2.1.2 Windows-based Programming

As mentioned earlier, many modern graphics systems are *windows based* and manage the display of multiple overlapping *windows*. The user can move windows around the screen by means of the mouse and can resize the windows. Using OpenGL, we will do our drawing in one of these windows, as we saw in Figure 2.1(c).

Event-driven Programming

Another property of most windows-based programs is that they are *event driven*. This means that the program responds to various events, such as the click of a mouse, the press of a key on a keyboard, or the resizing of a window on a screen. The system automatically manages an *event queue*, which receives messages stating that certain events have occurred and deals with them on a first-come, first-served, basis. The programmer organizes a program as a collection of *callback functions* that are executed when events occur. A callback function is created for each type of event that might take place. When the system removes an event from the queue, it simply executes the callback function associated with the type of that event. For programmers used to building programs with a "do this, then do this..." structure, some rethinking is required. The new structure is more like "do nothing until an event occurs, and then do the specified thing."

The method of associating a callback function with a particular type of event is often quite system dependent. But OpenGL comes with a *Utility Toolkit* (see Appendix 1), which provides tools to assist with event management. For instance,

```
glutMouseFunc(myMouse);     // register the mouse action function
```

registers the function `myMouse()` as the function to be executed when a mouse event occurs. The prefix "glut" indicates that this function is part of the Open*GL* *U*tility *T*oolkit. The programmer puts code in `myMouse()` to handle all of the possible mouse actions that are of interest.

Figure 2.2 shows a skeleton of an example `main()` function for an event-driven program. We will base most of our programs in this book on this skeleton. There are four principal types of events we will work with, and a "glut" function is available for each:

- `glutDisplayFunc(myDisplay);` Whenever the system determines that a window should be redrawn on the screen, it issues a "redraw" event. This happens when the window is first opened and when the window is exposed by moving another window off of it. Here, the function `myDisplay()` is registered as the callback function for a redraw event.
- `glutReshapeFunc(myReshape);` Screen windows can be reshaped by the user, usually by dragging a corner of the window to a new position with the mouse.

FIGURE 2.2 A skeleton of an event-driven program using OpenGL.

```
void main()
{
    initialize things⁵
    create a screen window
    glutDisplayFunc(myDisplay);   // register the redraw function
    glutReshapeFunc(myReshape);   // register the reshape function
    glutMouseFunc(myMouse);       // register the mouse action function
    glutKeyboardFunc(myKeyboard); // register the keyboard action function
    perhaps initialize other things
    glutMainLoop();               // enter the unending main loop
}
    all of the callback functions are defined here
```

[5] Notes shown in italics in fragments of code are pseudocode rather than actual program code. They suggest the actions that real code substituted there should accomplish.

(Simply moving the window does not produce a reshape event.) Here, the function `myReshape()` is registered with the reshape event. As we shall see, `myReshape()` is automatically passed arguments that report the new width and height of the reshaped window.

- `glutMouseFunc(myMouse)`; When one of the mouse buttons is pressed or released, a mouse event is issued. Here, `myMouse()` is registered as the function to be called when a mouse event occurs. The function `myMouse()` is automatically passed arguments that describe the location of the mouse and the nature of the action initiated by pressing the button.
- `glutKeyboardFunc(myKeyboard)`; This command registers the function `myKeyboard()` with the event of pressing or releasing some key on the keyboard. The function `myKeyboard()` is automatically passed arguments that tell which key was pressed. Conveniently, it is also passed data indicating the location of the mouse at the time the key was pressed.

If a particular program does not make use of a mouse, the corresponding callback function need not be registered or written. Then mouse clicks have no effect in the program. The same is true for programs that do not use a keyboard.

The final function shown in Figure 2.2 is `glutMainLoop()`. When this instruction is executed, the program draws the initial picture and enters an unending loop, in which it simply waits for events to occur. (A program is normally terminated by clicking in the "go away" box that is attached to each window.)

