CHAP 2. Getting Started: Drawing Figures

Machines exist; let us then exploit them to create beauty , a modern beauty, while we are about it. For we live in the twentieth century.
Aldous Huxley

Goals of the Chapter
- To get started writing programs that produce pictures.
- To learn the basic ingredients found in every OpenGL program
- To develop some elementary graphics tools for drawing lines, polylines, and polygons.
- To develop tools that allow the user to control a program with the mouse and keyboard.

Preview
Section 2.1 discusses the basics of writing a program that makes simple drawings. The importance of device-independent programming is discussed, and the characteristics of windows-based and event-driven programs are described. Section 2.2 introduces the use of OpenGL as the device-independent application programmer interface (API) that is emphasized throughout the book and shows how to draw various graphics primitives. Sample drawings, such as a picture of the Big Dipper, a drawing of the Sierpinski gasket, and a plot of a mathematical function illustrate the use of OpenGL. Section 2.3 discusses how to make pictures based on polylines and polygons, and begins the building of a personal library of graphics utilities. Section 2.4 describes interactive graphics programming, whereby the user can indicate positions on the screen with the mouse or press keys on the keyboard to control the action of a program. The chapter ends with a number of case studies which embellish ideas discussed earlier and that delve deeper into the main ideas of the chapter.

2.1. Getting started making pictures.
Like many disciplines, computer graphics is mastered most quickly by doing it: by writing and testing programs that produce a variety of pictures. It is best to start with simple tasks. Once these are mastered you can try variations, see what happens, and move towards drawing more complex scenes.

To get started you need an environment that lets you write and execute programs. For graphics this environment must also include hardware to display pictures (usually a CRT display which we shall call the "screen"), and a library of software tools that your programs can use to perform the actual drawing of graphics primitives.

Every graphics program begins with some initializations; these establish the desired display mode, and set up a coordinate system for specifying points, lines, etc. Figure 2.1 shows some of the different variations one might encounter. In part a) the entire screen is used for drawing: the display is initialized by switching it into "graphics mode", and the coordinate system is established as shown. Coordinates *x* and *y* are measured in pixels, with x increasing to the right and y increasing downward.

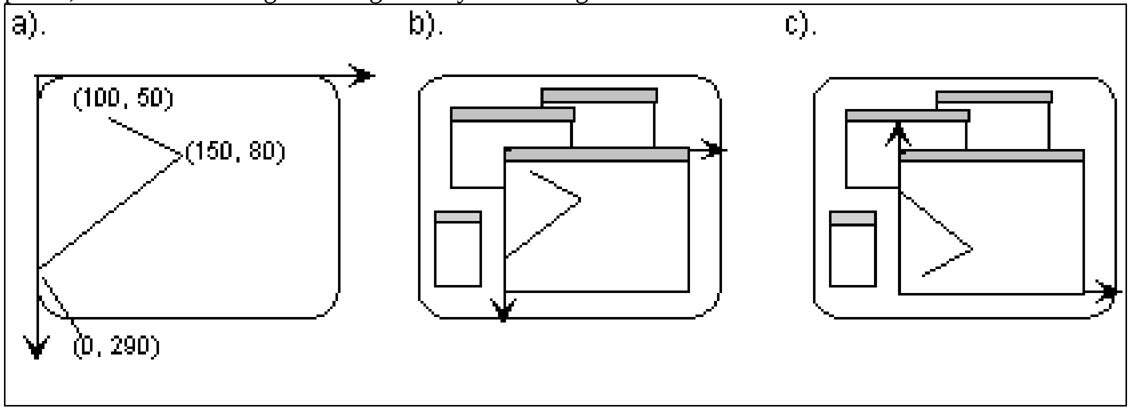

Figure 2.1. Some common varieties of display layouts.

In part b) a more modern "window-based" system is shown. It can support a number of different rectangular *windows* on the display screen at one time. Initialization involves creating and “opening” a new window (which we shall call the **screen window**[1]) for graphics. Graphics commands use a coordinate system that is attached to the window: usually *x* increases to the right and *y* increases downward[2]. Part c) shows a variation where the initial coordinate system is “right side up”, with *y* increasing upward[3].

