Our approach to computer graphics is programming oriented. Consequently, we want you to get started programming graphics as soon as possible. To this end, we will introduce a minimal application programmer’s interface (API). This API will be sufficient to allow you to program many interesting two- and three-dimensional problems and to familiarize you with the basic graphics concepts.

We regard two-dimensional graphics as a special case of three-dimensional graphics. This perspective allows us to get started, even though we will touch on three-dimensional concepts lightly in this chapter. Our two-dimensional code will execute without modification on a three-dimensional system.

Our development will use a simple but informative problem: the Sierpinski gasket. It shows how we can generate an interesting and, to many people, unexpectedly sophisticated image using only a handful of graphics functions. We use OpenGL as our API, but our discussion of the underlying concepts is broad enough to encompass most modern systems. The functionality that we introduce in this chapter is sufficient to allow you to write basic two- and three-dimensional programs that do not require user interaction.

### 2.1 THE SIERPINSKI GASKET

We will use as a sample problem the drawing of the Sierpinski gasket—an interesting shape that has a long history and is of interest in areas such as fractal geometry. The Sierpinski gasket is an object that can be defined recursively and randomly; in the limit, however, it has properties that are not at all random. We start with a two-dimensional version, but as we will see in Section 2.11, the three-dimensional version is almost identical.

Suppose that we start with three points in space. As long as the points are not collinear, they are the vertices of a unique triangle and also define a unique plane. We assume that this plane is the plane $z = 0$ and that these points, as
specified in some convenient coordinate system, are \((x_1, y_1, 0)\), \((x_2, y_2, 0)\), and \((x_3, y_3, 0)\). The construction proceeds as follows:

1. Pick an initial point \((x, y, 0)\) at random inside the triangle.
2. Select one of the three vertices at random.
3. Find the location halfway between the initial point and the randomly selected vertex.
4. Display this new point by putting some sort of marker, such as a small circle, at the corresponding location on the display.
5. Replace the point at \((x, y, 0)\) with this new point.
6. Return to step 2.

Thus, each time that we generate a new point, we display it on the output device. This process is illustrated in Figure 2.1, where \(p_0\) is the initial location, and \(p_1\) and \(p_2\) are the first two locations generated by our algorithm.

Before we develop the program, you might try to determine what the resulting image will be. Try to construct it on paper; you might be surprised by your results.

A possible form for our graphics program might be this:

```c
main() {
    initialize_the_system();

    for(some_number_of_points) {
        pt = generate_a_point();
        display_the_point(pt);
    }
    cleanup();
}
```

This form can be converted into a real program fairly easily. However, even at this level of abstraction, we can see two other alternatives. Consider the pseudocode

```c
main() {
    initialize_the_system();

    for(some_number_of_points) {
        pt = generate_a_point();
        display_the_point(pt);
    }
    cleanup();
}
```

1. In Chapter 3, we expand the concept of a coordinate system to the more general formulation of a frame.
In this algorithm, we compute all the points first and put them into an array or some other data structure. We then display all the points through a single function call. This approach avoids the overhead of sending small amounts of data to the graphics processor for each point we generate at the cost having to store all the data. The strategy used in the first algorithm is known as immediate mode graphics and, until recently, was the standard method for creating displays, especially where interactive performance was needed. One consequence of immediate mode is that there is no memory of the geometric data. With our first example, if we want to display the points again, we would have to go through the entire creation and display process a second time.

In our second algorithm, because the data are stored in a data structure, we can redisplay the data, perhaps with some changes such as altering the color or changing the size of a displayed point, by resending the array without regenerating the points. The method of operation is known as retained mode graphics and goes back to some of the earliest special purpose graphics displays. The architecture of modern graphics systems that employ a GPU leads to a third version of our program.

Our second approach has one major flaw. Suppose that, as we might in an animation, we wish to redisplay the same objects. The geometry of the objects is unchanged but the objects may be moving. Displaying all the points involves sending the data from the CPU to the GPU each time we wish to display the objects in a new position. For large amounts of data, this data transfer is the major bottleneck in the display process. Consider the following alternative scheme:

```c
main( )
{
    initialize_the_system();

    for(some_number_of_points)
    {
        pt = generate_a_point();
        store_the_point(pt);
    }
    send_all_points_to_GPU();
    display_data_on_GPU;
    cleanup();
}
```
As before, we place data in an array but now we have broken the display process into two parts: storing the data on the GPU and displaying the data that has been stored. If we only have to display our data once there is no advantage over our previous method but if we want to animate the display, our data are already on the GPU and redisplay does not require any additional data transfer, only a simple function call that alters the location of some spatial data describing the objects that have moved.

Although our final OpenGL program will have a slightly different organization, it will follow this third strategy. We develop the full program in stages. First, we concentrate on the core: generating and displaying points. We must answer two questions:

- How do we represent points in space?
- Should we use a two-dimensional, three-dimensional, or other representation?

Once we answer these questions, we will be able to place our geometry on the GPU in a form that can be rendered. Then, we will be able to address how we view our objects using the power of programmable shaders.

### 2.2 PROGRAMMING TWO-DIMENSIONAL APPLICATIONS

For two-dimensional applications, such as the Sierpinski gasket, although we could use a pen-plotter API, such an approach would limit us. Instead, we choose to start with a three-dimensional world; we regard two-dimensional systems, such as the one on which we will produce our image, as special cases. Mathematically, we view the two-dimensional plane, or a simple two-dimensional curved surface, as a subspace of a three-dimensional space. Hence, statements—both practical and abstract—about the larger three-dimensional world hold for the simpler two-dimensional world.

We can represent a point in the plane $z = 0$ as $p = (x, y, 0)$ in the three-dimensional world, or as $p = (x, y)$ in the two-dimensional plane. OpenGL, like most three-dimensional graphics systems, allows us to use either representation, with the underlying internal representation being the same, regardless of which form the user chooses. We can implement representations of points in a number of ways, but the simplest is to think of a three-dimensional point as being represented by a triplet $p = (x, y, z)$ or a column matrix

$$p = \begin{bmatrix} x \\ y \\ z \end{bmatrix},$$

whose components give the location of the point. For the moment, we can leave aside the question of the coordinate system in which $p$ is represented.
We use the terms *vertex* and *point* in a somewhat different manner in OpenGL. A *vertex* is a position in space; we use two-, three-, and four-dimensional spaces in computer graphics. We use vertices to define the atomic geometric primitives that are recognized by our graphics system. The simplest geometric primitive is a point in space, which is usually specified by a single vertex. Two vertices can specify a line segment, a second primitive object; three vertices can specify either a triangle or a circle; four vertices can specify a quadrilateral, and so on. Two vertices can also specify either a circle or a rectangle. Likewise, three vertices can also specify three points or two connected line segments and four vertices can specify a variety of objects including two triangles.

The heart of our Sierpinski gasket program is generating the points. In order to go from our third algorithm to a working OpenGL program, we need to introduce a little more detail on OpenGL. We want to start with as simple a program as possible. One simplification is to delay a discussion of coordinate systems and transformations among them by putting all the data we want to display inside a square centered at the origin whose diagonal goes from (-1, -1, -1) and (1, 1, 1). This system known as *clip coordinates* is the one that our vertex shader uses to send information to the rasterizer. Objects outside this cube will be eliminated or *clipped* and cannot appear on the display. Later, we will learn to specify geometry in our application program in coordinates better suited for our application—*object coordinates*—and use transformations to convert the data to a representation in clip coordinates. The code

```c
void init()
{
    #define N 5000
    GLfloat points [N][2];

    /* A triangle in the plane z= 0 */
    GLfloat vertices[3][2]={{-1.0,-1.0}, {0.0,1.0}, {1.0,-1.0}};

    int j, k;
    int rand(); /* standard random number generator */

    /* An arbitrary initial point inside the triangle */
    points[0][0] = 0.25;
    points[0][1] = 0.50;

    /* compute and store N-1 new points */
    for( k=1; k<N; k++)
    {
        j=rand()%3; /* pick a vertex at random */

        /* Compute point halfway between selected vertex and old point */
```
Chapter 2 Graphics Programming

```
points[k][0] = (points[k-1][0]+vertices[j][0])/2.0;
points[k][1] = (points[k-1][1]+vertices[j][1])/2.0;
```

generates 5000 points starting with the vertices of a triangle that lie in the plane \( z = 0 \). Because every point we generate must lie inside the triangle determined by these vertices, we know that none of the generated points will be clipped out.

In OpenGL, we often use basic OpenGL types, such as `GLfloat` and `GLint`, rather than the corresponding C types `float` and `int`. These types are defined in the OpenGL header files and usually in the obvious way—for example,

```c
#define GLfloat float
```

However, use of the OpenGL types allows additional flexibility for implementations where, for example, we might want to change floats to doubles without altering existing application programs.

The function `rand` is a standard random-number generator that produces a new random integer each time it is called. We use the modulus operator to reduce these random integers to the three integers 0, 1, and 2. For a small number of iterations, the particular characteristics of the random-number generator are not crucial, and any other random-number generator should work at least as well as `rand`.

We intend to generate the points only once and then place them on the GPU. Hence, we make their creation part of an initialization function `init`.

We specified our points in two dimensions. We could have also specified them in three dimensions by adding a \( z \)-coordinate which is always zero. The changes to the code would be minimal. We would have the lines

```c
GLfloat points[N][3];
GLfloat vertices[3][3]={{-1.0,-1.0, 0.0}, {0.0,1.0, 0.0}, {1.0,-1.0, 0.0}};
```

and code that sets the third component of `points`

```c
points[0][2] = 0.0;
```

initially and

```c
points[k][2] = (points[k-1][2]+vertices[j][2])/2.0;
```

in the loop. Although, we still do not have a complete program; Figure 2.2 shows the output that we expect to see.
2.2 Programming Two-Dimensional Applications

Note that because any three noncollinear points define a plane, had we started with three points \((x_1, y_1, z_1)\), \((x_2, y_2, z_2)\), and \((x_3, y_3, z_3)\) along with an initial point in the same plane, then the gasket would be generated in the plane defined by the original three vertices.

We have now written the core of the program. Although we have some data, we have not placed these data on the GPU nor have we asked the GPU to display anything. We have not even introduced a single OpenGL function. Before we can display anything we still have to address issues such as the following:

1. In what colors are we drawing?
2. Where on the display does our image appear?
3. How large will the image be?
4. How do we create an area of the display—a window—for our image?
5. How much of our infinite drawing surface will appear on the display?
6. How long will the image remain on the display?

The answers to all these questions are important, although initially they may appear to be peripheral to our major concerns. As we will see, the basic code that we develop to answer these questions and to control the placement and appearance of our renderings will not change substantially across programs. Hence, the effort that we expend now will be repaid later.
Chapter 2 Graphics Programming

2.3 THE OPENGL API

We have the heart of a simple graphics program; now, we want to gain control over how our objects appear on the display. We also want to control the flow of the program, and we have to interact with the window system. Before completing our program, we describe the OpenGL API in more detail. Because vertices are represented in the same manner internally, whether they are specified as two- or three-dimensional entities, everything that we do here will be equally valid in three dimensions. Of course, we can do much more in three dimensions, but we are only getting started. In this chapter, we concentrate on how to specify primitives to be displayed.

OpenGL’s structure is similar to that of most modern APIs, including Java3D and DirectX. Hence, any effort that you put into learning OpenGL will carry over to other software systems. Although OpenGL is easy to learn, compared with other APIs, it is nevertheless powerful. It supports the simple two- and three-dimensional programs that we will develop in Chapters 2 through 6; it also supports the advanced rendering techniques that we study in Chapters 8 through 13.

Our prime goal is to study computer graphics; we are using an API to help us attain that goal. Consequently, we do not present all OpenGL functions, and we omit many details. However, our sample programs will be complete. More detailed information on OpenGL and on other APIs is given in the Suggested Readings section at the end of the chapter.

2.3.1 Graphics Functions

Our basic model of a graphics package is a black box, a term that engineers use to denote a system whose properties are described only by its inputs and outputs; we may know nothing about its internal workings. We can think of the graphics system as a box whose inputs are function calls from an application program; measurements from input devices, such as the mouse and keyboard; and possibly other input, such as messages from the operating system. The outputs are primarily the graphics sent to our output devices. For now, we can take the simplified view of inputs as function calls and outputs as primitives displayed on our monitor, as shown in Figure 2.3.

A graphics system performs multiple tasks to produce output and handle user input. An API for interfacing with this system can contain hundreds of
individual functions. It will be helpful to divide these functions into seven major groups:

1. Primitive functions
2. Attribute functions
3. Viewing functions
4. Transformation functions
5. Input functions
6. Control functions
7. Query functions

Although we will focus on OpenGL as the particular system that we use, all graphics APIs support similar functionality. What differs among APIs is where these functions are supported. OpenGL is designed around a pipeline architecture and modern versions are based on using programmable shaders. Consequently, OpenGL and other APIs such as DirectX that support a similar architecture will have much in common whereas OpenGL and an API for a ray tracer will have less overlap. Nevertheless, regardless of the underlying architecture and API, we still have to address all the seven tasks.

The primitive functions define the low-level objects or atomic entities that our system can display. Depending on the API, the primitives can include points, line segments, polygons, pixels, text, and various types of curves and surfaces. OpenGL supports a very limited set of primitives directly, only points line segments and triangles. Support for other primitives comes from the application approximating them with the supported primitives. For the most important objects such as regular polyhedra, quadrics, and Bezier curves and surface that are not directly supported by OpenGL, there are libraries that provide the necessary code. Support for expanded sets of primitives is usually done with great efficiency through programmable shaders.

If primitives are the what of an API—the primitive objects that can be displayed—then attributes are the how. That is, the attributes govern the way that a primitive appears on the display. Attribute functions allow us to perform operations ranging from choosing the color with which we display a line segment, to picking a pattern with which to fill the inside of a polygon, to selecting a typeface for the titles on a graph. In OpenGL, we can set colors by passing the information from the application to the shader or by having a shader compute a color, for example through a lighting model that uses data specifying light sources and properties of the surfaces in our model.

Our synthetic camera must be described if we are to create an image. As we saw in Chapter 1, we must describe the camera’s position and orientation in our world and must select the equivalent of a lens. This process will not only fix the view but also allow us to clip out objects that are too close or too far away. The viewing functions allow us to specify various views, although APIs differ in the degree of flexibility they provide in choosing a view. OpenGL does
not provide any viewing functions but relies on the use of transformations in the shaders to provide the desired view.