2.1.3 Opening a Window for Drawing

The first task in making pictures is to open a screen window for drawing. This effort can be quite involved and is system dependent. Because OpenGL functions are device independent, they provide no support for controlling windows on specific systems. But the OpenGL Utility Toolkit *does* include functions to open a window on whatever system you are using.

Figure 2.3 fleshes out the skeleton of Figure 2.2 above to show the entire `main()` function for a program that will draw graphics in a screen window. The first five function calls use the OpenGL Utility Toolkit to open a window for drawing. In your first graphics programs, you can just copy these functions as is; later, we will see what the various arguments mean and how to substitute others for them to achieve certain

FIGURE 2.3 Code using the OpenGL Utility Toolkit to open the initial window for drawing.

```
// appropriate #includes go here - see Appendix 1

void main(int argc, char** argv)
{
      glutInit(&argc, argv); // initialize the toolkit
      glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB); // set the display mode
      glutInitWindowSize(640,480); // set window size
      glutInitWindowPosition(100, 150); // set the window position on screen
      glutCreateWindow("my first attempt"); // open the screen window

      // register the callback functions
      glutDisplayFunc(myDisplay);
      glutReshapeFunc(myReshape);
      glutMouseFunc(myMouse);
      glutKeyboardFunc(myKeyboard);

      myInit();                 // additional initializations as necessary
      glutMainLoop();           // go into a perpetual loop

}
```

effects. The first five functions initialize and display the screen window in which our program will produce graphics. We give a brief description of what each one does:

- `glutInit(&argc, argv);` This function initializes the OpenGL Utility Toolkit. Its arguments are the standard ones for passing information about the command line; we will make no use of them here.
- `glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);` This function specifies how the display should be initialized. The built-in constants `GLUT_SINGLE` and `GLUT_RGB`, which are `OR`ed together, indicate that a single display buffer should be allocated and that colors are specified using desired amounts of red, green, and blue. (Later we will alter these arguments: For example, we will use double buffering for smooth animation.)
- `glutInitWindowSize(640,480);` This function specifies that the screen window should initially be 640 pixels wide by 480 pixels high. When the program is running, the user can resize the window as desired.
- `glutInitWindowPosition(100, 150);` This function specifies that the window's upper left corner should be positioned on the screen 100 pixels over from the left edge and 150 pixels down from the top. When the program is running, the user can move this window wherever desired.
- `glutCreateWindow("my first attempt");` This function actually opens and displays the screen window, putting the title "my first attempt" in the title bar.

The remaining functions in `main()` register the callback functions as described earlier, perform any initializations specific to the program at hand, and start the main event loop processing. The programmer (you) must implement each of the callback functions as well as `myInit()`.

2.2 DRAWING BASIC GRAPHICS PRIMITIVES

We want to develop programming techniques for drawing a large number of geometric shapes that make up interesting pictures. The drawing commands will be placed in the callback function associated with a redraw event, such as the `myDisplay()` function.

We first must establish the coordinate system in which we will describe graphical objects and prescribe where they will appear in the screen window. Computer graphics programming, it seems, involves an ongoing struggle with defining and managing different coordinate systems. So we start simply and work up to more complex approaches.

We begin with an intuitive coordinate system that is tied directly to the coordinate system of the screen window [see Figure 2.1(c)], and that measures distances in pixels. Our first sample screen window, shown in Figure 2.4, is 640 pixels wide by 480

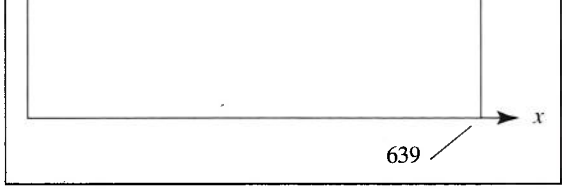

FIGURE 2.4 The initial coordinate system for drawing.

pixels high. The x-coordinate increases from 0 at the left edge to 639 at the right edge. The y-coordinate increases from 0 at the bottom edge to 479 at the top edge. We establish this coordinate system later, after examining some basic primitives.