Each system normally has some elementary drawing tools that help to get started. The most basic has a name like `setPixel(x, y, color)`: it sets the individual pixel at location (*x*, *y*) to the color specified by `color`. It sometimes goes by different names, such as `putPixel()`, `SetPixel()`, or `drawPoint()`. Along with `setPixel()` there is almost always a tool to draw a straight line, `line(x1, y1, x2, y2)`, that draws a line between (x1, y1) and (x2, y2). In other systems it might be called `drawLine()` or `Line()`. The commands

```
line(100, 50, 150, 80);
line(150, 80, 0, 290);
```

would draw the pictures shown in each system in Figure 2.1. Other systems have no `line()` command, but rather use `moveto(x, y)` and `lineto(x, y)`. They stem from the analogy of a pen plotter, where the pen has some **current position.** The notion is that `moveto(x, y)` moves the pen invisibly to location `(x, y)`, thereby setting the current position to (*x*, *y*); `lineto(x, y)` draws a line from the current position to (*x*, *y*), then updates the current position to this (*x*, *y*). Each command moves the pen from its current position to a new position. The new position then becomes the current position. The pictures in Figure 2.1 would be drawn using the commands

```
moveto(100, 50);
lineto(150, 80);
lineto(0, 290);
```

For a particular system the energetic programmer can develop a whole toolkit of sophisticated functions that utilize these elementary tools, thereby building up a powerful library of graphics routines. The final graphics applications are then written making use of this personal library.

An obvious problem is that each graphics display uses different basic commands to “drive it”, and every environment has a different collection of tools for producing the graphics primitives. This makes it difficult to *port* a program from one environment to another (and sooner or later everyone is faced with reconstructing a program in a new environment): the programmer must build the necessary tools on top of the new environment’s library. This may require major alterations in the overall structure of a library or application, and significant programmer effort.

2.1.1. Device Independent Programming, and OpenGL.

It is a boon when a uniform approach to writing graphics applications is made available, such that the *same* program can be compiled and run on a variety of graphics environments, with the guarantee that it will produce nearly identical graphical output on each display. This is known as **device independent** graphics programming. OpenGL offers such a tool. Porting a graphics program only requires that you install the appropriate OpenGL libraries on the new machine; the application itself requires no change: it calls the same functions in this library with the same parameters, and the same graphical results are produced. The OpenGL way of creating graphics has been adopted by a large number of industrial companies, and OpenGL libraries exist for all of the important graphics environments[4].

OpenGL is often called an “application programming interface” (API): the interface is a collection of routines that the programmer can call, along with a model of how the routines work together to produce graphics. The programmer “sees” only the interface, and is therefore shielded from having to cope with the specific hardware or software idiosyncrasies on the resident graphics system.

[1] The word "window" is overused in graphics: we shall take care to distinguish the various instances of the term.

[2] Example systems are unix workstations using X Windows, an IBM pc running Windows 95 using the basic Windows Application Programming Interface, and an Apple Macintosh using the built-in QuickDraw library.

[3] An example is any window-based system using OpenGL.

[4] Appendix 1 discusses how to obtain and get started with OpenGL in different environments.

OpenGL is at its most powerful when drawing images of complex three dimensional (3D) scenes, as we shall see. It might be viewed as overkill for simple drawings of 2D objects. But it works well for 2D drawing, too, and affords a *unified* approach to producing pictures. We start by using the simpler constructs in OpenGL, capitalizing for simplicity on the many default states it provides. Later when we write programs to produce elaborate 3D graphics we tap into OpenGL's more powerful features.

Although we will develop most of our graphics tools using the power of OpenGL, we will also "look under the hood" and examine how the classical graphics algorithms work. It is important to see how such tools might be implemented, even if for most applications you use the ready-made OpenGL versions. In special circumstances you may wish to use an alternative algorithm for some task, or you may encounter a new problem that OpenGL does not solve. You also may need to develop a graphics application that does not use OpenGL at all.

2.1.2. Windows-based programming.

As described above, many modern graphics systems are *windows-based*, and manage the display of multiple overlapping *windows*. The user can move the windows around the screen using the mouse, and can resize them. Using OpenGL we will do our drawing in one of these windows, as we saw in Figure 2.1c.