One of the characteristics of a good API is that it provides the user with a set of **transformation functions** that allows her to carry out transformations of objects, such as rotation, translation, and scaling. Our developments of viewing in Chapter 5 and of modeling in Chapter 10 will make heavy use of matrix transformations. In OpenGL, carry out transformations by forming transformations in applications and loading them in the shaders.

For interactive applications, an API must provide a set of **input functions** to allow us to deal with the diverse forms of input that characterize modern graphics systems. We need functions to deal with devices such as keyboards, mice, and data tablets. Later in this chapter, we will introduce functions for working with different input modes and with a variety of input devices.

In any real application, we also have to worry about handling the complexities of working in a multiprocessing, multiwindow environment—usually an environment where we are connected to a network and there are other users. The **control functions** enable us to communicate with the window system, to initialize our programs, and to deal with any errors that take place during the execution of our programs.

If we are to write device-independent programs, we should expect the implementation of the API to take care of differences between devices, such as how many colors are supported or the size of the display. However, there are applications where we need to know some properties of the particular implementation. For example, we would probably choose to do things differently if we knew in advance that we were working with a display that could support only two colors rather than millions of colors. More generally, within our applications we can often use other information within the API, including camera parameters or values in the frame buffer. A good API provides this information through a set of **query functions**.

### 2.3.2 The Graphics Pipeline and State Machines

If we put together some of these perspectives on graphics APIs, we can obtain another view, one closer to the way OpenGL, in particular, is actually organized and implemented. We can think of the entire graphics system as a **state machine**, a black box that contains a finite-state machine. This state machine has inputs that come from the application program. These inputs may change the state of the machine or can cause the machine to produce a visible output. From the perspective of the API, graphics functions are of two types: those that define primitives that flow through a pipeline inside the state machine and those that either change the state inside the machine or return state information. In OpenGL, there are very few functions that can cause any output. Most set the state, either by enabling various OpenGL features—hidden-surface removal, texture—or set parameters used for rendering.
Until recently, OpenGL defined many state variables and contained separate functions for setting the values of individual variables. The latest versions have eliminated most of these variables and functions. Instead the application program can define its own state variables and use them or send their values to the shaders.

One important consequence of state machine view is that most parameters are persistent; their values remain unchanged until we explicitly change them through functions that alter the state. For example, once we set a color, that color remains the current color until it is changed through a color-altering function. Another consequence of this view is that attributes that we may conceptualize as bound to objects—a red line or a blue circle—are in fact part of the state, and a line will be drawn in red only if the current color state calls for drawing in red. Although within our applications it is usually harmless, and often preferable, to think of attributes as bound to primitives, there can be annoying side effects if we neglect to make state changes when needed or lose track of the current state.

2.3.3 The OpenGL Interface

OpenGL functions are in a single library named GL (or OpenGL in Windows). Function names begin with the letters gl. Shaders are written in the OpenGL Shading Language (GL) which has a separate specification from OpenGL although the functions to interface the shaders with the application are part of the OpenGL library.

To interface with the window system and to get input from external devices into our programs, we need at least one more library. For each major window system there is a system-specific library that provides the "glue" between the window system and OpenGL. For the X Window System, this library is called GLX, for Windows, it is wgl, and for the Macintosh, it is agl. Rather than using a different library for each system, we use a readily available library called the OpenGL Utility Toolkit (GLUT), which provides the minimum functionality that should be expected in any modern windowing system. We introduce a few of its functions in this chapter and describe more of them in Chapter 3, where we consider input and interaction in detail.

Figure 2.4 shows the organization of the libraries for an X Window System environment. For this window system, GLUT will use GLX and the X libraries. The application program, however, can use only GLUT functions and thus can be recompiled with the GLUT library for other window systems.

OpenGL makes heavy use of macros to increase code readability and avoid the use of magic numbers. Thus, strings such as GL_FILL and GL_POINTS are defined in header (.h) files. In most implementations, one of the include lines

```
#include <GL/glut.h>
```
FIGURE 2.4  Library organization.

or

#include <GLUT/glut.h>

is sufficient to read in glut.h and gl.h.

Although OpenGL is not object oriented, it supports a variety of data types through multiple forms for many functions. For example, we will use the function various forms of the function glUniform to transfer data to shaders. If we transfer a floating point number such as a time, we would use glUniform1f. We could use glUniform3iv to transfer an integer position in three dimensions through a pointer to three dimensional array of ints. Later, we will use the form glUniformMatrix4fv to transfer a $4 \times 4$ matrix of floats. We will refer to such functions using the notation

```c
glSomeFunction*
```

where the * can be interpreted as either two or three characters of the form nt or ntv, where n signifies the number of dimensions (2, 3, 4, or matrix); t denotes the data type, such as integer (i), float (f), or double (d); and v, if present, indicates that the variables are specified through a pointer to an array, rather than through an argument list. We will use whatever form is best suited for our discussion, leaving the details of the various other forms to the OpenGL Programming Guide [Ope10]. Regardless of which form an application programmer chooses, the underlying representation is the same, just as the plane on which we are constructing the gasket can be looked at as either a two-dimensional space or the subspace of a three-dimensional space corresponding to the plane $z = 0$. In Chapter 3, we will see that the underlying representation is four-dimensional; however, we do not need to worry about that fact yet. In general, the application programmer chooses the form to use that is best suited for her application.
2.3.4 Coordinate Systems

At this point, if we look back at our Sierpinski gasket code, you may be puzzled about how to interpret the values of $x$, $y$, and $z$ in our specification of vertices. In what units are they? Are they in feet, meters, microns? Where is the origin? In each case, the simple answer is that it is up to you.

Originally, graphics systems required the user to specify all information, such as vertex locations, directly in units of the display device. If that were true for high-level application programs, we would have to talk about points in terms of screen locations in pixels or centimeters from a corner of the display. There are obvious problems with this method, not the least of which is the absurdity of using distances on the computer screen to describe phenomena where the natural unit might be light years (such as in displaying astronomical data) or microns (for integrated-circuit design). One of the major advances in graphics software systems occurred when the graphics systems allowed users to work in any coordinate system that they desired. The advent of device-independent graphics freed application programmers from worrying about the details of input and output devices. The user’s coordinate system became known as the world coordinate system, or the application or object coordinate system. Within the slight limitations of floating-point arithmetic on our computers, we can use any numbers that fit our application.

We will refer to the units that the application program uses to specify vertex positions as vertex coordinates. In most applications, vertex coordinates will be the same as object or world coordinates but depending on what we choose to do or not do in our shaders, vertex coordinates can be one of the other internal coordinate systems used in the pipeline. We will discuss these other coordinate systems in Chapters 3 and 4.

Units on the display were first called physical-device coordinates or just device coordinates. For raster devices, such as most CRT and flat panel displays, we use the term window coordinates or screen coordinates. Window coordinates are always expressed in some integer type, because the center of any pixel in the frame buffer must be located on a fixed grid or, equivalently, because pixels are inherently discrete and we specify their locations using integers.

At some point, the values in vertex coordinates must be mapped to window coordinates, as shown in Figure 2.5. The graphics system, rather than the user, is responsible for this task, and the mapping is performed automatically as part of the rendering process. As we will see in the next few sections, to define this mapping the user needs to specify only a few parameters—such as the area of the world that she would like to see and the size of the display. However, between the application and the frame buffer are the two shaders and rasterizer and as we shall see when we discuss viewing, there are three other intermediate coordinate systems of importance.
2.4 PRIMITIVES AND ATTRIBUTES

Within the graphics community, there has been an ongoing debate about which primitives should be supported in an API. The debate is an old one and has never been fully resolved. On the minimalist side, the contention is that an API should contain a small set of primitives that all hardware can be expected to support. In addition, the primitives should be orthogonal, each giving a capability unobtainable from the others. Minimal systems typically support lines, polygons, and some form of text (strings of characters), all of which can be generated efficiently in hardware. On the other end are systems that can also support a variety of primitives, such as circles, curves, surfaces, and solids. The argument here is that users need more complex primitives to build sophisticated applications easily. However, because few hardware systems can be expected to support the large set of primitives that is the union of all the desires of the user community, a program developed with such a system probably would not be portable, because few implementations could be expected to support the entire set of primitives.

As graphics hardware has improved and realtime performance has become measured in the tens of millions of polygons, the balance has tilted towards supporting a minimum set of primitives. One reason is that GPUs achieve their speed largely because they are optimized for points, lines, and triangles. We will develop code later that will approximate various curves and surfaces with primitives that are supported on GLUs.

We can separate primitives into two classes: geometric primitives and image, or raster, primitives. Geometric primitives are specified in the problem domain and include points, line segments, polygons, curves, and surfaces. These primitives pass through a geometric pipeline, as shown in Figure 2.6, where they are subject to a series of geometric operations that determine whether a primitive is visible, where on the display it appears if it is visible, and the rasterization of the primitive into pixels in the frame buffer. Because geometric primitives exist in a two- or three-dimensional space, they can be manipulated by operations such as rotation and translation. In addition,
they can be used as building blocks for other geometric objects using these
same operations. Raster primitives, such as arrays of pixels, lack geometric
properties and cannot be manipulated in space in the same way as geometric
primitives. They pass through a separate parallel pipeline on their way to the
frame buffer. We will defer our discussion of raster primitives until Chapter 8.

The basic OpenGL geometric primitives are specified by sets of ver-
tices. An application starts by computing vertex data—positions and other
attributes—and putting the results into arrays that are sent to the GPU for
display. When we want to display some geometry we execute functions whose
parameters specify how the vertices are to be interpreted. For example, we
can display the \( N \) vertices we computed for the Sierpinski gasket, starting
with the first vertex, as points through the function call

\[
glDrawArrays(GL_POINTS, 0, N);
\]

after they have been placed on the GPU.

All OpenGL geometric primitives are variants of points, line segments,
and triangular polygons. A point can be displayed as a single pixel or a
small group of pixels. Finite sections of lines between two vertices, called
line segments—in contrast to lines that are infinite in extent—are of great
importance in geometry and computer graphics. You can use line segments to
define approximations to curves, or you can use a sequence of line segments
to connect data values for a graph. You can also use line segments to
display the the edges of closed objects, such as polygons, that have interiors.
Consequently, it is often helpful to think in terms of both vertices and line
segments.

If we wish to display points or line segments, we have a few choices in
OpenGL (Figure 2.7). The primitives and their type specifications include
the following:

**Points (GL_POINTS)** Each vertex is displayed at a size of at least one pixel.
Line segments (GL_LINES) The line-segment type causes successive pairs of vertices to be interpreted as the endpoints of individual segments. Note that successive segments usually are disconnected because the vertices are processed on a pairwise basis.

Polylines (GL_LINE_STRIP, GL_LINE_LOOP) If successive vertices (and line segments) are to be connected, we can use the line strip, or polyline form. Many curves can be approximated via a suitable polyline. If we wish the polyline to be closed, we can locate the final vertex in the same place as the first, or we can use the GL_LINE_LOOP type, which will draw a line segment from the final vertex to the first, thus creating a closed path.

2.4.1 Polygon Basics

Line segments and polylines can model the edges of objects, but closed objects have interiors (Figure 2.8). Usually we reserve the name polygon for an object that has a border that can be described by a line loop but also has a well-defined interior. Polygons play a special role in computer graphics because we can display them rapidly and use them to approximate arbitrary surfaces. The performance of graphics systems is characterized by the number of polygons per second that can be rendered. We can render a polygon in a variety of ways: We can render only its edges; we can render its interior with a solid color or a pattern; and we can render or not render the edges, as shown in Figure 2.9. Although the outer edges of a polygon are defined easily by an ordered list of vertices, if the interior is not well defined, then the list of vertices may not be rendered at all or rendered in an undesirable manner. Three properties will ensure that a polygon will be displayed correctly: It must be simple, convex, and flat.

In two dimensions, as long as no two edges of a polygon cross each other, we have a simple polygon. As we can see in Figure 2.10, simple two-dimensional polygons have well-defined interiors. Although the locations of

FIGURE 2.8 Filled objects.

FIGURE 2.9 Methods of displaying a polygon.

2. The term fill area is sometimes used instead of polygon.

3. Measuring polygon rendering speeds involves both the number of vertices and the number of pixels inside.
the vertices determine whether or not a polygon is simple, the cost of testing is sufficiently high (see Exercise 2.12) that most graphics systems require that the application program does any necessary testing. We can ask what a graphics system will do if it is given a nonsimple polygon to display and whether there is a way to define an interior for a nonsimple polygon. We will examine these questions further in Chapter 8.

From the perspective of implementing a practical algorithm to fill the interior of a polygon, simplicity alone is often not enough. Some APIs guarantee a consistent fill from implementation to implementation only if the polygon is convex. An object is **convex** if all points on the line segment between any two points inside the object, or on its boundary, are inside the object. Thus, in Figure 2.11, $p_1$ and $p_2$ are arbitrary points inside a polygon and the entire line segment connecting them is inside the polygon. Although so far we have been dealing with only two-dimensional objects, this definition makes reference neither to the type of object nor to the number of dimensions. Convex objects include triangles, tetrahedra, rectangles, circles, spheres, and parallelepipeds (Figure 2.12). There are various tests for convexity (see Exercise 2.19). However, like simplicity testing, convexity testing is expensive and usually left to the application program.

In three dimensions, polygons present a few more difficulties because, unlike all two-dimensional objects, all the vertices that define the polygon need not lie in the same plane. One property that most graphics systems exploit, and that is the basis of OpenGL polygons, is that any three vertices that are not collinear determine both a triangle and the plane in which that triangle lies. Hence, if we always use triangles, we are safe—we can be sure that these objects will be rendered correctly. Often, we are almost forced to use triangles because typical rendering algorithms are guaranteed to be correct only if the vertices form a flat convex polygon. In addition, hardware and software often support a triangle type that is rendered faster than is a polygon with three vertices.
2.4.2 Polygons in OpenGL

Returning to the OpenGL types, the only OpenGL polygons (Figure 2.13) that OpenGL supports are triangles. Triangles can be displayed in three ways: as points corresponding to the vertices, as edges, or with the interiors filled. In OpenGL, we use the function `glPolygonMode` to tell the renderer to generate only the edges or just points for the vertices, instead of fill (the default). However, if we want to draw a polygon that is filled and to display its edges, then we have to render it twice, once in each mode, or to draw a filled polygon and a line loop with the same vertices.

