching to another font in another system does not often present major problems. We can obtain such a font through system specific functions.

GLUT offers a third possibility by providing a few of its own bitmap and stroke fonts. We can obtain a bitmap character through the function `glutBitmapCharacter()`, which uses the patterns from certain fonts in the X Window System.

```
void glutBitMapCharacter(void *font, int char)
```

Renders the character `char`, given by an ASCII code, in the font given by `font`.

Thus, we can get a bitmapped Times Roman 10-point character 'a' by

```
glutBitMapCharacter(GLUT_BITMAP_TIMES_ROMAN_10, 'a');
```

Similarly, we can get an 8×13 bit character for 'a' by

```
glutBitMapCharacter(GLUT_BITMAP_8_BY_13, 'a');
```

But where does the character appear on the screen? In OpenGL, bitmaps are handled differently from our geometric primitives. Bimaps appear in the size specified at a location determined the **raster position**, which is part of the OpenGL

state. This position determines where the lower-left corner of the next bitmap will appear on the display. We can set that position with the function `glRasterPos*()`.

```
void glRasterPos{234}{sifd}(TYPE x, TYPE y, TYPE z, TYPE w)
void glRasterPos{234}{sifd}v(TYPE *array)
```

Specifies the raster position. The position is mapped to screen coordinates, using the current model-view and projection matrices.

Thus, we can set the raster position in world coordinates. The current raster position is updated automatically so that the next character will not be rendered on top of the previous one. GLUT provides the helper function `glutBitMapWidth()`, which allows the application program to determine the width of character so that it can determine how to change the raster position if necessary.

```
int glutBitmapWidth(GLUTbitmapFont font, int char)
```

Returns the width in pixels of `char` in the GLUT font named by `font`.

The corresponding functions for stroke fonts are `glutStrokeCharacter()` and `glutStrokeWidth()`.

```
void glutStrokeCharacter(void *font, int char)
```

Renders the character `char`, given by an ASCII code, in the stroke font given by `font`.

```
int glutStrokeWidth(GLUTbitmapFont font, int char)
```

Returns the width in bits of `char` in the GLUT font named by `font`.

The fonts `GLUT_STROKE_MONO_ROMAN` and `GLUT_STROKE_ROMAN` are fixed and proportional stroke fonts. However, their sizes are not in bits. Their sizes are approximately 100×100 units in world coordinates. Because they are defined as other geometric primitives, they pass through the geometric pipeline. Consequently, the usual way to position stroke characters is through the OpenGL transformations, such as scaling and translation, as discussed in Chapter 5.

2.15 Inquiries and Errors

Thus far, we have assumed that everything works perfectly; our programs compile and run just as we imagined. Now let's consider reality. We make errors; programs may run but not give the results we expect or may run and display nothing. OpenGL provides some error-checking facilities. It also allows us to obtain the values of any part of the state.

The OpenGL state can be inquired through the six functions `glGetBooleanv()`, `glGetIntegerv()`, `glGetFloatv()`, `glGetDoublev()`, `glGetPointerv()`, and `glIsEnabled()`. The first five return a pointer to the proper type. The application program needs to know the parameter it is seeking. We can obtain both present values, which are part of the state, and system parameters. For example, the following returns the current RGBA values to the array, which should have been allocated to hold four floats:

```
glGetFloatv(GL_CURRENT_COLOR, color_array);
```

The following call returns the number of red bits used for color in the implementation.

```
glGetIntegerv(GL_RED_BITS, bits);
```

Later, we will see a few other `get` functions in OpenGL.

```
void glGetBooleanv(GLenum name, GLboolean *value)
void glGetIntegerv(GLenum name, GLint *value)
void glGetFloatv(GLenum name, GLfloat *value)
void glGetDoublev(GLenum name, GLdouble *value)
void glGetPointerv(GLenum name, GLvoid **value)
```

Returns values of parameters in the state or system parameters named by `name` to user variables.

The function `glIsEnabled()` allows a program to check whether a particular feature has been enabled by a `glEnable()`.

```
GLboolean glIsEnabled(GLenum feature)
```

Returns `GL_TRUE` or `GL_FALSE`, depending on whether `feature` is enabled.