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 a mouse click, the press of a keyboard key, or the resizing of a screen window. The system automatically manages an *event queue*, which receives messages 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 occur. 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, then do the specified thing".

The method of associating a callback function with an event type 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 it is part of the OpenGL Utility Toolkit. The programmer puts code in `myMouse()` to handle all of the possible mouse actions 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 principle types of events we will work with, and a "glut" function is available for each:

```
void main()
{
     initialize things[5]
     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
```

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

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

- `glutDisplayFunc(myDisplay);` Whenever the system determines that a screen window should be redrawn 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. (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. `myMouse()` is automatically passed arguments that describe the mouse location and the nature of the button action.
- `glutKeyboardFunc(myKeyboard);` This registers the function `myKeyboard()` with the event of pressing or releasing some key on the keyboard. `myKeyboard()` is automatically passed arguments that tell which key was pressed. Conveniently, it is also passed data as to the location of the mouse at the time the key was pressed.

If a particular program does not use mouse interaction, 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 have no keyboard interaction.

The final function shown in Figure 2.2 is `glutMainLoop()`. When this 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 is to open a screen window for drawing. This can be quite involved, and is system dependent. Because OpenGL functions are device independent, they provide no support for window control on specific systems. But the OpenGL Utility Toolkit introduced above *does* include functions to open a window on whatever system you are using.

Figure 2.3 fleshes out the skeleton 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 toolkit to open a window for drawing with OpenGL. In your first graphics programs you can just copy these as is: later we will see what the various arguments mean and how to substitute others for them to achieve certain 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.

```
// 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

}
```
Figure 2.3. Code using the OpenGL utility toolkit to open the initial window for drawing.

- `glutInit(&argc, argv);` This function initializes the toolkit. Its arguments are the standard ones for passing command line information; 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`'d 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 this window as desired.
- `glutInitWindowPosition(100, 150);` This function specifies that the window's upper left corner should be positioned on the screen 100 pixels 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 mentioned above.

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. It is tied directly to the coordinate system of the screen window (see Figure 2.1c), and measures distances in pixels. Our first example screen window, shown in Figure 2.4, is 640 pixels wide by 480 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.

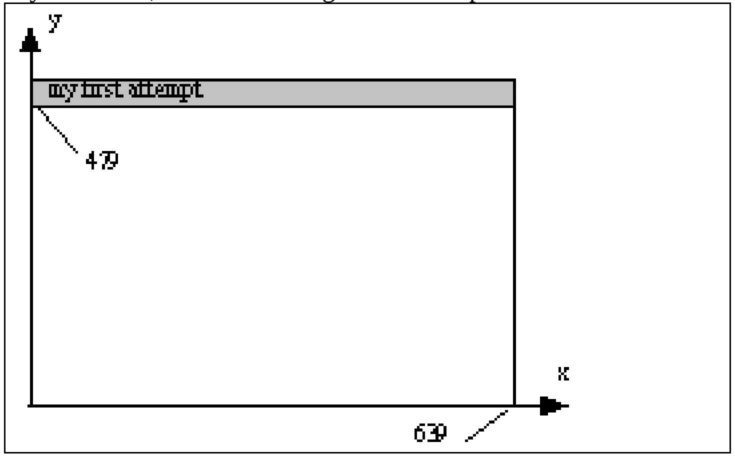

Figure 2.4. The initial coordinate system for drawing.

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 of 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:

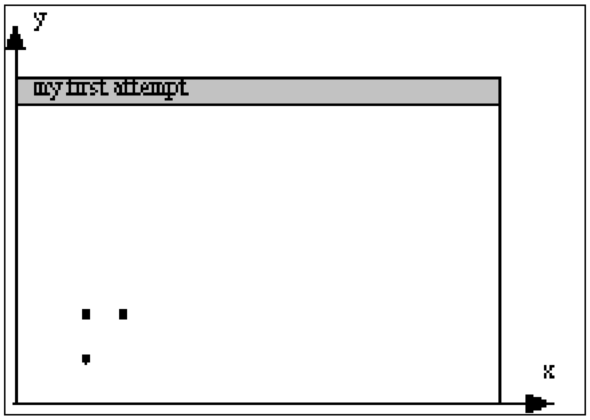

Figure 2.5. Drawing three dots.