OpenGL provides tools for drawing all of the output primitives described in Chapter 1. Most of them, such as points, lines, polylines, and polygons, are defined by one or more *vertices*. To draw such objects in OpenGL, you pass it a list of vertices. The list occurs between the two OpenGL function calls `glBegin()` and `glEnd()`. The argument of `glBegin()` determines which object is drawn. For instance, Figure 2.5 shows three points drawn in a window 640 pixels wide and 480 pixels high. These dots are drawn using the command sequence:

```
glBegin(GL_POINTS);
  glVertex2i(100, 50);
  glVertex2i(100, 130);
  glVertex2i(150, 130);
glEnd();
```

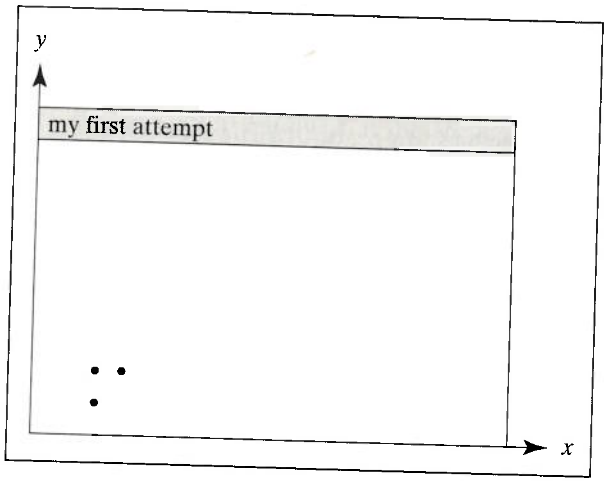

FIGURE 2.5 Drawing three dots.

The constant `GL_POINTS` is built-into OpenGL. To draw other primitives, you replace `GL_POINTS` with `GL_LINES`, `GL_POLYGON`, etc. Each of these will be introduced in turn.

As we shall see later, these commands send the information about a vertex down a "graphics pipeline," in which it goes through several processing steps. For present purposes, just think of the information as being sent more or less directly to the coordinate system in the screen window.

Many functions in OpenGL, such as `glVertex2i()`, have several variations, that distinguish the number and type of arguments passed to the function. Figure 2.6 shows how such function calls are formatted.

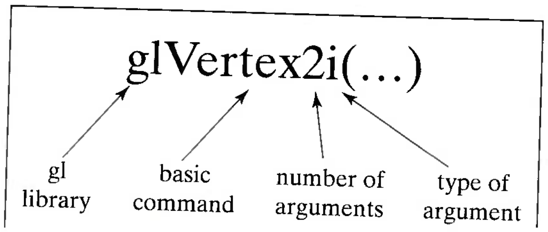

FIGURE 2.6 Format of OpenGL commands.

The prefix "`gl`" indicates a function from the OpenGL library (as opposed to "`glut`" for the GL Utility Toolkit). Then comes the basic command root, followed by the number of arguments being sent to the function (often, 3 and 4), and, finally, the type of argument (`i` for an integer, `f` or `d` for a floating-point value, etc., as we describe shortly). When we wish to refer to the basic command without regard to the specifics of its arguments, we will use an asterisk, as in `glVertex*()`.

To generate the same three-dot picture as in Figure 2.5, you could, for example, pass the function floating-point values instead of integers, using the following commands:

```
glBegin(GL_POINTS);
  glVertex2d(100.0, 50.0);
  glVertex2d(100.0, 130.0);
  glVertex2d(150.0, 130.0);
glEnd();
```

OpenGL Data Types

OpenGL works internally with specific data types. For instance, functions such as `glVertex2i()` expect integers of a certain size (32 bits). It is well known that some systems treat the C or C++ data type `int` as a 16-bit quantity, whereas others treat it as a 32-bit quantity. There is no standard size for `float` or `double` either. To ensure that OpenGL functions receive the proper data types, it is wise to use the built-in names, such as `GLint` or `GLfloat`, for OpenGL types. The OpenGL types are listed in Figure 2.7. Some of these types will not be encountered until later in the book.