Here are the types:

**Triangles (GL_TRIANGLES)** The edges are the same as they would be if we used line loops. Each successive group of three vertices specifies a new triangle.

**Strips and Fans (GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN)** These objects are based on groups of triangles that share vertices and edges. In the triangle strip, for example, each additional vertex is combined with the previous two vertices to define a new triangle (Figure 2.14). For the quadstrip, we combine two new vertices with the previous two vertices to define a new quadrilateral. A triangle fan is based on one fixed point. The next two points determine the first triangle, and subsequent triangles are formed from one new point, the previous point, and the first (fixed) point.

2.4.3 Approximating a Sphere

Fans and strips allow us to approximate many curved surfaces simply. For example, one way to construct an approximation to a sphere is to use a set of
polygons defined by lines of longitude and latitude as shown in Figure 2.15. We can do so very efficiently using either quad strips or triangle strips. Consider a unit sphere. We can describe it by the following three equations:

\[ x(\theta, \phi) = \sin \theta \cos \phi, \]
\[ y(\theta, \phi) = \cos \theta \cos \phi, \]
\[ z(\theta, \phi) = \sin \phi. \]

If we fix \( \theta \) and draw curves as we change \( \phi \), we get circles of constant longitude. Likewise, if we fix \( \phi \) and vary \( \theta \), we obtain circles of constant latitude. By generating points at fixed increments of \( \theta \) and \( \phi \), we can specify quadrilaterals as shown in Figure 2.15. However, because OpenGL supports triangles, not quadrilaterals, we generate the data for two triangles for each quadrilateral. Remembering that we must convert degrees to radians for the standard trigonometric functions, the code for the quadrilaterals corresponding to increments of 20 degrees in \( \theta \) and to 20 degrees in \( \phi \) is

```c
float quad_data[162][3];
float c=M_PI/180.0; /*degrees to radians, M_PI = 3.14159... */
int k = 0;
for(phi=-80.0; phi<=80.0; phi+=20.0)
{
    phir=c*phi;
    phir20=c*(phi+20);
    for(theta=-180.0; theta<=180.0;theta+=20.0)
    {
        thetar=c*theta;
        quad_data[k][0]=sin(thetar)*cos(phir);
        quad_data[k][1]=cos(thetar)*cos(phir);
        quad_data[k][s]=sin(phir);
        k++;
        quad_data[k][0]=sin(thetar)*cos(phir20);
        quad_data[k][1]=cos(thetar)*cos(phir20);
        quad_data[k][2]=sin(phir20);
        k++;
    }
}
```

Later we can render these data using `glDrawArrays(GL_LINE_LOOP,....)` or some other drawing function.

However, we have a problem at the poles, where we can no longer use strips because all lines of longitude converge there. We can, however, use two triangle fans, one at each pole as follows:

```c
float strip_data[18][3];
int k =1;
```
strip_data[0][0] = 0.0;
strip_data[0][1] = 0.0;
strip_data[0][2] = 1.0;
c=M_PI/180.0;
c80=c*80.0;
z=sin(c80);
for(theta =-180.0; theta<=180.0;theta+=20.0)
{
    thetar=c*theta;
    strip_data[k][0] =sin(thetar)*cos(c80);
    strip_data[k][1] =cos(thetar)*cos(c80);
    strip_data[k][2] = z;
    k++;
}

strip_data[k][0] = 0.0;
strip_data[k][1] = 0.0;
strip_data[k][2] = -1.0;
z=-sin(c80);
for(theta=-180.0;theta<=180.0;theta+=20.0)
{
    thetar=c*theta;
    strip_data[k][0] =sin(thetar)*cos(c80);
    strip_data[k][1] =cos(thetar)*cos(c80);
    strip_data[k][2] = z;
    k++;
}

These data could be rendered with `glDrawArrays(GL_TRIANGLE_FAN, ....)` or another drawing function. Note that because triangle fans are polygons, if we want to get the line segment display in Figure 2.15, we would first have to set the polygon mode to lines instead of fill.

### 2.4.4 Triangulation

We have been using the terms polygon and triangle somewhat interchangeably. If we are interested in objects with interiors, general polygons are problematic. A set of vertices may not all lie in the same plane or specify a polygon which is neither simple nor convex. Such problems do not arise with triangles. As long as the the three vertices of a triangle are not collinear its interior is well defined and the triangle is simple, flat, and convex. Consequently, triangles are easy to render and for these reasons triangles are the only fillable geometric entity that OpenGL recognizes. In practice we need to deal with more general polygons. The usual strategy is to start with a list of vertices and generate a set of triangles consistent with the polygon defined by the list, a process known as **triangulation**.
2.4 Primitives and Attributes

2.4.5 Text

Graphical output in applications such as data analysis and display requires annotation, such as labels on graphs. Although, in nongraphical programs, textual output is the norm, text in computer graphics is problematic. In non-graphical applications, we are usually content with a simple set of characters,
always displayed in the same manner. In computer graphics, however, we often wish to display text in a multitude of fashions by controlling type styles, sizes, colors, and other parameters. We also want to have available a choice of fonts. Fonts are families of typefaces of a particular style, such as Times, Computer Modern, or Helvetica.

There are two forms of text: stroke and raster. Stroke text (Figure 2.20) is constructed as are other geometric objects. We use vertices to define line segments or curves that outline each character. If the characters are defined by closed boundaries, we can fill them. The advantage of stroke text is that it can be defined to have all the detail of any other object, and because it is defined in the same way as other graphical objects are, it can be manipulated by our standard transformations and viewed like any other graphical primitive. Using transformations, we can make a stroke character bigger or rotate it, retaining its detail and appearance. Consequently, we need to define a character only once, and we can use transformations to generate it at the desired size and orientation.

Defining a full 128- or 256-character stroke font, however, can be complex, and the font can take up significant memory and processing time. The standard PostScript fonts are defined by polynomial curves, and they illustrate all the advantages and disadvantages of stroke text. The various PostScript fonts can be used for both high- and low-resolution applications. Often, developers mitigate the problem of slow rendering of such stroke characters by putting considerable processing power in the printer. This strategy is related to the client–server concepts that we will discuss in Chapter 3.

Raster text (Figure 2.21) is simple and fast. Characters are defined as rectangles of bits called bit blocks. Each block defines a single character by the pattern of 0 and 1 bits in the block. A raster character can be placed in the frame buffer rapidly by a bit-block-transfer (bitblt) operation, which moves the block of bits using a single function call. We will discuss bitblt in Chapter 8: OpenGL allows the application program to use functions that support direct manipulation of the contents of the frame buffer.

You can increase the size of raster characters by replicating or duplicating pixels, a process that gives larger characters a blocky appearance.
2.4 Primitives and Attributes

(Figure 2.22). Other transformations of raster characters, such as rotation, may not make sense, because the transformation may move the bits defining the character to locations that do not correspond to the location of pixels in the frame buffer.

Because stroke and bitmap characters can be created from other primitives, OpenGL does not have a text primitive. However, the GLUT library provides a few predefined bitmap and stroke character sets that are defined in software and are portable. For example, we can put out a bitmap character that is $8 \times 13$ pixels by

\[
glutBitmapCharacter(GLUT_BITMAP_8_BY_13, c)
\]

where $c$ is the number of the ASCII character that we wish to be placed on the display.

2.4.6 Curved Objects

The primitives in our basic set have all been defined through vertices. With the exception of the point type, all consist of line segments or use line segments to define the boundary of a region that can be filled with a solid color or a pattern. We can take two approaches to creating a richer set of objects.

First, we can use the primitives that we have to approximate curves and surfaces. For example, if we want a circle, we can use a regular polygon of $n$ sides. Likewise, we have approximated a sphere with triangles and quadrilaterals. More generally, we approximate a curved surface by a mesh of convex polygons—a tessellation—which can occur either at the rendering stage or within the user program.

The other approach, which we will explore in Chapter 11, is to start with the mathematical definitions of curved objects, and then build graphics functions to implement those objects. Objects such as quadric surfaces and parametric polynomial curves and surfaces are well understood mathematically, and we can specify them through sets of vertices. For example, we can define a sphere by its center and a point on its surface, or we can define a cubic polynomial curve by four points.
2.4.7 Attributes

Although we can describe a geometric object through a set of vertices, a given object can be displayed in many different ways. Properties that describe how an object should be rendered are called attributes. Available attributes depend on the type of object. For example, a line could be black or green. It could be solid or dashed. A polygon could be filled with a single color or with a pattern. We could display it as filled or only by its edges. Several of these attributes are shown in Figure 2.23 for lines and polygons.

Attributes may be associated with, or bound to geometric objects, such as the color of a cube. Often we will find it better to model a object such as the cube by its individual faces and to specify attributes for the faces. Hence, a cube would be green because its six faces are green. Each face could than be described by two triangles so ultimately a green cube would be rendered as 12 green triangles.

If we go one step further, we see that each of the triangles is specified through three vertices. In a pipeline architecture, each vertex processed independently through a vertex shader. Hence, we can associate properties with each vertex. For example, if we assign a different color to each vertex of a polygon, the rasterizer can interpolate these colors to obtain different colors for each fragment. These vertex attributes may also be dependent on the application. For example, in a simulation of heat distribution of some object, the application might determine a temperature for each vertex defining the object. In Chapter 3, we will include vertex attribute data in the array with our vertex locations that is sent to the GPU.

In systems that use immediate-mode graphics and a pipeline architecture, some attributes are part of the state of the graphics systems. Hence, there would be a current color that would be used to render all primitives until changed by some state changing function such as

```c
set_current_color(color);
```
Likewise, there would attribute setting functions for a variety of attributes.  

Each geometric type has a set of attributes. For example, a point has a 
color attribute and a size attribute. Line segments can have color, thickness, 
and pattern (solid, dashed, or dotted). Filled primitives, such as polygons, 
have more attributes because we must use multiple parameters to specify how 
the fill should be done. We can fill with a solid color or a pattern. We can 
decide not to fill the polygon and to display only its edges. If we fill the 
polygon, we might also display the edges in a color different from that of the 
interior.

In systems that support stroke text as a primitive, there is a variety of 
attributes. Some of these attributes are demonstrated in Figure 2.24; they 
include the direction of the text string, the path followed by successive 
characters in the string, the height and width of the characters, the font, 
and the style (bold, italic, underlined).

Although the notion of current state works well for interactive applica
tions, it is inconsistent with our physical intuition. A box is green or red. It 
either contains a pattern on its surfaces or it doesn’t. Object-oriented graph
cs takes a fundamentally different approach in which attributes are part of 
a geometric object. In Chapter 10, we will discuss scene graphs, which are 
fundamental to systems such as Open Scene Graph, and we will see that they 
provide another higher-level object-oriented approach to computer graphics.

4. Earlier versions of OpenGL contained state setting functions such as `glColor`, `glLineWidth`, and `glStipple`. 
2.5 COLOR

Color is one of the most interesting aspects of both human perception and computer graphics. We can use the model of the human visual system from Chapter 1 to obtain a simple but useful color model. Full exploitation of the capabilities of the human visual system using computer graphics requires a far deeper understanding of the human anatomy, physiology, and psychophysics. We will present a more sophisticated development in Chapter 7.

A visible color can be characterized by a function \( C(\lambda) \) that occupies wavelengths from about 350 to 780 nm, as shown in Figure 2.25. The value for a given wavelength \( \lambda \) in the visible spectrum gives the intensity of that wavelength in the color.

Although this characterization is accurate in terms of a physical color whose properties we can measure, it does not take into account how we perceive color. As noted in Chapter 1, the human visual system has three types of cones responsible for color vision. Hence, our brains do not receive the entire distribution \( C(\lambda) \) for a given color but rather three values—the tristimulus values—that are the responses of the three types of cones to the color. This reduction of a color to three values leads to the basic tenet of three-color theory: *If two colors produce the same tristimulus values, then they are visually indistinguishable.*

A consequence of this tenet is that, in principle, a display needs only three primary colors to produce the three tristimulus values needed for a human observer. We vary the intensity of each primary to produce a color as we saw for the CRT in Chapter 1. The CRT is one example of additive color where the primary colors add together to give the perceived color. Other examples of additive color include projectors and slide (positive) film. In such systems, the primaries are usually red, green, and blue. With additive color, primaries add light to an initially black display, yielding the desired color.

For processes such as commercial printing and painting, a subtractive color model is more appropriate. Here we start with a white surface, such as a sheet of paper. Colored pigments remove color components from light that is striking the surface. If we assume that white light hits the surface, a particular point will be red if all components of the incoming light are absorbed by the surface except for wavelengths in the red part of the spectrum, which are reflected. In subtractive systems, the primaries are usually the complementary colors: cyan, magenta, and yellow (CMY; Figure 2.26). We will not explore subtractive color here. You need to know only that an RGB additive system has a dual with a CMY subtractive system (see Exercise 2.8).

We can view a color as a point in a color solid, as shown in Figure 2.27 and in Color Plate 21. We draw the solid using a coordinate system corresponding to the three primaries. The distance along a coordinate axis represents the amount of the corresponding primary in the color. If we normalize the maximum value of each primary to be 1, then we can represent any color...
that we can produce with this set of primaries as a point in a unit cube. The vertices of the cube correspond to black (no primaries on); red, green, and blue (one primary fully on); the pairs of primaries, cyan (green and blue fully on), magenta (red and blue fully on), and yellow (red and green fully on); and white (all primaries fully on). The principal diagonal of the cube connects the origin (black) with white. All colors along this line have equal tristimulus values and appear as shades of gray.

There are many matters that we are not exploring fully here and will return to in Chapter 7. Most concern the differences among various sets of primaries or the limitations conferred by the physical constraints of real devices. In particular, the set of colors produced by one device—its color gamut—is not the same as for other devices, nor will it match the human’s color gamut. In addition, the tristimulus values used on one device will not produce the same visible color as the same tristimulus values used on another device.
2.5.1 RGB Color

Now we can look at how color is handled in a graphics system from the programmer’s perspective—that is, through the API. There are two different approaches. We will stress the **RGB-color model** because an understanding of it will be crucial for our later discussion of shading. Historically, the **indexed-color model** (Section 2.5.2) was easier to support in hardware because of its lower memory requirements and the limited colors available on displays, but in modern systems RGB color has become the norm.