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

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 vertex information down a "graphics pipeline", in which they go through several processing steps (look ahead to Figure ????). For present purposes just think of them as being sent more or less directly to the coordinate system in the screen window.

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

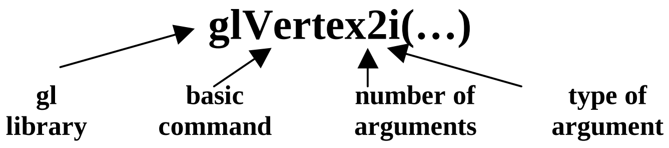

Figure 2.6. Format of OpenGL commands.

The prefix "`gl`" indicates a function from the OpenGL library (as opposed to "`glut`" for the utility toolkit). It is followed by the basic command root, then the number of arguments being sent to the function (this number will often be 3 and 4 in later contexts), and finally the type of argument, (`i` for an integer, `f` for a floating point value, etc., as we describe below). 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 above, for example, you could pass it floating point values instead of integers, using:

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

On 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 a `float` or `double` either. To insure that OpenGL functions receive the proper data types it is wise to use the built-in type names

like `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.

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**

Figure 2.7. Command suffixes and argument data types.

As an example, a function using 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:

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

which passes `int's` to `glVertex2i`(). This will work on systems that use 32-bit `int's`, but might cause trouble on those that use 16-bit `int's`. It is much safer to write `drawDot`() as in Figure 2.8, and to use `GLint's` 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 a generic function `drawDot`()[6].

The OpenGL "State".

OpenGL keeps track of many *state variables*, such as the current "size" of points, the current color of 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 its 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 appropriate OpenGL documentation (some of which is on-line; see Appendix 1). The drawing color 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 1.3.24??? could be set using:

```
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
```

[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.

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 our 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.

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

Figure 2.9. Establishing a simple coordinate system.

Putting it 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(). As this program is non-interactive, no other callback functions are used. glFlush() is called after the dots are drawn to insure that all data is completely processed and sent to the display. This is important in some systems that operate over a network: data is buffered on the host machine and only sent to the remote display when the buffer becomes full or a glFlush() is executed.

```
#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
}
```
Figure 2.10. A complete OpenGL program to draw three dots.

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 experience.

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.

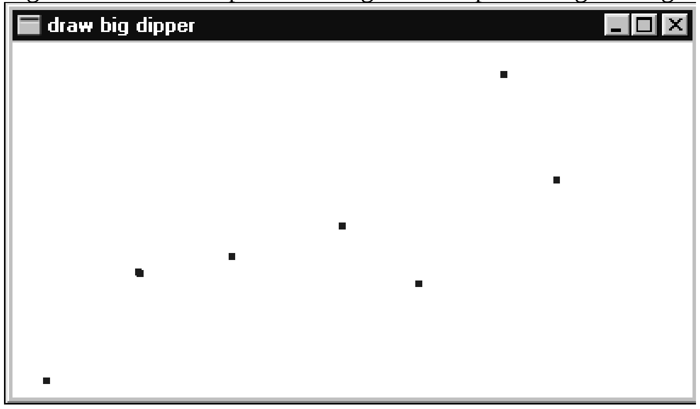

Figure 2.11. Two simple Dot Constellations.

The names and "positions" of the eight stars in the Big Dipper (for one particular view of the night sky), are given by: {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 "**hard-wire**" 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.

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 fra**c**tal (see Chapter 8)! We first approach the rules 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*.