FIGURE 2.7 Command suffixes and argument data types.

Suffix	Data type	Typical C or C++ type	OpenGL type name
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int or long	GLint, GLsizei
f	32-bit floating point	float	GLfloat, GLclampf
d	64-bit floating point	double	GLdouble, GLclampd
ub	8-bit unsigned number	unsigned char	GLubyte, GLboolean
us	16-bit unsigned number	unsigned short	GLushort
ui	32-bit unsigned number	unsigned int or unsigned long	GLuint, GLenum, GLbitfield

As an example, a function using the suffix `i` "expects" a 32-bit integer, but your system might translate `int` as a 16-bit integer. Therefore, if you wished to encapsulate the OpenGL commands for drawing a dot in a generic function such as `drawDot()`, you might be tempted to use the following code:

```
void drawDot(int x, int y)              ← danger: passes ints
{       // draw dot at integer point (x, y)
   glBegin(GL_POINTS);
     glVertex2i(x, y);
   glEnd();
}
```

This code passes `ints` to `glVertex2i()`. This will work on systems that use 32-bit `ints`, but might cause trouble on those that use 16-bit `ints`. It is safer to write `drawDot()` as in Figure 2.8 and to use `GLints` in your programs. When you recompile your programs on a new system, `GLint GLfloat`, etc. will be associated with the appropriate C++ types (in the OpenGL header `GL.h`; see Appendix 1) for that system, and these types will be used consistently throughout the program.

```
void drawDot(GLint x, GLint y)
{     // draw dot at integer point (x, y)
  glBegin(GL_POINTS);
    glVertex2i(x, y);
  glEnd();
}
```

FIGURE 2.8 Encapsulating OpenGL details in the generic function `drawDot()`.[6]

The OpenGL "State"

OpenGL keeps track of many *state variables*, such as the current size of a point, the current color of a drawing, the current background color, etc. The value of a state variable remains active until a new value is given. The size of a point can be set with `glPointSize()`, which takes one floating-point argument. If the argument is 3.0, the point is usually drawn as a square, three pixels on a side. (For additional details on this and other OpenGL functions, consult the appropriate OpenGL documentation, some of which is on-line; see Appendix 1.) The color of a drawing can be specified using

```
glColor3f(red, green, blue);
```

where the values of red, green, and blue vary between 0.0 and 1.0. For example, some of the colors listed in Figure 1.31 could be set using the following string of commands:

```
glColor3f(1.0, 0.0, 0.0);    // set drawing color to red
glColor3f(0.0, 0.0, 0.0);    // set drawing color to black
glColor3f(1.0, 1.0, 1.0);    // set drawing color to white
glColor3f(1.0, 1.0, 0.0);    // set drawing color to yellow
```

The background color is set with `glClearColor(red, green, blue, alpha)`, where `alpha` specifies a degree of transparency and is discussed later. (Use 0.0 for now.) To clear the entire window to the background color, use `glClear(GL_COLOR_BUFFER_BIT)`. The argument `GL_COLOR_BUFFER_BIT` is another constant built into OpenGL.

Establishing the Coordinate System

Our method for establishing an initial choice of coordinate system will seem obscure here, but will become clearer in the next chapter, when we discuss windows, viewports, and clipping. Here, we just take the few required commands on faith. The `myInit()` function in Figure 2.9 is a good place to set up the coordinate system. As we shall see later, OpenGL routinely performs a large number of transformations. It uses matrices to do this, and the commands in `myInit()` manipulate certain matrices to accomplish the desired goal. The `gluOrtho2D()` routine sets the transformation we need for a screen window that is 640 pixels wide by 480 pixels high.

[6] Using this function instead of the specific OpenGL commands makes a program more readable. It is not unusual to build up a personal collection of such utilities.