In a three-primary-color, additive-color RGB system, there are conceptually separate buffers for red, green, and blue images. Each pixel has separate red, green, and blue components that correspond to locations in memory (Figure 2.28). In a typical system, there might be a $1280 \times 1024$ array of pixels, and each pixel might consist of 24 bits (3 bytes): 1 byte for each of red, green, and blue. With present commodity graphics cards having from 128MB to 512MB of memory, there is no longer a problem of storing and displaying the contents of the frame buffer at video rates.

As programmers, we would like to be able to specify any color that can be stored in the frame buffer. For our 24-bit example, there are $2^{24}$ possible colors, sometimes referred to as 16M colors, where M denotes $1024^2$. Other systems may have as many as 12 (or more) bits per color or as few as 4 bits per color. Because our API should be independent of the particulars of the hardware, we would like to specify a color independently of the number of bits in the frame buffer and to let the drivers and hardware match our specification as closely as possible to the available display. A natural technique is to use the color cube and to specify color components as numbers between 0.0 and 1.0, where 1.0 denotes the maximum (or saturated value) of the corresponding primary, and 0.0 denotes a zero value of that primary.

In applications in which we want to assign a color to each vertex, we can put colors into a separate data structure, such as

```c
float colors[3][] = {{1.0, 0.0, 0.0}, {0.0, 1.0, 0.0}, {0.0, 0.0, 1.0}}
```
which holds the colors red, green, and blue or we could create a single array that contains both vertex locations and vertex colors. These data can be sent to the shaders where colors will be applied to pixels in the frame buffer.

Later, we shall be interested in a four-color (RGBA) system. The fourth color (A, or \textit{alpha}) also is stored in the frame buffer as are the RGB values; it can be set with four-dimensional versions of the color functions. In Chapter 7, we will see various uses for alpha, such as for creating fog effects or combining images. Here we need to specify the alpha value as part of the initialization of an OpenGL program. If blending is enabled (Chapter 8), then the alpha value will be treated by OpenGL as either an \textit{opacity} or \textit{transparency} value. Transparency and opacity are complements of each other. An opaque object passes no light through it; a transparent object passes all light. Opacity values can range from fully transparent (A=0.0) to fully opaque (A=1.0).

One of the first tasks that we must do in a program is to clear an area of the screen—a drawing window—in which to display our output. We also must clear this window whenever we want to draw a new frame. By using the four-dimensional (RGBA) color system, we can create effects where the drawing window interacts with other windows that may be beneath it by manipulating the opacity assigned to the window when it is cleared. The function call

\begin{verbatim}
glClearColor(1.0, 1.0, 1.0, 1.0);
\end{verbatim}

defines an RGB-color clearing color that is white, because the first three components are set to 1.0, and is opaque, because the alpha component is 1.0. We can then use the function \texttt{glClear} to make the window on the screen solid and white. Note that by default blending is not enabled. Consequently, the alpha value can be set in \texttt{glClearColor} to a value other than 1.0 and the default window will still be opaque.

\subsection{2.5.2 Indexed Color}

Early graphics systems had frame buffers that were limited in depth. For example, we might have had a frame buffer with a spatial resolution of 1280 × 1024, but each pixel was only 8 bits deep. We could divide each pixel’s 8 bits into smaller groups of bits and assign red, green, and blue to each. Although this technique was adequate in a few applications, it usually did not give us enough flexibility with color assignment. Indexed color provided a solution that allowed applications to display a wide range of colors as long as the application did not need more colors than could be referenced by a pixel. This technique is still used today.

We follow an analogy with an artist who paints in oils. The oil painter can produce an almost infinite number of colors by mixing together a limited number of pigments from tubes. We say that the painter has a potentially large color \textit{palette}. At any one time, however, perhaps due to a limited number of brushes, the painter uses only a few colors. In this fashion, she can create
an image that, although it contains a small number of colors, expresses her
choices because she can select the few colors from a large palette.

Returning to the computer model, we can argue that if we can choose
for each application a limited number of colors from a large selection (our
palette), we should be able to create good-quality images most of the time.

We can select colors by interpreting our limited-depth pixels as indices
into a table of colors rather than as color values. Suppose that our frame
buffer has \( k \) bits per pixel. Each pixel value or index is an integer between
0 and \( 2^k - 1 \). Suppose that we can display colors with a precision of \( m \) bits;
that is, we can choose from \( 2^m \) reds, \( 2^m \) greens, and \( 2^m \) blues. Hence, we can
produce any of \( 2^{3m} \) colors on the display, but the frame buffer can specify
only \( 2^k \) of them. We handle the specification through a user-defined **color-
lookup table** that is of size \( 2^k \times 3m \) (Figure 2.29). The user program fills the
\( 2^k \) entries (rows) of the table with the desired colors, using \( m \) bits for each of
red, green, and blue. Once the user has constructed the table, she can specify
a color by its index, which points to the appropriate entry in the color-lookup
table (Figure 2.30). For \( k = m = 8 \), a common configuration, she can choose
256 out of 16 M colors. The 256 entries in the table constitute the user’s color
palette.

\[
\begin{array}{cccc}
\text{Input} & \text{Red} & \text{Green} & \text{Blue} \\
0 & 0 & 0 & 0 \\
1 & 2^m & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots \\
2^k & 1 & 2^m & 0 \\
\end{array}
\]

\begin{center}
FIGURE 2.29 Color-lookup table.
\end{center}

\[
\begin{array}{ccc}
\text{Frame buffer} & \text{Color-lookup table} & \text{Display} \\
& \text{Red} & \text{Green} & \text{Blue} \\
\end{array}
\]

\begin{center}
FIGURE 2.30 Indexed color.
\end{center}
If we are in color-index mode, the present color is selected by a function that selects a particular color out of the table. Setting and changing the entries in the color-lookup table involves interacting with the window system. One difficulty arises if the window system and underlying hardware support only a limited number of colors because the window system may have only a single color table that must be used for all its windows, or it might have to juggle multiple tables, one for each window on the screen.

Historically, color-index mode was important because it required less memory for the frame buffer and fewer other hardware components. However, cost is no longer an issue and color-index mode presents a few problems. When we work with dynamic images that must be shaded, usually we need more colors than are provided by color-index mode. In addition, the interaction with the window system is also more complex than with RGB color. Consequently, for the most part, we will assume that we are using RGB color.

2.5.3 Setting of Color Attributes
For our simple example program, we use RGB color. We have three attributes to set. The first is the clear color, which is set to white by the following function call:

```c
glClearColor(1.0, 1.0, 1.0, 1.0);
```

Note this function uses RGBA color.

The color we use to render points is set in the shaders. We can set an RGB color in the application such as

```c
GLfloat point_color[] = (1.0, 0.0, 0.0);
```

or as an RGBA color as

```c
GLfloat point_color[] = (1.0, 0.0, 0.0, 1.0);
```

and send this color to the vertex shader. We could also set the color totally in the shader. We will see a few options later in this chapter. We can set the size of our rendered points to be 2 pixels wide, by using the following OpenGL function:

```c
glPointSize(2.0);
```

Note that attributes, such as the point size[^5] and line width, are specified in terms of the pixel size. Hence, if two displays have different-sized pixels

[^5]: Note that point size is one of the few state variables predefined in the latest versions of OpenGL
(due to their particular screen dimensions and resolutions), then the rendered images may appear slightly different. Certain graphics APIs, in an attempt to ensure that identical displays will be produced on all systems with the same user program, specify all attributes in a device-independent manner. Unfortunately, ensuring that two systems produce the same display has proved to be a difficult implementation problem. OpenGL has chosen a more practical balance between desired behavior and realistic constraints.

2.6 VIEWING

We can now put a variety of graphical information into our world, and we can describe how we would like these objects to appear, but we do not yet have a method for specifying exactly which of these objects should appear on the screen. Just as what we record in a photograph depends on where we point the camera and what lens we use, we have to make similar viewing decisions in our program.

A fundamental concept that emerges from the synthetic-camera model that we introduced in Chapter 1 is that the specification of the objects in our scene is completely independent of our specification of the camera. Once we have specified both the scene and the camera, we can compose an image. The camera forms an image by exposing the film, whereas the computer system forms an image by carrying out a sequence of operations in its pipeline. The application program needs to worry only about the specification of the parameters for the objects and the camera, just as the casual photographer is concerned about the resulting picture, not about how the shutter works or the details of the photochemical interaction of film with light.

There are default viewing conditions in computer image formation that are similar to the settings on a basic camera with a fixed lens. However, a camera that has a fixed lens and sits in a fixed location forces us to distort our world to take a picture. We can create pictures of elephants only if we place the animals sufficiently far from the camera, or we can photograph ants only if we put the insects relatively close to the lens. We prefer to have the flexibility to change the lens to make it easier to form an image of a collection of objects. The same is true when we use our graphics system.

2.6.1 The Orthographic View

The simplest and OpenGL’s default view is the orthographic projection. We discuss this projection and others in detail in Chapter 4, but we introduce the orthographic projection here so that you can get started writing three-dimensional programs. Mathematically, the orthographic projection is what we would get if the camera in our synthetic-camera model had an infinitely long telephoto lens and we could then place the camera infinitely far from our objects. We can approximate this effect, as shown in Figure 2.31, by leaving the image plane fixed and moving the camera far from this plane. In the limit,
all the projectors become parallel and the center of projection is replaced by a direction of projection.

Rather than worrying about cameras an infinite distance away, suppose that we start with projectors that are parallel to the positive z-axis and the projection plane at $z = 0$, as shown in Figure 2.32. Note that not only are the
projectors perpendicular or orthogonal to the projection plane, but also we can slide the projection plane along the \( z \)-axis without changing where the projectors intersect this plane.

For orthographic viewing, we can think of there being a special orthographic camera that resides in the projection plane, something that is not possible for other views. Perhaps more accurately stated, there is a reference point in the projection plane from which we can make measurements of a view volume and a direction of projection. In OpenGL, the reference point starts off at the origin and the camera points in the negative \( z \)-direction, as shown in Figure 2.33. The orthographic projection takes a point \((x, y, z)\) and projects it into the point \((x, y, 0)\), as shown in Figure 2.34. Note that if we are working in two dimensions with all vertices in the plane \( z = 0 \), a point and its projection are the same; however, we can employ the machinery of a three-dimensional graphics system to produce our image.
In OpenGL, an orthographic projection with a right-parallelepiped viewing volume is the default. The volume is the cube defined by the planes
\[ x = \pm 1, \]
\[ y = \pm 1, \]
\[ x = \pm 1. \]

The orthographic projection "sees" only those objects in the volume specified by this viewing volume. Unlike a real camera, the orthographic projection can include objects behind the camera. Thus, as long as the plane \( z = 0 \) is located between near and far, the two-dimensional plane will intersect the viewing volume.

In Chapters 3 and 4, we will learn to use transformations to create other views. For now, we will scale and position our objects so those that we wish to view are inside the default volume.

### 2.6.2 Two-Dimensional Viewing

Remember that, in our view, two-dimensional graphics is a special case of three-dimensional graphics. Our viewing area is in the plane \( z = 0 \) within a three-dimensional viewing volume, as shown in Figure 2.35.

We could also consider two-dimensional viewing directly by taking a rectangular area of our two-dimensional world and transferring its contents to the display, as shown in Figure 2.36. The area of the world that we image is known as the viewing rectangle, or clipping rectangle. Objects inside the rectangle are in the image; objects outside are clipped out and are not displayed. Objects that straddle the edges of the rectangle are partially visible in the image. The size of the window on the display and where this window is placed on the display are independent decisions that we examine in Section 2.7.
2.7 CONTROL FUNCTIONS

We are almost done with our first program, but we must discuss the minimal interactions with the window and operating systems. If we look at the details for a specific environment, such as the X Window System on a Linux platform or Windows on a PC, we see that the programmer’s interface between the graphics system and the operating and window systems can be complex. Exploitation of the possibilities open to the application programmer requires knowledge specific to these systems. In addition, the details can be different for two different environments, and discussing these differences will do little to enhance our understanding of computer graphics.

Rather than deal with these issues in detail, we look at a minimal set of operations that must take place from the perspective of the graphics application program. Earlier we discussed the OpenGL Utility Toolkit (GLUT); it is a library of functions that provides a simple interface between the systems. Details specific to the underlying windowing or operating system are inside the implementation, rather than being part of its API. Operationally, we add another library to our standard library search path. GLUT will help us to understand the interactions that characterize modern interactive graphics systems, including a wide range of APIs, operating systems, and window systems. The application programs that we produce using GLUT should run under multiple window systems.

2.7.1 Interaction with the Window System

The term *window* is used in a number of different ways in the graphics and workstation literature. We use *window*, or *screen window*, to denote a rectangular area of our display. We are concerned only with raster displays. A window has a height and width, and because the window displays the contents
of the frame buffer, positions in the window are measured in \textit{window} or \textit{screen coordinates},\footnote{In OpenGL, window coordinates are three-dimensional, whereas screen coordinates are two-dimensional. Both systems use units measured in pixels, but window coordinates retain depth information.} where the units are pixels.

In a modern environment, we can display many windows on the monitor. Each can have a different purpose, ranging from editing a file to monitoring our system. We use the term \textit{window system} to refer to the multiwindow environment provided by systems such as the X Window System and Microsoft Windows. The window in which the graphics output appears is one of the windows managed by the window system. Hence, to the window system, the graphics window is a particular type of window—one in which graphics can be displayed or rendered. References to positions in this window are relative to one corner of the window. We have to be careful about which corner is the origin. In science and engineering, the lower-left corner is the origin and has window coordinates (0,0). However, virtually all raster systems display their screens in the same way as commercial television systems do—from top to bottom, left to right. From this perspective, the top-left corner should be the origin. Our OpenGL commands assume that the origin is bottom left, whereas information returned from the windowing system, such as the mouse position, often has the origin at the top left and thus requires us to convert the position from one coordinate system to the other.

Although our screen may have a resolution of, say, $1280 \times 1024$ pixels, the window that we use can have any size. Thus, the frame buffer must have a resolution equal to the screen size. Conceptually, if we use a window of $300 \times 400$ pixels, we can think of it as corresponding to a $300 \times 400$ frame buffer, even though it uses only a part of the real frame buffer.