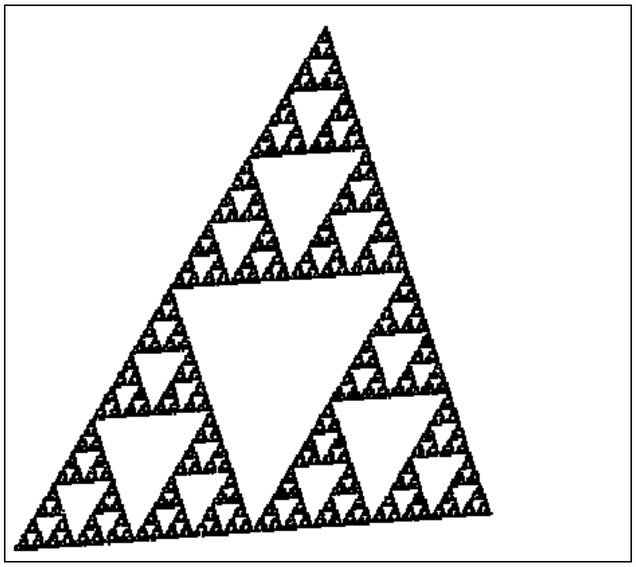
Figure 2.12. The Sierpinski Gasket. (file: fig2.12.bmp)

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),.... determined by a simple algorithm. Denote the k-th point $p_k = (x_k, y_k)$. Each point is based on the previous point p_{k-1}.The procedure is:

1. Choose three fixed points T_0, T_1, and T_2 to form some triangle, as shown in Figure 2.13a.
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 steps 3-5 until the pattern is satisfyingly filled in:

3. Chose 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}. Hence

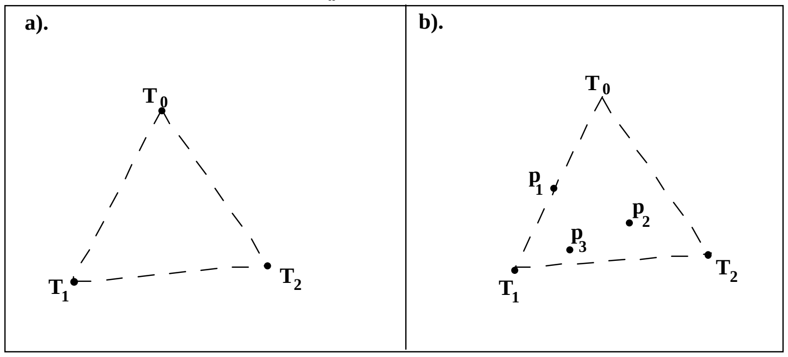

Figure 2.13. Building the Sierpinski gasket.

p_k = midpoint of p_{k-1} and T,

5. Draw p_k using `drawDot()`.

Figure 2.13b shows a few iterations of this: Suppose the initial point p_0 happens to be T_0, and that T_1 is 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 p_2 lies halfway between p_1 and T_2. Next suppose T_1 is chosen again, so 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 2 points, say (3, 12) and (5, 37) simply *average* their x and y components individually: add them and divide by 2. 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, or 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 1000 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 `struct` is useful here: `typedef struct{GLint x, y;}GLintPoint`;

[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.

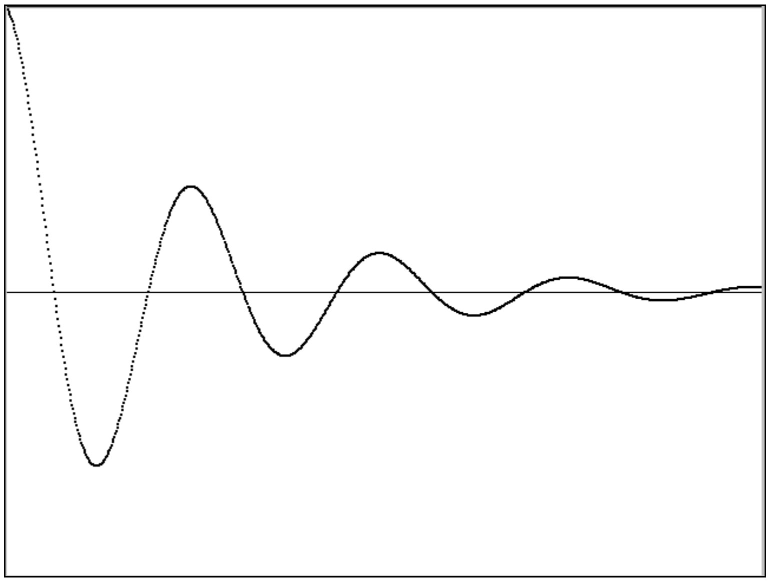
Figure 2.15. A “dot plot” of $e^{-x}cos(2\pi\, x)$ versus x. (file: fig2.15.bmp)