FIGURE 2.9 Establishing a simple coordinate system.

```
void myInit(void)
{
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(0, 640.0, 0, 480.0);
}
```

Putting It All Together: A Complete OpenGL program

Figure 2.10 shows a complete program that draws the lowly three dots of Figure 2.5. It is easily extended to draw more interesting objects, as we shall see. The initialization in `myInit()` sets up the coordinate system, the point size, the background color, and the drawing color. The drawing is encapsulated in the callback function `myDisplay()`. Because this program is not interactive, no other callback functions are used. `glFlush()` is called after the dots are drawn, to ensure that all data are completely processed and sent to the display. This is important in some systems that

FIGURE 2.10 A complete OpenGL program to draw three dots.

```
#include<windows.h>   // use as needed for your system
#include<gl/Gl.h>
#include<gl/glut.h>
//<<<<<<<<<<<<<<<<<<<<<<< myInit >>>>>>>>>>>>>>>>>>>>
 void myInit(void)
 {
    glClearColor(1.0,1.0,1.0,0.0);       // set white background color
    glColor3f(0.0f, 0.0f, 0.0f);          // set the drawing color
    glPointSize(4.0);                 // a 'dot' is 4 by 4 pixels
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, 640.0, 0.0, 480.0);
}
//<<<<<<<<<<<<<<<<<<<<<<<< myDisplay >>>>>>>>>>>>>>>>>
void myDisplay(void)
{
    glClear(GL_COLOR_BUFFER_BIT);      // clear the screen
    glBegin(GL_POINTS);
        glVertex2i(100, 50);          // draw three points
        glVertex2i(100, 130);
        glVertex2i(150, 130);
    glEnd();
    glFlush();                    // send all output to display
}
//<<<<<<<<<<<<<<<<<<<<<<<< main >>>>>>>>>>>>>>>>>>>>>>
void main(int argc, char** argv)
{
    glutInit(&argc, argv);          // initialize the toolkit
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB); // set display mode
    glutInitWindowSize(640,480);     // set window size
    glutInitWindowPosition(100, 150); // set window position on screen
    glutCreateWindow("my first attempt"); // open the screen window
    glutDisplayFunc(myDisplay);     // register redraw function
    myInit();
    glutMainLoop();                 // go into a perpetual loop
}
```

operate over a network: Data are buffered on the host machine and sent to the remote display only when the buffer becomes full or a `glFlush()` is executed.

2.2.1 Drawing Dot Constellations

A dot constellation is some pattern of dots or points. We describe several examples of interesting dot constellations that are easily produced using the basic program in Figure 2.10. In each case, the appropriate function is named in `glutDisplayFunc()` as the callback function for the redraw event. You are strongly encouraged to implement and test each example, in order to build up your experience with graphics.

EXAMPLE 2.2.1 The Big Dipper

Figure 2.11 shows a pattern of eight dots representing the Big Dipper, a familiar sight in the night sky.

The names and positions of the eight stars in the Big Dipper (in one particular view of the night sky), are given by the following ordered triplets: {Dubhe, 289, 190}, {Merak, 320, 128}, {Phecda, 239, 67}, {Megrez, 194, 101}, {Alioth, 129, 83}, {Mizar, 75, 73}, {Alcor, 74, 74}, {Alkaid, 20, 10}. Since so few data points are involved, it is easy to list them explicitly, or "**hardwire**" them into the code. (When many dots are to be drawn, it is more convenient to store them in a file and then have the program read them from the file and draw them. We do this in a later chapter.) These points can replace the three points specified in Figure 2.10. It is useful to experiment with this constellation, trying different point sizes, as well as different background and drawing colors.