Before we can open a window, there must be interaction between the windowing system and OpenGL. In GLUT, this interaction is initiated by the following function call:

\begin{verbatim}
glutInit(int *argcv, char **argv)
\end{verbatim}

The two arguments allow the user to pass command-line arguments, as in the standard C \texttt{main} function, and are usually the same as in \texttt{main}. We can now open an OpenGL window using the GLUT function

\begin{verbatim}
glutCreateWindow(char *title)
\end{verbatim}

where the title at the top of the window is given by the string \texttt{title}.

The window that we create has a default size, a position on the screen, and characteristics such as the use of RGB color. We can also use GLUT functions before window creation to specify these parameters. For example, the code
specifies a 480 × 640 window in the top-left corner of the display. We specify RGB rather than indexed (GLUT_INDEX) color, a depth buffer for hidden-surface removal, and double rather than single (GLUT_SINGLE) buffering. The defaults, which are all we need for now, are RGB color, no hidden-surface removal, and single buffering. Thus, we do not need to request these options explicitly, but specifying them makes the code clearer. Note that parameters are logically OR’ed together in the argument to glutInitDisplayMode.

2.7.2 Aspect Ratio and Viewports

The aspect ratio of a rectangle is the ratio of the rectangle’s width to its height. The independence of the object, viewing, and workstation window specifications can cause undesirable side effects if the aspect ratio of the viewing rectangle, specified by glutOrtho, is not the same as the aspect ratio of the window specified by glutInitWindowSize. If they differ, as depicted in Figure 2.37, objects are distorted on the screen. This distortion is a consequence of our default mode of operation, in which the entire clipping rectangle is mapped to the display window. The only way that we can map the entire contents of the clipping rectangle to the entire display window is to distort the contents of the former to fit inside the latter. We can avoid this distortion if we ensure that the clipping rectangle and display window have the same aspect ratio.

Another, more flexible, method is to use the concept of a viewport. A viewport is a rectangular area of the display window. By default, it is the entire window, but it can be set to any smaller size in pixels via the function.
2.7 Control Functions

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

where \((x,y)\) is the lower-left corner of the viewport (measured relative to the lower-left corner of the window), and \(w\) and \(h\) give the height and width, respectively. The types are all integers that allow us to specify positions and distances in pixels. Primitives are displayed in the viewport, as shown in Figure 2.38. For a given window, we can adjust the height and width of the viewport to match the aspect ratio of the clipping rectangle, thus preventing any object distortion in the image.

The viewport is part of the state. If we change the viewport between rendering objects or rerendering the same objects with the viewport changed, we achieve the effect of multiple viewports with different images in different parts of the window. We will see further uses of the viewport in Chapter 3, where we consider interactive changes in the size and shape of the window.

2.7.3 The main, display, and myinit Functions

In principle, we should be able to combine the simple initialization code with our code from Section 2.1 to form a complete OpenGL program that generates the Sierpinski gasket. Unfortunately, life in a modern system is not that simple. There are two problems: One is generic to all graphics systems; the second has more to do with problems of interacting with the underlying windowing system.

Our basic mechanism for display will be to form a data structure that contains all the geometry and attributes we need to specify a primitive and how we would like it displayed. We then send this structure to the shaders which will process our data and display the results. Once the application has sent the data to the shaders, it is free to do other tasks. In an interactive application, we would continue to generate more primitives.

However, for an application such as our sample program, we draw a few primitives and are finished. As the application ends, the application window
will disappear from the display before we have had a chance to see our output. A simple solution for this problem might be to insert a delay, for example, via a standard function such as `sleep(Enough_time)` to give us enough time to view our output. For any but the most trivial applications, however, we need a more sophisticated mechanism.

The mechanism employed by most graphics and window systems is to use **event processing**, which gives us interactive control in our programs. **Events** are changes that are detected by the operating system and include such actions as a user pressing a key on the keyboard, the user clicking a mouse button or moving the mouse, or the user iconifying a window on the display. Events are varied and usually only a subset of them is important to graphics applications. An event may generated data that is stored with the occurrence of the event. For example, if a key is pressed, the code for the key will be stored.

When events occur they are placed in queue, the **event queue**, that can be examined by an application program or by the operating system. A given event can be ignored or cause an action to take place. For example, an application that does not use the keyboard will ignore all pressing and releasing of keys whereas an application that uses the keyboard might use keyboard events to control the flow of the application.

With GLUT, we can execute the function function

```c
void glutMainLoop();
```

to begin an event-processing loop. If there are no events to process, the program will sit in a wait state, with our graphics on the screen, until we terminate the program through some external means—say, by hitting a special key or a combination of keys, such as control-c—that terminates the execution of the program.

If there are events in the queue, our program responds to them through functions called **callbacks**. A callback function is associated with a specific type of event. Hence a typical interactive application would use a mouse callback and perhaps a keyboard callback. For our simple example, we need only a single callback called the **display callback**. A display callback is generated when the application program or the operating system determines that the graphics in a window need to be redrawn. One of these times is when during initialization, the application creates a window on the display. Hence, virtually every program must have a display callback function that is executed when the callback occurs.

The display callback function is named through the GLUT function

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

and registered with the window system. Here the function named `func` will be called whenever the windowing system determines that the OpenGL window needs to be redisplayed. Because one of these times is when the window is first
opened, if we put all our graphics into this function (for our noninteractive example), func will be executed once and our gasket will be drawn. Although it may appear that our use of the display function is merely a convenience for organizing our program, the display function is required by GLUT. A display callback also occurs, for example, when the window is moved from one location on the screen to another and when a window in front of the OpenGL window is destroyed, making visible the whole OpenGL window.

Following is a main function that works for most noninteractive applications:

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

void main(int argc, char **argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500,500);
    glutInitWindowPosition(0,0);
    glutCreateWindow(“simple OpenGL example”);
    glutDisplayFunc(display);
    myinit();
    glutMainLoop();
}
```

We use an initialization function myinit() to set the OpenGL state variables dealing with viewing and attributes—parameters that we prefer to set once, independently of the display function. The standard include (.h) file for GLUT is loaded before the beginning of the function definitions. In most implementations, the compiler directive

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

will add in the header files for the GLUT library, the OpenGL library (gl.h), and the OpenGL utility library (glu.h). The macro definitions for our standard values, such as GL_LINES and GL_RGB, are in these files.

### 2.7.4 Program Structure

Every program we write will have a similar structure to our gasket program. We will always use the GLUT toolkit. The main function will then consist of calls to GLUT functions to set up our window(s) and to make sure that the local environment supports the required display properties. The main

7. We hope to avoid confusion by using the same function names as those in the *OpenGL Programming Guide* [Ope07] and in the GLUT documentation [Kil94a].
will also name the required callbacks and callback functions. Every program
must have a display callback, and most will have other callbacks to set up
interaction. The myinit function will set up user options, usually through
OpenGL functions in the GL library. Although these options could be set in
main, it is clearer to keep GLUT functions separate from OpenGL functions.
In the majority of programs, the graphics output will be generated in the
display callback.

Every application, no matter how simple, must provide both a vertex
shader and a fragment shader. Setting up the shaders requires a numbers of
steps including reading the shader code from files, compiling the code, and
linking the shaders with the application. These steps are almost identical for
most applications. Hence, we will put this code into a function initShaders.
These operations require a handful of OpenGL functions that have little to
do with graphics. Consequently, we place the details of these functions are in
Appendix AA.

2.8 THE GASKET PROGRAM

We can now complete our gasket program. We have already created the
points and put them in an array. Now we have to get these data to our GPU
and render them. We start by creating a buffer object on the GPU and placing
our data in that object. We need three functions that we can call after we
have generated our points:

```
GLUnit buff;

glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW);
```

First we use glGenBuffers to give us an unused identifier for our buffer
object that is put into the variable buffer. The function glBindBuffer
creates the buffer with the identifier from glGenBuffers. The type GL_ARRAY_-
BUFFER, indicates that the data in the buffer will be arrays rather than some
one of the other storage types that we will encounter later. Finally, with
glBufferData, we allocate sufficient memory on the GPU for our data and
provide a pointer to the array holding the data. Once data is in GPU memory,
we might, as in this example, simply display it once. But in more realistic
applications we might alter the data, redisplay it many times, and even read
data back from the GPU to the CPU. Modern GPUs can alter how they store
data to increase efficiency depending on the type of application. The final
parameter in glBufferData gives a hint of how the application plans to use
the date. In our case, we are sending it once and displaying it so the choice of
GL_STATIC_DRAW is appropriate. The code to compute the points and create
the buffer object can be part of initialization.
2.8 The Gasket Program

2.8.1 Rendering the Points

When we want to display our points, we can use the function

\[ \text{glDrawArrays(GL_POINTS, 0, N);} \]

causes \( N \) data to be rendered starting with the first point. The value of the first parameter, \( \text{GL_POINTS} \), tells the GPU we want to data to be used to display distinct points rather than other primitives such as lines or polygons that could be describe by the same data. Thus, a simple display callback is

\[ \text{void my display()} \{
    \text{glClear(GL_COLOR_BUFFER_BIT);} \\
    \text{glDrawArrays(GL_POINTS, 0, N);} \\
    \text{glFlush();} 
\} \]

We clear the frame buffer and then render the point data that is on the GPU. The \text{glFlush} ensures that all the data are rendered as soon as possible. If you leave it out, the program should work correctly but you notice a delay in a busy or networked environment.

But this is just the beginning of the story. The rendering process must be carried out by the pipeline of the vertex shader, the rasterizer, and the fragment shader in order to get the proper pixels displayed in the frame buffer. Because our example uses only points, we can develop very simple shaders and put together the whole application. Even though our shader will be almost trivial, we must provide both a vertex shader and fragment shader to have a complete application. There are no default shaders.

2.8.2 The Vertex Shader

The only information that we put in our buffer object is just the location each point. When we execute \text{glDrawArrays} each of the \( N \) vertices generates the execution of a vertex shader that we must provide. If we leave the color determination to the fragment shader, all the vertex shader must do is pass the vertex’s location to the rasterizer.

We write our shader using the OpenGL Shading Language (GLSL), which is a C-like language with which we can write both vertex and fragment shaders. We will discuss GLSL in more detail later when want to write more sophisticated shader but here is the code for a simple \text{pass-through} vertex shader

\[ \text{in vec4 vPosition;} \]

\[ \text{void main()} \{
\]
Chapter 2 Graphics Programming

```c
    gl_Position = vPosition;
}
```

Each shader is a main program. GLSL expands the C data types to include matrix and vector types. The input vertex's location is given by the four dimensional vector \( \mathbf{vPosition} \) whose specification includes the keyword \texttt{in} to signify that its value is input to the shader when the shader is initiated. There is one special state variable in our shader: \texttt{gl_Position}, which is known to OpenGL, and is the required output of every vertex shader. Because we specified the values in our application in clip coordinates, our shader does not have to make any changes to the values input to the shader and merely passes them through via \texttt{gl_Position}.

We still have to establish a connection between the array \texttt{points} in the application and the input array \texttt{vPosition} in the shader. We will do this after we compile and link our shaders. First, we look at the fragment shader.

### 2.8.3 The Fragment Shader

Each invocation of the vertex shader outputs a vertex which then goes through primitive assembly and clipping before reaching the rasterizer. The rasterizer outputs fragments for each primitive inside the clipping volume. Each fragment invokes an execution of the fragment shader. At a minimum, each execution of the fragment shader must output a color for the fragment unless the fragment is to discarded. Here is a minimum GLSL fragment shader:

```c
void main()
    	gl_FragColor = vec4 ( 1.0, 0.0, 0.0, 1.0 );
```

All this shader does is assign a four dimensional RGBA color to each fragment through the built-in variable \texttt{gl_FragColor}. The A component of the color is its opacity. We want our points to be opaque and not translucent so we use \( A = 1.0 \). Setting R to 1.0 and the other two components to 0.0 colors each fragment red.

### 2.8.4 Combining the Parts

We now have the pieces but need to put them together. In particular, we have to compile the shaders, connect variables in the application with their counterparts in the shaders, and link everything together. We start with the bare minimum. Shaders must be compiled and linked. Most of the time we will do these operations as part of initialization so we can put the necessary code in a function \texttt{initShaders} which will remain almost unchanged from application to application.
2.8.5 The initShader function

A typical application contains three distinct parts: the application program, which comprises a main function and other functions such as init, a vertex shader and a fragment shader. The first part is a set of C (or C++) functions whereas the shaders are written in GLSL. To obtain a module that we can execute, we have to connect these entities, a process that involves reading source code from files, compiling the individual parts, and linking everything together. We can control this process through our application using a set of OpenGL functions that we will discuss in detail in Chapter 3. Here it will be sufficient to describe the steps briefly.

Our first step is to create a container called a program object to hold our shaders and two shader objects, one for each type of shader. The program object has an integer identifier we can use to refer to it in the application. After we create these objects, we can attach the shaders to the program object. Generally, the shader source code will be in standard text files files. We read them into strings which can be attached to the program and compiled. If the compilation is successful, the application and shaders can be linked together. Assuming we have the vertex shader source in a file vshader.glsl and the fragment shader in a file fshader.glsl, we can execute the above steps by a function call of the form

```c
GLuint program;

program = initShader("vsource.glsl", "fsource.glsl");
```

in the main function of the application.

When we link the program object and the shaders, the names of shader variables are bound to indices in tables that are created in the linking process. The function glGetUniformLocation returns the index of a attribute variable, such as the vertex location attribute vPosition in our vertex shader. From the perspective of the application program, the client, we have to do two things. We have to enable the vertex attributes that are in the shaders (glEnableVertexAttribArray) and we must describe the form of the data in the vertex array (glVertexAttribPointer) as in the code