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 the process is basically:

```
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 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 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 height `screenHeight` pixels. Then to place the plot in the center of the window 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:

$$sx = A * x + B \qquad (2.1)$$
$$sy = C * y + D$$

for properly chosen values of the constants *A*, *B*, *C*, and *D*. *A* and *C* perform scaling; *B* and *D* perform shifting. This 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:

```
 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 to draw the dot plot, to illustrate how the various ingredients fit together. The initializations are very 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.

```
#include <windows.h> // use proper includes for your system
#include <math.h>
#include <gl/Gl.h>
#include <gl/glut.h>
const int screenWidth = 640;     // width of screen window in pixels
const int screenHeight = 480;    // height of screen window in pixels
GLdouble A, B, C, D;  // values used for scaling and shifting
//<<<<<<<<<<<<<<<<<<<<<<< myInit >>>>>>>>>>>>>>>>>>>
 void myInit(void)
 {
    glClearColor(1.0,1.0,1.0,0.0);       // background color is white
    glColor3f(0.0f, 0.0f, 0.0f);         // drawing color is black
    glPointSize(2.0);                   // a 'dot' is 2 by 2 pixels
    glMatrixMode(GL_PROJECTION);       // set "camera shape"
    glLoadIdentity();
    gluOrtho2D(0.0, (GLdouble)screenWidth, 0.0, (GLdouble)screenHeight);
    A = screenWidth / 4.0; // set values used for scaling and shifting
    B = 0.0;
    C = D = screenHeight / 2.0;
}
//<<<<<<<<<<<<<<<<<<<<<<<< myDisplay >>>>>>>>>>>>>>>>
void myDisplay(void)
{
  glClear(GL_COLOR_BUFFER_BIT);     // clear the screen
  glBegin(GL_POINTS);
  for(GLdouble x = 0; x < 4.0 ; x += 0.005)
  {
     Gldouble func = exp(-x) * cos(2 * 3.14159265 * x);
     glVertex2d(A * x + B, C * func + D);
   }
  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(screenWidth, screenHeight); // set window size
  glutInitWindowPosition(100, 150); // set window position on screen
  glutCreateWindow("Dot Plot of a Function"); // open the screen window
  glutDisplayFunc(myDisplay);     // register redraw function
  myInit();
  glutMainLoop();           // go into a perpetual loop
}
```
Figure 2.16. A complete program to draw the "dot plot" of a function.

Practice Exercise 2.2.2. Dot plots for any function *f*(). Consider drawing a dot plot of the function *f*(.) as in Example 2.2.4, where it is known that as x varies from x_{low} to x_{high}, *f*(*x*) takes on values between y_{low} to y_{high}. Find the appropriate scaling and translation factors 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 end points as vertices. Thus to draw a line between (`40,100`) and (`202,96`) use:

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

This code might be encapsulated for convenience in the routine `drawLineInt`():

```
void drawLineInt(GLint x1, GLint y1, GLint x2, GLint y2)
{
   glBegin(GL_LINES);
   glVertex2i(x1, y1);
   glVertex2i(x2, y2);
   glEnd();
}
```

and an alternate routine, `drawLineFloat`() could be implemented similarly (how?).

If more than two vertices are specified between `glBegin`(`GL_LINES`) and `glEnd`() they are taken in pairs and a separate line is drawn between each pair. The tic-tac-toe board shown in Figure 2.17a would be drawn using:

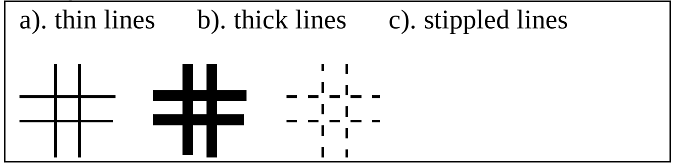

Figure 2.17. Simple picture built from four lines.

```
glBegin(GL_LINES);
   glVertex2i(10, 20);  // first horizontal line
   glVertex2i(40, 20)
   glVertex2i(20, 10);  // first vertical line
   glVertex2i(20, 40);
   <four more calls to glVertex2i() here for other two lines>
glEnd();
glFlush();
```

OpenGL provides tools for setting the attributes of lines. A line's color is set in the same way as for points, using `glColor3f()`. Figure 2.17b shows the use of thicker lines, as set by `glLineWidth`(`4.0`). The default thickness is 1.0. Figure 2.17c shows stippled (dotted and dashed) lines. The details of stippling are addressed in Case Study 2.5 at the end of this chapter.