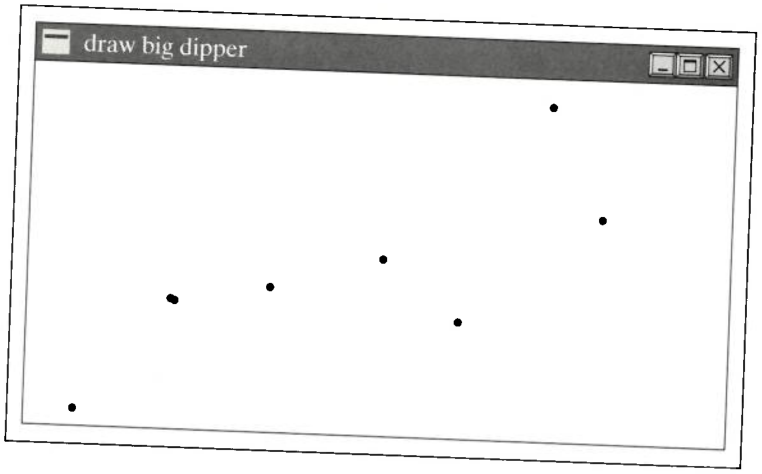

FIGURE 2.11 Two simple dot constellations.

EXAMPLE 2.2.2 Drawing the Sierpinski gasket

Figure 2.12 shows the Sierpinski gasket. Its dot constellation is generated *procedurally*, which means that each successive dot is determined by a procedural rule. Although the rule here is very simple, the final pattern is a fractal (See Chapter 9.) We first approach the rule for generating the Sierpinski gasket in an intuitive fashion. In Case Study 2.2, we see that it is one example of an *iterated function system*.

The Sierpinski gasket is produced by calling `drawDot()` many times with dot positions $(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots$, determined by a simple algorithm. Denote the kth point $p_k = (x_k, y_k)$. Each point is based on the previous point p_{k-1}. The procedure is as follows:

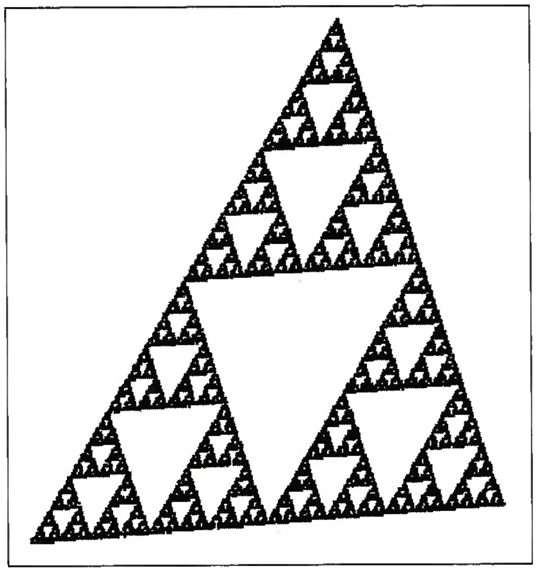
FIGURE 2.12 The Sierpinski Gasket.

1. Choose three fixed points T_0, T_1, and T_2 to form some triangle, as shown in Figure 2.13(a).

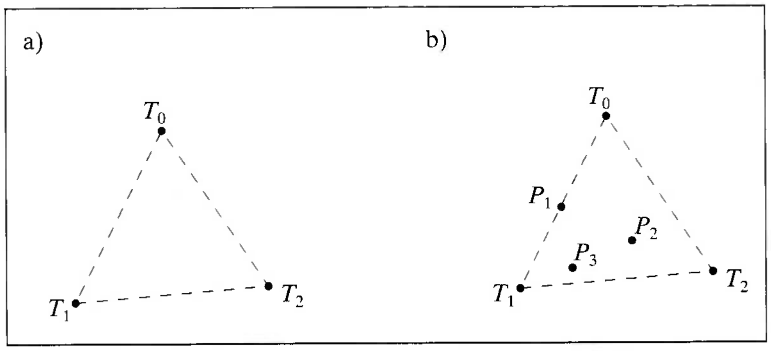

FIGURE 2.13 Building the Sierpinski gasket.

2. Choose the initial point p_0 to be drawn by selecting one of the points T_0, T_1, and T_2 at random.

Now iterate the following steps until the pattern is satisfyingly filled in:

3. Choose one of the three points T_0, T_1, and T_2 at random; call it T.
4. Construct the next point p_k as the **midpoint**[7] between T and the previously found point p_{k-1}. That is,

   p_k = midpoint of p_{k-1} and T.
5. Draw p_k using `drawDot()`.