Error checking can be done in two parts. We can check if an error has been made by `glGetError()`, which returns an error type or `GL_NO_ERROR` if no error

has been made. Errors are measured from initialization until this function is called. When the function is called, the error flag is reset to GL_NO_ERROR. A string for the particular error can be obtained by gluGetErrorString().

```
GLenum glGetError()
```

Returns the type of the last error since initialization or the last call to glGetError(). If no error has occurred, GL_NO_ERROR is returned.

```
GLubyte* gluErrorString(GLenum error)
```

Returns a string corresponding to the error returned by glGetError().

The error-reporting mechanism in GLUT is implementation dependent. Generally, if you request a facility that is unsupported, the program will terminate with an error message. GLUT provides a function, glutGet(), that lets the application obtain information about the state of GLUT.

```
int glutGet(GLenum state)
```

Returns the state of the specified GLUT state variable.

We can also obtain other information. We can determine depth of the color buffer (GLUT_WINDOW_BUFFER_SIZE) or whether the current display mode is supported (GLUT_DISPLAY_MODE_POSSIBLE).

2.16 Saving the State

The OpenGL state determines how primitives are rendered. Virtually all changes we make to attributes and to other items, such as the model-view and projection matrices, change the state. Programs that alter these variables can spend most of their time making state changes, many of which require recalculation of variables. For example, we often have to compute our camera parameters or the colors we wish to use. Rather than recalculate values we have used previously, OpenGL provides two types of stacks on we can store values for later use.

The **matrix stacks** store projection and model-view matrices, with a separate stack for each type. We push and pop matrices with glPushMatrix() and glPopMatrix(). The stack used is the one corresponding to the present matrix mode (GL_MODELVIEW or GL_PROJECTION).

```
void glPushMatrix()
void glPopMatrix()
```

Pushes and pops matrices to the stack for the present matrix mode.

Matrix stacks have two major uses. When we build hierarchical models in Chapter 5, we shall use stacks to traverse the tree data structures that we will use to describe these models. This application involves the model-view matrix. The second use involves the projection matrix. We often have to do a fair amount of calculation or use user input to determine the required projection matrix. Suppose that we then want to zoom in on the scene temporarily. We can do this by altering the present projection matrix in a fairly simple manner. The problem is that when we want to return to the unzoomed view, we do not want to—or may be unable to—recalculate the original projection matrix. Saving it on the stack before we zoom solves the problem. Thus, we often see such code as

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glutPostRedisplay();
glPushMatrix();
/* change the projection matrix */
glutPostRedisplay();
glutPopMatrix();
```

Note the pairing of the push and pop operations: one pop for each push. In hierarchical systems, not having the correct pairing can leave the stack in the wrong state.

Pushes and pops must be paired in a program to be able to return to the desired state.

OpenGL breaks its attributes into 20 groups corresponding to sets of related attributes. For example, all the polygon attributes are in the group `GL_POLYGON_BIT`. All the line attributes are in the group `GL_LINE_BIT`. We can push any groups of attributes or all attributes (`GL_ALL_ATTRIBUTE_BITS`) onto the attribute stack through the function `glPushAttrib()` and recover them through `glPopAttrib()`.

```
void glPushAttrib(GLbitfield mask)
void glPopAttrib()
```

Pushes and pops groups of attributes to the attribute stack. The identifiers for the groups are combined with logical `OR` to form `mask`.

2.17 The Viewport

So far we have used the entire window for our graphics. We can also restrict OpenGL to draw to any part of the window through the use of a **viewport**. A viewport is a rectangular area of the window on the screen. Its size is measured in pixels. We set the viewport by the function `glViewport()`.

```
void glViewport(GLint x, GLint y, GLsizei w, GLsizei h)
```

Sets the viewport of width `w` and height `h` pixels in the window with its lower-left corner at (`x`, `y`). The default viewport is the entire initial window.

One use of viewports is to divide the window so that different types of information can be rendered to different parts of the window. For example, in a CAD application we might use some viewports for menus and instructions and another for constructing the design. Each of these viewports can have its own viewing conditions specified by its own use of `gluOrtho2d()`. By using viewports that do not overlap, we can produce a complex image in a single window with minimum complexity in the code.

In Chapter 3, we use viewports to control the appearance of our images when the user changes the shape of the window.