```c
GLuint loc;

loc = glGetUniformLocation(program, "vPosition");
glEnableVertexAttribArray(loc);
glVertexAttribPointer(loc, 2, GL_FLOAT, GL_FALSE, 0, points);
```

In glVertexAttribPointer, the second and third parameters specify that the array pointer is a two dimensional array of floats. The fourth parameter says that we do not want the data normalized whereas the fifth states that the values in the array are contiguous.
Note that the data in points in the application consists of only $x$ and $y$ values, the array vPosition in the vertex shader is four dimensional. This difference does not create a problem since we have described the data correctly in our function parameters. The underlying reason for the differences is a fundamental aspect of how our graphics systems work. We want our application programs to be as close to the problem as possible. Some of our applications will be two dimensional; most will be three dimensional and some may even be four dimensional.

A complete listing of this program, the initShader function and a function for reading shader source code, as well as other example programs that we generate in subsequent chapters, are given in Appendix AA.

### 2.9 POLYGONS AND RECURSION

The output from our gasket program (Figure 2.2) shows considerable structure. If we were to run the program with more iterations, then much of the randomness in the image would disappear. Examining this structure, we see that regardless of how many points we generate, there are no points in the middle. If we draw line segments connecting the midpoints of the sides of the original triangle, then we divide the original triangle into four triangles, the middle one contains no points (Figure 2.39).

Looking at the other three triangles, we see that we can apply the same observation to each of them; that is, we can subdivide each of these triangles into four triangles by connecting the midpoints of the sides, and each middle triangle will contain no points.

This structure suggests a second method for generating the Sierpinski gasket—one that uses polygons instead of points and does not require the use of a random-number generator. One advantage of using polygons is that we can fill solid areas on our display. Our strategy is to start with a single triangle, to subdivide it into four smaller triangles by bisecting the sides, and then to remove the middle triangle from further consideration. We repeat this procedure on the remaining triangles until the size of the triangles that we are removing is small enough—about the size of one pixel—that we can draw the remaining triangles.

We can implement the process that we just described through a recursive program. We start its development with a simple function that adds the locations of the three vertices that specify a triangle to an array points:

```c
void triangle( GLfloat *a, GLfloat *b, GLfloat *c)
{
    static int i = 0;

    points[i][0] = a[0];
```

FIGURE 2.39 Bisecting the sides of a triangle.
points[i][1] = a[1];
i++;
points[i][0] = b[0];
points[i][1] = b[1];
i++;
points[i][0] = c[0];
points[i][1] = c[1];
i++;
}

Hence, each time that triangle is called it adds three two-dimensional vertices to the data array.

Suppose that the vertices of our original triangle are given by the following array:

GLfloat v[3][2];

Then the midpoints of the sides are given by the array m[3][3], which can be computed using the following code:

for(j=0; j<2; j++) m[0][j]=(v[0][j]+v[1][j])/2.0;
for(j=0; j<2; j++) m[1][j]=(v[0][j]+v[2][j])/2.0;
for(j=0; j<2; j++) m[2][j]=(v[1][j]+v[2][j])/2.0;

With these six locations, we can use triangle to place the data for the three triangles formed by (v[0], m[0], m[1]); (v[2], m[1], m[2]); and (v[1], m[2], m[0]) in points. However, we do not simply want to draw these triangles; we want to subdivide them. Hence, we make the process recursive. We define a recursive function

divide_triangle(float *a, float *b, float *c, int k)

that will draw the triangles only if k is zero. Otherwise, it will subdivide the triangle specified by a, b, and c and decrease k. Here is the code:

void divide_triangle(GLfloat *a, GLfloat *b, GLfloat *c, int k)
{
    GLfloat ab[2], ac[2], bc[2];
    int j;
    if(k>0)
    {
        /* compute midpoints of sides */

        for(j=0; j<2; j++) ab[j]=(a[j]+b[j])/2;
        for(j=0; j<2; j++) ac[j]=(a[j]+c[j])/2;
        for(j=0; j<2; j++) bc[j]=(b[j]+c[j])/2;
    }
Chapter 2 Graphics Programming

*/ subdivide all but inner triangle */

divide_triangle(a, ab, ac, k-1);
divide_triangle(c, ac, bc, k-1);
divide_triangle(b, bc, ab, k-1);
}

else triangle(a, b, c); /* draw triangle at
end of recursion */

The display function is now almost trivial. It uses a global value of \( n \) determined by the main program to fix the number of subdivision steps we would like, and it calls divide_triangle once with the single function call

divide_triangle(v[0], v[1], v[2], N);

We set up the buffer object exactly as we did previously and we can then render all the triangles by

void display( void )
{
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_POLYGONS, 0, M);
    glFlush();
}

The rest of the program is the same as our previous gasket program except that we read in the value of \( n \). Output for five subdivision steps is shown in Figure 2.40. The complete program is given in Appendix A.

2.10 THE THREE-DIMENSIONAL GASKET

We have argued that two-dimensional graphics is a special case of three-dimensional graphics, but we have not yet seen a complete three-dimensional program. Next, we convert our two-dimensional Sierpinski gasket program to a program that will generate a three-dimensional gasket; that is, one that is not restricted to a plane. We can follow either of the two approaches that we used for the two-dimensional gasket. Both extensions start in a similar manner, replacing the initial triangle with a tetrahedron (Figure 2.41).

---

8. Note that often we have no convenient way to pass variables to GLUT callbacks other than through global parameters. Although we prefer not to pass values in such a manner, because the form of these functions is fixed, we have no good alternative.
2.10.1 Use of Three-Dimensional Points

Because every tetrahedron is convex, the midpoint of a line segment between a vertex and any point inside a tetrahedron is also inside the tetrahedron. Hence, we can follow the same procedure as before, but this time, instead of the three vertices required to define a triangle, we need four initial vertices to define the tetrahedron. Note that as long as no three vertices are collinear, we can choose the four vertices of the tetrahedron at random without affecting the character of the result.

The required changes are primarily in the function display. We define and initialize an array to hold the vertices as follows:

```c
/* vertices of an arbitrary tetrahedron */
GLfloat vertices[4][3]={{ -1.0, -1.0, -1.0},{ 1.0, -1.0, -1.0},
{ 0.0, 1.0, -1.0},{ 0.0, 0.0, 1.0}};

/* arbitrary initial location inside tetrahedron */
GLfloat p[3] ={ 0.0, 0.0, 0.0};
```

We now use the array

```c
GLfloat points[M][3];
```

to store the vertex data. We compute a new location as before but add a midpoint computation for the z component:

```c
/* computes and plots a single new location */
```
int rand();
int i;
j=rand()%4; /* pick a vertex at random */

/* compute point halfway between vertex and old location */
p[0] = (p[0]+vertices[j][0])/2.0;
p[1] = (p[1]+vertices[j][1])/2.0;
}

We create a buffer object exactly as with the two-dimensional version and can use the same display function.

One problem with the three-dimensional gasket that we did not have with the two-dimensional gasket occurs because points are not restricted to a single plane; thus, it may be difficult to envision the three-dimensional structure from the two-dimensional image displayed, especially if render each point in the same color.

To get around this problem, we can add a more sophisticated color setting process to our shaders, one that makes the color of each point depend on that point's location. We can map the color cube to the default view volume by noting that both are cubes but that whereas $x$, $y$, and $z$ range from -1 to 1, each color component must be between 0 and 1. If we use the mapping

$$ = \frac{1 + x}{2},$$

$$g = \frac{1 + y}{2},$$

$$b = \frac{1 + z}{2},$$

every point in the viewing volume maps to a distinct color. In the vertex shader we can set the color using the components of $vPosition$ so our shader becomes

attribute vec4 vPosition;
out vec4 color;

void main()
{
    color = vec4((1.0+vPosition.x)/2.0, (1.0+vPosition.y)/2.0, (1.0+vPosition.z)/2.0,1.0);
    gl_Position = vPosition;
}
This color is output so the fragment shader can use it as input to set the color of a fragment so the fragment shader becomes

```glsl
in vec4 color;
void main()
{
    gl_FragColor = color;
}
```

Figure 2.42 and Front Plate 1 show that if we generate enough points, the resulting figure will look like the initial tetrahedron with increasingly smaller tetrahedrons removed.

### 2.10.2 Use of Polygons in Three Dimensions

There is a more interesting approach to the three dimensional Sierpinski gasket that uses both polygons and subdivision of a tetrahedron into smaller tetrahedrons. Suppose that we start with a tetrahedron and find the midpoints of its six edges and connect these midpoints as shown in Figure 2.43. There are now four smaller tetrahedrons, one for each of the original vertices, and another area in the middle that we will discard.

Following our second approach to a single triangle, we will use recursive subdivision to subdivide the four tetrahedrons that we keep. Because the faces of a tetrahedron are the four triangles determined by its four vertices, at the end of the subdivisions, we can render each of the final tetrahedrons by drawing four triangles.

![Three-dimensional Sierpinski gasket](image)
Most of our code is almost the same as in two dimensions. Our triangle routine now uses points in three dimensions rather than in two dimensions:

```c
void triangle( GLfloat *a, GLfloat *b, GLfloat *c)
/* specify one triangle */
{
    static int i = 0;
    points[i][0] = a[0];
    points[i][1] = a[1];
    points[i][2] = a[2];
    i++;
    points[i][0] = b[0];
    points[i][1] = b[1];
    points[i][2] = b[2];
    i++;
    points[i][0] = c[0];
    points[i][1] = c[1];
    points[i][2] = c[2];
    i++;
}
```

We draw each tetrahedron, coloring each face with a different color by using the following function:

```c
void tetra(GLfloat *a, GLfloat *b, GLfloat *c, GLfloat *d)
{
    triangle(a, b, c);
    triangle(a, c, d);
    triangle(a, d, b);
    triangle(b, d, c);
}
```

We subdivide a tetrahedron in a manner similar to subdividing a triangle. Our code for `divide_triangle` does the same:

```c
void divide_tetra(GLfloat *a, GLfloat *b, GLfloat *c, GLfloat *d, int m)
{
    GLfloat mid[6][3];
    int j;
    if(m>0)
    {
        /* compute six midpoints */
        for(j=0; j<3; j++) mid[0][j]=(a[j]+b[j])/2;
```
2.10 The Three-Dimensional Gasket

```c
for(j=0; j<3; j++) mid[1][j]=(a[j]+c[j])/2;
for(j=0; j<3; j++) mid[2][j]=(a[j]+d[j])/2;
for(j=0; j<3; j++) mid[3][j]=(b[j]+c[j])/2;
for(j=0; j<3; j++) mid[4][j]=(c[j]+d[j])/2;
for(j=0; j<3; j++) mid[5][j]=(b[j]+d[j])/2;

/* create 4 tetrahedrons by subdivision */
divide_tetra(a, mid[0], mid[1], mid[2], m-1);
divide_tetra(mid[0], b, mid[3], mid[5], m-1);
divide_tetra(mid[1], mid[3], c, mid[4], m-1);
divide_tetra(mid[2], mid[4], d, mid[5], m-1);
```

```
} else(tetra(a,b,c,d)); /* draw tetrahedron at end of recursion */
}
```

We can now start with four vertices and do \( n \) subdivisions as follows:

```c
{
    divide_tetra(v[0], v[1], v[2], v[3], n);
}
```

There are two more problems that we must address before we have a useful three-dimensional program. The first is how to deal with color. If we use just a single color as in our first example, we won't be able to see any of the three dimensional structure. Alternately, we could use the approach of our last example of letting the color of each fragment be determined by where the point is located in three dimensions. But we would prefer to use a small number of colors and color the face of each triangle with one of these colors. We can set this scheme by choosing some base colors in the application such as

```c
GLfloat base_colors[4][3] = {1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0};
```

and then assigning colors to each point as it is generated. We set a color index as we generate the triangles

```c
void tetra(GLfloat *a, GLfloat *b, GLfloat *c, GLfloat *d) {
    colorindex = 0;
    triangle(a,b,c);
    colorindex = 1;
    triangle(a,c,d);
    colorindex = 2;
```
and then form a color array with a color for each point

GLfloat colors[M][3];

void triangle( GLfloat *a, GLfloat *b, GLfloat *c)

/* specify one triangle */
{
    static int i = 0;
    int j;

    for(j=0; j<3; j++) colors[i][j] = base_colors[colorindex][j];
    points[i][0] = a[0];
    points[i][1] = a[1];
    points[i][2] = -a[2];
    i++;
    for(j=0; j<3; j++) colors[i][j] = base_colors[colorindex][j];
    points[i][0] = b[0];
    points[i][1] = b[1];
    points[i][2] = -b[2];
    i++;
    for(j=0; j<3; j++) colors[i][j] = base_colors[colorindex][j];
    points[i][0] = c[0];
    points[i][1] = c[1];
    points[i][2] = -c[2];
    i++;
}

We send these colors to the GPU by specifying a second vertex array and buffer object

GLint buffers[2];

glGenBuffers(2, buffers);
glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, buffers[1]);
glBufferData(GL_ARRAY_BUFFER, sizeof(colors), colors, GL_STATIC_DRAW);

If in the shader the color is named Vcolor, the second vertex array can be set up in the shader initialization
2.10 The Three-Dimensional Gasket

In the vertex shader, we use vColor to set a color to be sent the fragment shader. example

2.10.3 Hidden-Surface Removal

If you execute the code in the previous section, you might be confused by the results. The program draws triangles in the order that they are specified in the program. This order is determined by the recursion in our program and not by the geometric relationships among the triangles. Each triangle is drawn (filled) in a solid color and is drawn over those triangles that have already been rendered to the display.

Contrast this order to the way that we would see the triangles if we were to construct the three-dimensional Sierpinski gasket out of small solid tetrahedra. We would see only those faces of tetrahedra that were in front of all other faces as seen by a viewer. Figure 2.44 shows a simplified version of this hidden-surface problem. From the viewer’s position, quadrilateral A is seen clearly, but triangle B is blocked from view, and triangle C is only partially visible. Without going into the details of any specific algorithm, you should be able to convince yourself that given the position of the viewer and the triangles, we should be able to draw the triangles such that the correct image is obtained. Algorithms for ordering objects so that they are drawn correctly are called visible-surface algorithms or hidden-surface–removal algorithms, depending on how we look at the problem. We discuss such algorithms in detail in Chapters 4 and 7.

For now, we can simply use a particular hidden-surface–removal algorithm, called the z-buffer algorithm, that is supported by OpenGL. This algorithm can be turned on (enabled) and off (disabled) easily. In our main program, we must request the auxiliary storage, a z (or depth) buffer, by modifying the initialization of the display mode to the following:

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

Note that the z-buffer is one of the buffers that make up the frame buffer. We enable the algorithm by the function call

```c
glEnable(GL_DEPTH_TEST)
```

either in main.c or in an initialization function such as myinit.c. Because the algorithm stores information in the depth buffer, we must clear this buffer whenever we wish to redraw the display; thus, we modify the clear procedure in the display function:
Chapter 2 Graphics Programming

FIGURE 2.45 Three-dimensional gasket after five recursion steps.