Figure 2.13(b) shows a few iterations of the foregoing procedure. Suppose the initial point p_0 happens to be T_0, and let T_1 be chosen next. Then p_1 is formed so that it lies halfway between T_0 and T_1. Suppose T_2 is chosen next, so that p_2 lies halfway between p_1 and T_2. Next, suppose T_1 is chosen again, so that p_3 is formed as shown, etc. This process goes on generating and drawing points (conceptually, forever), and the pattern of the Sierpinski gasket quickly emerges.

[7] To find the midpoint between two points, say, (3, 12) and (5, 37), simply *average* their x and y components individually by adding them and dividing by two. So the midpoint of (3, 12) and (5, 37) is $((3 + 5)/2, (12 + 37)/2) = (4, 24)$.

It is convenient to define a simple class, `GLintPoint`, that describes a point whose coordinates are integers:[8]

```
class GLintPoint{
public:
   GLint x, y;
};
```

We then build and initialize an array of three such points, `T[0]`, `T[1]`, and `T[2]`, to hold the three corners of the triangle using `GLintPoint T[3]= {{10,10},{300,30},{200, 300}}`. There is no need to store each point p_k in the sequence as it is generated, since we simply want to draw it and then move on. So we set up a variable `point` to hold this changing point. At each iteration, `point` is updated to hold the new value.

We use `i=random(3)` to choose one of the points `T[i]` at random. `random(3)` returns one of the values 0, 1, and 2 with equal likelihood. It is defined as[9]

```
int random(int m)
{
  return rand()%m;
}
```

Figure 2.14 shows the remaining details of the algorithm, which generates 1,000 points of the Sierpinski gasket.

```
void Sierpinski(void)
{
  GLintPoint T[3]= {{10,10},{300,30},{200, 300}};

  int index = random(3);          // 0, 1, or 2 equally likely
  GLintPoint point = T[index];    // initial point
  drawDot(point.x, point.y);      // draw initial point
  for(int i = 0; i < 1000; i++)   // draw 1000 dots
  {
    index = random(3);
    point.x = (point.x + T[index].x) / 2;
    point.y = (point.y + T[index].y) / 2;
    drawDot(point.x,point.y);
  }
  glFlush();
}
```

FIGURE 2.14 Generating the Sierpinski gasket.

EXAMPLE 2.2.3 Simple "Dot Plots"

Suppose you wish to learn the behavior of some mathematical function $f(x)$ as x varies. For example, how does

$$f(x) = e^{-x}\cos(2\pi x)$$

vary for values of x between 0 and 4? A quick plot of $f(x)$ versus x, such as that shown in Figure 2.15, can reveal a lot.

[8] If C rather than C++ is being used, a simple `typedef struct{GLint x, y;}GLintPoint;` is useful here.

[9] Recall that the standard function rand() returns a pseudorandom value in the range 0 to 32767. The modulo function reduces it to a value in the range 0 to 2.

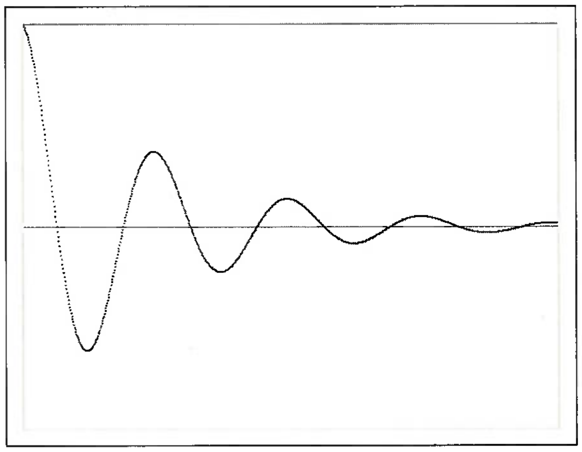
FIGURE 2.15 A "dot plot" of $e^{-x}\cos(2\pi x)$ versus x.