```c
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

The display callback is as follows:

```c
void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, M);
    glFlush();
}
```

The results are shown in Figure 2.45 and Front Plate 2 for a recursion of four steps. The complete program is given in Appendix A.

2.11 ADDING INTERACTION

In this section, we develop event-driven input through a set of simple examples that use the callback mechanism that we introduced in Section 2.7. We examine various events that are recognized by the window system and, for those of interest to our application, we write callback functions that govern how the application program responds to these events.

2.11.1 Using the Pointing Device

We start by altering the `main` function in the gasket program. In the original version, we used functions in the GLUT library to put a window on the screen
and then entered the event loop by executing the function `glutMainLoop`. We entered the loop but could do nothing else because there were no callbacks other than the display callback. We could not even terminate the program, except through an external system-dependent mechanism, such as pressing control-C. Our first example will remedy this omission by using the pointing device to terminate a program. We accomplish this task by having the program execute a standard termination function called `exit` when a particular mouse button is depressed.

We discuss only those events recognized by GLUT. Standard window systems such as the X Window System or Microsoft Windows recognize many more events, which differ among systems. However, the GLUT library recognizes a small set of events that is common to most window systems and is sufficient for developing basic interactive graphics programs. Because GLUT has been implemented for the major window systems, we can use our simple applications on multiple systems by recompiling the application.

Two types of events are associated with the pointing device, which is conventionally assumed to be a mouse but could be a trackpad or a data tablet. A move event is generated when the mouse is moved with one of the buttons depressed. If the mouse is moved without a button being held down, this event is called a passive move event. After a move event, the position of the mouse is made available to the application program. A mouse event occurs when one of the mouse buttons is either depressed or released. A button being held down does not generate a mouse event until the button is released. The information returned includes the button that generated the event, the state of the button after the event (up or down), and the position of the cursor tracking the mouse in window coordinates (with the origin in the upper-left corner of the window). We register the mouse callback function, usually in the main function, by means of the GLUT function

```c
void myMouse(int button, int state, int x, int y)
```

The mouse callback must have the form

```c
void myMouse(int button, int state, int x, int y)
```

and is provided by the application programmer. Within the callback function, we define the actions that we want to take place if the specified event occurs. There may be multiple actions defined in the mouse callback function corresponding to the many possible button and state combinations. For our simple example, we want the depression of the left mouse button to terminate the program. The required callback is the single-line function

```c
void myMouse(int button, int state, int x, int y)
{
    if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
```
exit(0);
}

If any other mouse event—such as a depression of one of the other buttons—occurs, no response action will occur, because no action corresponding to these events has been defined in the callback function.

We will now develop an example that incorporates many of the aspects of CAD programs and adds some interactivity. Along the way we will introduce some additional call backs. We start by developing a simple program which will display a single triangle whose vertices are entered interactively using the pointing device. We will use the same shaders so most of the code will be similar to our previous examples.

We specify a global array to hold the three two-dimensional vertices

GLfloat points[3][2];

We can then use the mouse callback to capture the data each time the left mouse button is depressed. Consider the code

int w, h;

void mouse(int btn, int state, int x, int y)
{
    if(btn==GLUT_RIGHT_BUTTON && state==GLUT_DOWN) exit(0);
    if(btn==GLUT_LEFT && state==GLUT_DOWN)
    {
        points[count][0] = (float) x / (w/2)-1.0;
        points[count][1] = ((float) (h-y)) / (h/2) -1.0;
        count++;
    }
    if(count == 3)
    {
        glutPostRedisplay();
        count = 0;
    }
}

The right mouse button is used to end the program. The left mouse button is used to provide the vertex data for our triangle. We use the globals \( h \) and \( w \) to hold the height and width of the OpenGL window. Hence, in our main we might see the code

w = 512;
h = 512;
glutInitWindowSize(w, h);
in our main function, which would give us the same $512 \times 512$ window we used previously. The basic idea is that each time the left mouse button is depressed, we put the scaled location of the mouse into points and then move on to the next vertex. Scaling is necessary because the mouse callback returns the position of the mouse in screen coordinates measured from the the top left corner of the window. Thus, for our $w \times h$ window, the top left corner has coordinates $(0,0)$ whereas the bottom right corner has coordinates $(w-1, h-1)$. This number of increasing $y$ values from top to bottom is common in window systems and has its origins in television systems which display images top to bottom. The window we use in our application program has its origin in the center, the bottom left corner is at $(-1.0, 1.0)$ and the top right corner has coordinates $(1.0, 1.0)$. Because any primitives outside this region are clipped out, we want to scale the values returned by the mouse callback to this region and make sure to flip the $y$ values so that our triangles appear upright. The two lines

\[
\begin{align*}
\text{points}[\text{count}][0] &= (\text{float}) x / (w/2)-1.0; \\
\text{points}[\text{count}][1] &= (\text{float}) (h-y) / (h/2) -1.0;
\end{align*}
\]

carry out this transformation of coordinates.

Once we have the data for three vertices, we can draw the triangle. We cause the drawing of the triangle through the display callback. However, instead of invoking the display callback directly through an execution of display, we instead use

\texttt{glutPostRedisplay();}

What this function does is set an internal flag indicating that the display needs to be redrawn. Each time the system goes through the event loop, multiple events may occur whose callbacks require refreshing the display. Rather than each of these callbacks explicitly executing the display function, each uses \texttt{glutPostRedisplay} to set the display flag. Thus at the end of the event loop, if the flag is set, the display callback is invoked and the flag unset. This method prevents the display from being redrawn multiple times in a single pass through the event loop.\footnote{Some interactive applications may need to execute the display callback directly.} Returning to our example, we see that each successive three depressions of the left mouse button specifies a new triangle which replaces the previous triangle on the display.

Although we have a program that has some interactivity, introducing a few more callbacks will lead to a much more interesting program that can be expanded to a painting or CAD program.
2.11.2 Window Events

Most window systems allow a user to resize the window interactively, usually by using the mouse to drag a corner of the window to a new location. This event is called a **reshape** event and is an example of a **window event**. Other window events include iconifying a window and exposing a window that was covered by another window. Each can have a callback that specified what actions to take if the event occurs. Unlike most other callbacks, there is a default reshape callback that simply changes the viewport to the new window size, an action that might not be what the user desires. If the window size changes, we have to consider the three questions:

1. Do we redraw all the objects that were in the window before it was resized.
2. What do we do if the aspect ratio of the new window is different from that of the old window?
3. Do we change the sizes or attributes of new primitives if the size of the new window is different from that of the old?

There is no single answer to any of these questions. If we are displaying the image of a real-world scene, our reshape function probably should make sure that no shape distortions occur. But this choice may mean that part of the resized window is unused or that part of the scene cannot be displayed in the window. If we want to redraw the objects that were in the window before it was resized, we need a mechanism for storing and recalling them. Often we do this recall by encapsulating all drawing in a single function, such as the display callback function used in our previous examples. In interactive applications that is probably not the best choice, because we decide what we draw interactively.

The reshape event returns in its measure the height and width of the new window. We can use these values to rescale the data that we use to specify the geometry. Thus, we have the callback

```c
void reshape(GLsizei ww, GLsizei hh)
{
    glViewport(0, 0, ww, hh);
    w = ww;
    h = hh;
}
```

This function creates a new viewport that cover the whole resized window and copies the returned values of the new window width and height to the global variables $w$ and $h$ so they can be used by the mouse callback. Note that because the reshape callback generates a display callback we do not need to call `glutPostRedisplay`. The complete triangle-drawing program is given in Appendix A.
2.11.3 Keyboard Events

We can also use the keyboard as an input device. Keyboard events are generated when the mouse is in the window and one of the keys is depressed or released. The GLUT function `glutKeyboardFunc` is the callback for events generated by depressing a key, whereas `glutKeyboardUpFunc` is the callback for events generated by release of a key.

When a keyboard event occurs, the ASCII code for the key that generated the event and the location of the mouse are returned. All the key press callbacks are registered in a single callback function, such as

```c
glutKeyboardFunc(myKey);
```

For example, if we wish to use the keyboard only to exit the program, we can use the callback function

```c
void myKey(unsigned char key, int x, int y)
{
    if(key=='q' || key == 'Q') exit( );
}
```

GLUT includes a function `glutGetModifiers` that allows the user to define actions using the meta keys, such as the Control and Alt keys. These special keys can be important when we are using one- or two-button mice because we can then define the same functionality as having left, right, and middle buttons as we have assumed in this chapter. More information about these functions is in the Suggested Readings section at the end of the chapter.

2.11.4 The Idle Callback

The idle callback is invoked when there are no other events. Its default is the null function pointer. A typical use of the idle callback is to continue to generate graphical primitives through a display function while nothing else is happening. Another is to produce an animated display.

Let’s do a simple extension to our triangle program that rotates the triangle about the center of the window. Consider the two dimensional rotation in Figure ???. A point at \((x, y)\) when rotated by \(\phi\) degrees about the origin moves to a point \((x', y')\). We obtain the equations of rotation by expressing both points in polar coordinates. If the original point is at

\[
x = r \cos(\theta),
\]

\[
y = r \sin(\theta),
\]

then the rotated point is at

\[
x = r \cos(\theta + \phi) = r(\cos(\theta) \cos(\phi) - \sin(\theta) \sin(\phi)),
\]
\[ y = r \sin(\theta + \phi) = r(\cos(\theta) \sin(\phi) - \sin(\theta) \cos(\phi)), \]

or
\[ x' = x \cos(\phi) - y \sin(\phi), \]
\[ y' = x \sin(\phi) + y \cos(\phi). \]

Instead of displaying a triangle using the entered vertex positions, first we will rotate the positions by an angle that is changed by the idle callback. In the main function, we add a display callback

```c
    glutIdleFunc(idle);
```

and change the display callback to

```c
float angle = 0.0; /* global angle */

void display( void )
{
    float x, y;
    int i;
    for (i=0; i<3; i++)
    {
        x = cos(angle)*points[i][0]-sin(angle)*points[i][1];
        y = sin(angle)*points[i][0]+cos(angle)*points[i][1];
        points[i][0] = x;
        points[i][1] = y;
    }
    glClear(GL_COLOR_BUFFER_BIT); /* clear the window */
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glFlush;
}
```

A basic idle callback that increases the angle by 1/10000 of a degree each time it is executed and posts a redisplay is

```c
#define RADIANS_TO_DEGREES 3.14159/180.0

void idle(void)
{
    angle+=0.0001*RADIANS_TO_DEGREES;
    if (angle>360.0) angle-=360.0;
    glutPostRedisplay();
}
```

We can change most callback functions during program execution by simply specifying a new callback function. We can also disable a callback.
by setting its callback function to NULL. In our example, we want to stop the rotation while we are collecting data and then restart it once a new triangle is completely specified. We can modify the display callback accomplish this change:

```c
void mouse(int btn, int state, int x, int y)
{
    if(btn==GLUT_RIGHT_BUTTON && state==GLUT_DOWN) exit(0);
    if(btn==GLUT_LEFT & state==GLUT_DOWN)
    {
        glutIdleFunc(NULL);
        points[count][0] = (float) x / (w/2)-1.0;
        points[count][1] = ((float) (h-y)) / (h/2) -1.0;
        count++;
    }
    if(count == 3)
    {
        glutIdleFunc(idle);
        glutPostRedisplay();
        count = 0;
    }
}
```

### 2.11.5 Double Buffering

Although we have a complete program, depending on the speed of your computer and how much you increment the angle in the idle callback, you may see a display does not show a rotating triangle but rather a somewhat broken up display with pieces of the triangle showing. This problem can be far more severe if try to generate a display with many objects in motion.

The reason for this behavior is the decoupling of the displaying the contents of the frame buffer from the process that changes values in the frame buffer. Typically the frame buffer is redisplayed at a regular rate, known as the **refresh rate**, which is in the range of 60 to 100 Hz (or frames per second). However, an application program operates asynchronously and can cause changes to the frame buffer at any time. Hence, a redisplay of the frame buffer can occur when its contents are still being altered by the application and the viewer will see only a partially drawn display. There are a couple of solutions to this problem. Some operating systems give the user a parameter to set which will couple or sync the drawing into and display of the frame buffer.

The more common solution is **double buffering**. Instead of a single frame buffer, the hardware has two frame buffers. One, called the **front buffer**, is one that is displayed. The other, called the **back buffer**, is then available for constructing what we would like to display. Once the drawing is complete, we swap the front and back buffers. We then clear the new back buffer and
can start drawing into it. Thus, rather than using `glFlush` at the end of the display callback, we use

```c
glutSwapBuffers();
```

We have to make one other change to use double buffering. In our initialization we have to request a double buffer. Hence in `main` we use

```c
glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE);
```

Note that the default in GLUT is equivalent to using `GLUT_SINGLE` rather than `GLUT_DOUBLE`. However, modern graphics hardware has sufficient memory that we can always use double rather than single buffering.

### 2.11.6 Window Management

GLUT also supports both multiple windows and subwindows of a given window. We can open a second top-level window (with the label “second window”) by

```c
id=glutCreateWindow("second window");
```

The returned integer value allows us to select this window as the current window into which objects will be rendered by

```c
glutSetWindow(id);
```

We can make this window have properties different from those of other windows by invoking the `glutInitDisplayMode` before `glutCreateWindow`. Furthermore, each window can have its own set of callback functions because callback specifications refer to the present window.

### 2.12 MENUS

We could use our graphics primitives and our mouse callbacks to construct various graphical input devices. For example, we could construct a slidebar as shown in Figure 2.46, using filled rectangles for the device, text for any labels, and the mouse to get the position. However, much of the code would be tedious to develop, especially if we tried to create visually appealing and effective graphical devices (widgets). Most window systems provide a toolkit that contains a set of widgets, but because our philosophy is not to restrict our discussion to any particular window system, we shall not discuss the specifics of such widget sets. Fortunately, GLUT provides one additional feature, **popup menus**, that we can use with the mouse to create sophisticated interactive applications.

![FIGURE 2.46 Slidebar.](image-url)
Using menus involves taking a few simple steps. We must define the actions corresponding to each entry in the menu. We must link the menu to a particular mouse button. Finally, we must register a callback function for each menu. We can demonstrate simple menus with the example of a pop-up menu that has three entries. The first selection allows us to exit our program. The second and third start and stop the rotation. The function calls to set up the menu and to link it to the right mouse button should be placed in our main function. They are

```c
glutCreateMenu(demo_menu);
glutAddMenuEntry("quit", 1);
glutAddMenuEntry("start rotation", 2);
glutAddMenuEntry("stop rotation", 3);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

The function `glutCreateMenu` registers the callback function `demo_menu`.

The second argument in each entry’s definition is the identifier passed to the callback when the entry is selected. Hence, our callback function is

```c
void demo_menu(int id)
{
    switch(id)
    {
    case 1: exit(0);
        break;
    case 2: glutIdleFunc(idle);
        break;
    case 3: glutIdleFunc(NULL);
        break;
    }
    glutPostRedisplay( );
}
```

The call to `glutPostRedisplay` requests a redraw through the `glutDisplayFunc` callback, so that the screen is drawn again without the menu.
GLUT also supports hierarchical menus, as shown in Figure 2.47. For example, suppose that we want the main menu that we create to have only two entries. The first entry still causes the program to terminate, but now the second causes a submenu to pop up. The submenu contains the two entries for turning the rotation on and off. The following code for the menu (which is in \texttt{main}) should be clear:

```c
sub_menu = glutCreateMenu(rotation_menu);
glutAddMenuEntry("start rotation", 2);
glutAddMenuEntry("stop rotation", 3);
glutCreateMenu(top_menu);
glutAddMenuEntry("Quit",1);
glutAddSubMenu("start/stop rotation", sub_menu);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

Writing the callback functions, \texttt{rotation\_menu} and \texttt{top\_menu}, should be a simple exercise for you (Exercise 3.5).

**SUMMARY AND NOTES**

In this chapter, we introduced just enough of the OpenGL API to apply the basic concepts that we learned in Chapter 1. Although the first application we used to develop our first program was two-dimensional, we took the path of looking at two-dimensional graphics as a special case of three-dimensional graphics. We then were able to extend the example to three dimensions with minimal work.

The Sierpinski gasket provides a nontrivial beginning application. A few extensions and mathematical issues are presented in the exercises at the end of this chapter. The texts in the Suggested Readings section provide many other examples of interesting curves and surfaces that can be generated with simple programs.

The historical development of graphics APIs and graphical models illustrates the importance of starting in three dimensions. The pen-plotter model from Chapter 1 was used for many years and is the basis of many important APIs, such as PostScript. Work to define an international standard for graphics APIs began in the 1970s and culminated with the adoption of GKS by the International Standards Organization (ISO) in 1984. However, GKS had its basis in the pen-plotter model and as a two-dimensional API was of limited utility in the CAD community. Although the standard was extended to three dimensions with GKS-3D, the limitations imposed by the original underlying model led to a standard that was lacking in many aspects. The PHIGS and PHIGS+ APIs, started in the CAD community, are inherently three-dimensional and are based on the synthetic-camera model.

OpenGL is derived from the GL API, which is based on implementing the synthetic-camera model with a pipeline architecture. GL was developed for Silicon Graphics, Inc. (SGI) workstations, which incorporated a pipeline
architecture originally implemented with special-purpose VLSI chips. Hence, although PHIGS and GL have much in common, GL was designed specifically for high-speed real-time rendering. OpenGL was a result of application users realizing the advantages of GL programming and wanting to carry these advantages to other platforms. Because it removed input and windowing functions from GL and concentrated on rendering, OpenGL emerged as a new API that was portable while retaining the features that make GL such a powerful API.

Although most application programmers who use OpenGL prefer to program in C, there is a fair amount of interest in higher-level interfaces. Using C++ rather than C requires minimal code changes but does not provide a true object-oriented interface to OpenGL. Among object-oriented programmers, there has been much interest in both OpenGL and higher-level APIs. Although there is no official Java binding to OpenGL, there have been multiple efforts to come up with one. The problem is not simple, because application users want to make use of the object orientation of Java and various Java toolkits, together with a non–object-oriented OpenGL specification. There are a few bindings available on the Internet, and Sun Microsystems recently released their Java bindings. Many Java programmers use the JOGL bindings.

In Chapter 10, we will introduce scene graphs, which provide a much higher-level, object-oriented interface to graphics hardware. Most scene graph APIs are built on top of OpenGL.

Within the game community, the dominance of Windows makes it possible for game developers to write code for a single platform. DirectX runs only on Windows platforms and is optimized for speed on these systems. Although much DirectX code looks like OpenGL code, the coder can use device-dependent features that are available in commodity graphics cards. Consequently, applications written in DirectX do not have the portability and stability of OpenGL applications. Thus, we see DirectX dominating the game world, whereas scientific and engineering applications generally are written in OpenGL. For OpenGL programmers who want to use features specific to certain hardware, OpenGL has an extension mechanism for accessing these features but at the cost of portability. Programming pipelines that are accessible through the OpenGL Shading Language and Cg (Chapter 9) are leading to small performance differences between OpenGL and DirectX for high-end applications.

Our examples and programs have shown how we describe and display geometric objects in a simple manner. In terms of the modeling–rendering paradigm that we presented in Chapter 1, we have focused on the modeling. However, our models are completely unstructured. Representations of objects are lists of vertices and attributes. In Chapter 10, we will learn to construct hierarchical models that can represent relationships among objects. Nevertheless, at this point, you should be able to write interesting programs. Complete the exercises at the end of the chapter and extend a few of the two-dimensional problems to three dimensions.
SUGGESTED READINGS

The Sierpinski gasket provides a good introduction to the mysteries of fractal geometry; there are good discussions in several texts [Bar93, Hil91, Man82, Pru90].

The pen-plotter API is used by PostScript [Ado85] and LOGO [Pap81]. LOGO provides turtle graphics, an API that is both simple to learn and capable of describing several of the two-dimensional mathematical curves that we use in Chapter 11 (see Exercise 2.4).

GKS [ANSI85], GKS-3D [ISO88], PHIGS [ANSI88], and PHIGS+[PHI89] are both U.S. and international standards. Their formal descriptions can be obtained from the American National Standards Institute (ANSI) and from ISO. Numerous textbooks use these APIs [Ang90, End84, Fol94, Hea04, Hop83, Hop91].

The X Window System [Sch88] has become the standard on UNIX workstations and has influenced the development of window systems on other platforms. The RenderMan interface is described in [Ups89].

The standard reference for OpenGL is the OpenGL Programming Guide [Ope10]. The OpenGL Reference Manual [Ope04] has the man pages for older versions. There is also a formal specification of OpenGL [Seg92]. The OpenGL Shading Language is described in [Ros09]. The standards documents as well as many other references and pointers to code examples are on the OpenGL website www.opengl.org.

Starting with the second edition and continuing through the present edition, the Programming Guide uses the GLUT library that was developed by Mark Kilgard [Kil94b]. The Programming Guide provides many more code examples using OpenGL. GLUT was developed for use with the X Window System [Kil96], but there are also versions for Windows and the Macintosh. Much of this information and many of the example programs are available over the Internet. Representative sites are listed at the beginning of Appendix A.

OpenGL: A Primer [Ang08], the companion book to this text, contains details of the OpenGL functions used here and more example programs. Windows users can find more examples in [Wri07] and [Fos97]. Details for Mac OS X users are [Koe08].

The graphics part of the DirectX API was originally known as Direct3D. The present version is Version 10.0.

The marching-squares method is a special case of the marching-cubes method [Lor87] that was developed for the visualization of volumetric data.

EXERCISES

2.1 A slight variation on generating the Sierpinski gasket with triangular polygons yields the fractal mountains used in computer-generated animations. After you find the midpoint of each side of the triangle, perturb
this location before subdivision. Generate these triangles without fill. Later, you can do this exercise in three dimensions and add shading. After a few subdivisions, you should have generated sufficient detail that your triangles look like a mountain.

2.2 The Sierpinski gasket, as generated in Exercise 2.1, demonstrates many of the geometric complexities that are studied in fractal geometry [Man82]. Suppose that you construct the gasket with mathematical lines that have length but no width. In the limit, what percentage of the area of the original triangle remains after the central triangle has been removed after each subdivision? Consider the perimeters of the triangles remaining after each central triangle is removed. In the limit, what happens to the total perimeter length of all remaining triangles?

2.3 At the lowest level of processing, we manipulate bits in the frame buffer. OpenGL has pixel-oriented commands that allow users to access the frame buffer directly. You can experiment with simple raster algorithms, such as drawing lines or circles, by using the OpenGL function glPoint as the basis of a simple virtual-frame-buffer library. Write a library that will allow you to work in a frame buffer that you create in memory. The core functions should be WritePixel and ReadPixel. Your library should allow you to set up and display your frame buffer and to run a user program that reads and writes pixels.

2.4 Turtle graphics is an alternative positioning system that is based on the concept of a turtle moving around the screen with a pen attached to the bottom of its shell. The turtle’s position can be described by a triplet \((x, y, \theta)\), giving the location of the center and the orientation of the turtle. A typical API for such a system includes functions such as the following:

\[
\text{init}(x, y, \theta); /* initialize position and orientation of turtle */
\]

\[
\text{forward} (\text{distance});
\]

\[
\text{right} (\text{angle});
\]

\[
\text{left} (\text{angle});
\]

\[
\text{pen} (\text{up\_down});
\]

Implement a turtle-graphics library using OpenGL.

2.5 Use your turtle-graphics library from Exercise 2.4 to generate the Sierpinski gasket and fractal mountains of Exercises 2.1 and 2.2.

2.6 Space-filling curves have interested mathematicians for centuries. In the limit, these curves have infinite length, but they are confined to a finite rectangle and never cross themselves. Many of these curves can be generated iteratively. Consider the “rule” pictured in Figure 2.48 that replaces a single line segment with four shorter segments. Write a program that starts with a triangle and iteratively applies the replacement
FIGURE 2.48 Generation of the Koch snowflake.

FIGURE 2.49 Maze.

rule to all the line segments. The object that you generate is called the Koch snowflake. For other examples of space-filling curves, see [Hil07] and [Bar93].

2.7 You can generate a simple maze starting with a rectangular array of cells. Each cell has four sides. You remove sides (except from the perimeter of all the cells) until all the cells are connected. Then you create an entrance and an exit by removing two sides from the perimeter. A simple example is shown in Figure 2.49. Write a program using OpenGL that takes as input the two integers \( N \) and \( M \) and then draws an \( N \times M \) maze.

2.8 Describe how you would adapt the RGB-color model in OpenGL to allow you to work with a subtractive color model.

2.9 We saw that a fundamental operation in graphics systems is to map a point \((x, y)\) that lies within a clipping rectangle to a point \((x_s, y_s)\) that lies in the viewport of a window on the screen. Assume that the two rectangles are defined by the viewport specified by:

\[
\text{glViewport}(u, v, w, h);
\]

and a viewing rectangle specified by

\[
x_{\text{min}} \leq x \leq x_{\text{max}},
\]

\[
y_{\text{min}} \leq y \leq y_{\text{max}}.
\]

Find the mathematical equations that map \((x, y)\) into \((x_s, y_s)\).

2.10 Many graphics APIs use relative positioning. In such a system, the API contains functions such as

\[
\text{move\_rel}(x,y);
\]

\[
\text{line\_rel}(x,y);
\]

for drawing lines and polygons. The \text{move\_rel} function moves an internal position, or cursor, to a new position; the \text{line\_rel} function moves the cursor and defines a line segment between the old cursor position and the new position. What are the advantages and disadvantages of relative positioning as compared to the absolute positioning
used in OpenGL? Describe how you would add relative positioning to OpenGL.

2.11 In practice, testing each point in a polygon to determine whether it is inside or outside the polygon is extremely inefficient. Describe the general strategies that you might pursue to avoid point-by-point testing.

2.12 Devise a test to determine whether a two-dimensional polygon is simple.

2.13 Figure 2.50 shows a set of polygons called a mesh; these polygons share some edges and vertices. Find one or more simple data structures that represent the mesh. A good data structure should include information on shared vertices and edges. Using OpenGL, find an efficient method for displaying a mesh represented by your data structure. *Hint:* Start with an array or linked list that contains the locations of the vertices.

2.14 In Section 2.4, we saw that OpenGL defines polygons using lists of vertices. Why might it be better to define polygons by their edges? *Hint:* Consider how you might represent a mesh efficiently.

2.15 In OpenGL, we can associate a color with each vertex. If the endpoints of a line segment have different colors assigned to them, OpenGL will interpolate between the colors as it renders the line segment. It will do the same for polygons. Use this property to display the Maxwell triangle: an equilateral triangle whose vertices are red, green, and blue. What is the relationship between the Maxwell triangle and the color cube?

2.16 We can simulate many realistic effects using computer graphics by incorporating simple physics in the model. Simulate a bouncing ball in two dimensions incorporating both gravity and elastic collisions with a surface. You can model the ball with a closed polygon that has a sufficient number of sides to look smooth.

2.17 An interesting but difficult extension of Exercise 2.16 is to simulate a game of pool or billiards. You will need to have multiple balls that can interact with the sides of the table and with one another. *Hint:* Start with two balls and consider how to detect possible collisions.
2.18 A certain graphics system with a CRT display is advertised to display any four out of 64 colors. What does this statement tell you about the frame buffer and about the quality of the monitor?

2.19 Devise a test for the convexity of a two-dimensional polygon.

2.20 One problem for beginning users of OpenGL is the number of forms of the basic functions such as `glVertex`. In a language such as C++, we can use a single name and let the compiler pick the correct version by the type of the arguments. In C++, write a library to sit between OpenGL and a user program that minimizes the number of functions that the application programmer needs.

2.21 Another approach to the three-dimensional gasket is based on subdividing only the faces of an initial tetrahedron. Write a program that takes this approach. How do the results differ from the program that we developed in Section 2.10?

2.22 Each time that we subdivide the tetrahedron and keep only the four smaller tetrahedrons corresponding to the original vertices, we decrease the volume by a factor $f$. Find $f$. What is the ratio of the new surface area of the four tetrahedrons to the surface area of the original tetrahedron?

2.23 If we extend the marching-squares argument to surfaces in three dimensions, we get a method called `marching cubes`. We look at the possible ways that a surface can intersect a cube and color the cube’s vertices accordingly. How many black and white colorings of the cube are there? How many unique colorings remain when we remove symmetries?