To plot this function, simply "sample" it at a collection of equispaced x-values, and plot a dot at each coordinate pair $(x_i, f(x_i))$. Choosing some suitable increment, say, 0.005, between consecutive x-values, we see that the process runs basically as follows:

```
glBegin(GL_POINTS);
  for(GLdouble x = 0; x < 4.0 ; x += 0.005)
       glVertex2d(x, f(x));
glEnd();
glFlush();
```

But there is a problem here: The picture that it produced will be impossibly tiny because values of x between 0 and 4 map to the first four pixels at the bottom left of the screen window. Further, the negative values of $f(.)$ will lie below the window and will not be seen at all. We therefore need to scale and position the values to be plotted so that they cover the screen window area appropriately. Here we do it by brute force, in essence picking some values to make the picture show up adequately on the screen. Later we develop a general procedure that copes with these adjustments, the so-called procedure of mapping from world coordinates to window coordinates.

Scaling x Suppose we want the range from 0 to 4 to be scaled so that it covers the entire width of the screen window, given in pixels by `screenWidth`. Then we need only scale all x-values by `screenWidth`/4, using

```
sx = x * screenWidth /4.0;
```

which yields 0 when x is 0 and `screenWidth` when x is 4.0, as desired.

Scaling and shifting y The values of $f(x)$ lie between −1.0 and 1.0, so we must scale and shift them as well. Suppose we set the screen window to have a height of `screenHeight` pixels. Then, to place the plot in the center of the window, we scale by `screenHeight`/2 and shift up by `screenHeight`/2:

```
sy = (y + 1.0) * screenHeight / 2.0;
```

As desired, this yields 0 when y is −1.0, and `screenHeight` when y is 1.0.

Note that the conversions from x to sx and from y to sy are of the form

$$sx = Ax + B$$

and

$$sy = Cy + D \tag{2.1}$$

for properly chosen values of the constants A, B, C, and D. A and C perform scaling; B and D perform shifting. The operation of scaling and shifting is basically a form of "affine transformation." We study affine transformations in depth in Chapter 5; they provide a more consistent approach that maps any specified range in x and y to the screen window.

We need only set the values of A, B, C, and D appropriately, and draw the dot plot using the following code:

```
GLdouble A, B, C, D, x;
A = screenWidth / 4.0;
B = 0.0;
C = screenHeight / 2.0;
D = C;
glBegin(GL_POINTS);
  for(x = 0; x < 4.0 ; x += 0.005)
    glVertex2d(A * x + B, C * f(x) + D);
glEnd();
glFlush();
```

Figure 2.16 shows the entire program for drawing the dot plot, to illustrate how the various ingredients fit together. The initializations are similar to those for the program that draws three dots in Figure 2.10. Notice that the width and height of the screen window are defined as constants and used where needed in the code.

PRACTICE EXERCISES

2.2.1 Dot plots for any function *f( )*

Consider drawing a dot plot of the function $f(.)$ like the one in Example 2.2.3, where it is known that as x varies from x_{low} to x_{high}, $f(x)$ takes on values between y_{low} and y_{high}. Find the appropriate scaling and translation factors of Equation 2.1 so that the dots will lie properly in a screen window with width W pixels and height H pixels. ■

2.3 MAKING LINE DRAWINGS

Hamlet: Do you see yonder cloud that's almost in shape of a camel? Polonius: By the mass, and 'tis like a camel, indeed. Hamlet: Methinks it is like a weasel. - William Shakespeare, Hamlet

As discussed in Chapter 1, line drawings are fundamental in computer graphics, and almost every graphics system comes with "driver" routines to draw straight lines. OpenGL makes it easy to draw a line: Use `GL_LINES` as the argument to `glBegin()`, and pass it the two endpoints as vertices. Thus, to draw a line between `(40, 100)` and `(202,96)`, use the following code:

```
glBegin(GL_LINES);      // use constant GL_LINES here
    glVertex2i(40, 100);
    glVertex2i(202, 96);
glEnd();
```