2.3.1. Drawing Polylines and Polygons.

Recall from Chapter 1 that a **polyline** is a collection of line segments joined end to end. It is described by an ordered list of points, as in**:**

$$p_0 = (x_0, y_0),\ p_1 = (x_1, y_1), \ldots ,\ p_n = (x_n, y_n). \qquad (2.3.1)$$

In OpenGL a polyline is called a "line strip", and is drawn by specifying the vertices in turn between `glBegin(GL_LINE_STRIP)` and `glEnd()`. For example, the code:

```
glBegin(GL_LINE_STRIP);  // draw an open polyline
   glVertex2i(20,10);
   glVertex2i(50,10);
   glVertex2i(20,80);
   glVertex2i(50,80);
glEnd();
glFlush();
```

produces the polyline shown in Figure 2.18a. Attributes such as color, thickness and stippling may be applied to polylines in the same way they are applied to single lines. If it is desired to connect the last point with the first point to make the polyline into a polygon simply replace `GL_LINE_STRIP` with `GL_LINE_LOOP`. The resulting polygon is shown in Figure 2.18b.

Figure 2.18. A polyline and a polygon.

Polygons drawn using `GL_LINE_LOOP` cannot be filled with a color or pattern. To draw filled polygons you use `glBegin(GL_POLYGON)`, as described later.

Example 2.3.1. Drawing Line Graphs. In Example 2.2.3 we looked at plotting a function $f(x)$ versus x with a sequence of dots at positions $(x_i, f(x_i))$. A line graph is a straightforward extension of this: the dots are simply joined by line segments to form a polyline. Figure 2.19 shows an example, based on the function:

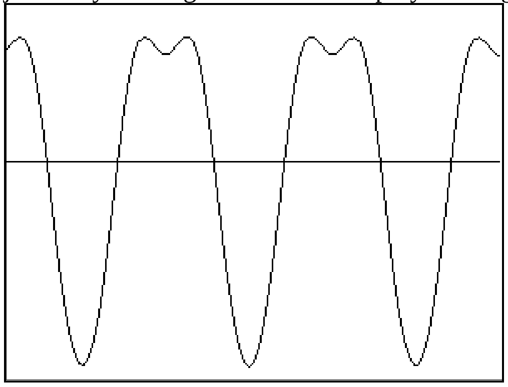

Figure 2.19. A plot of a mathematical formula.

$$f(x) = 300 - 100\ cos(2\pi\ x/100) + 30\ cos(4\pi\ x/100) + 6\ cos(6\pi\ x/100)$$

as x varies in steps of 3 for 100 steps. A blow-up of this figure would show a sequence of connected line segments; in the normal size picture they blend together and appear as a smoothly varying curve.

The process of plotting a function with line segments is almost identical to that for producing a dot plot: the program of Figure 2.17 can be used with only slight adjustments. We must scale and shift the lines being drawn here, to properly place the lines in the window. This requires the computation of the constants A, B, C and D in the same manner as we did before (see Equation 2.1). Figure 2.20 shows the changes necessary for the inner drawing loop in the `myDisplay()` function.

```
< Calculate constants A, B, C and D for scaling and shifting>
```

```
glBegin(GL_LINE_STRIP);
  for(x = 0; x <= 300; x += 3)
     glVertex2d(A * x + B, C * f(x) + D);
glEnd();
glFlush;
```

Figure 2.20. Plotting a function using a line graph.

Example 2.3.2. Drawing Polylines stored in a file.
Most interesting pictures made up of polylines contain a rather large number of line segments. It’s convenient to store a description of the polylines in a file, so that the picture can be redrawn at will. (Several interesting examples may be found on the Internet - see the Preface.)

It’s not hard to write a routine that draws the polylines stored in a file. Figure 2.21 shows an example of what might be drawn.

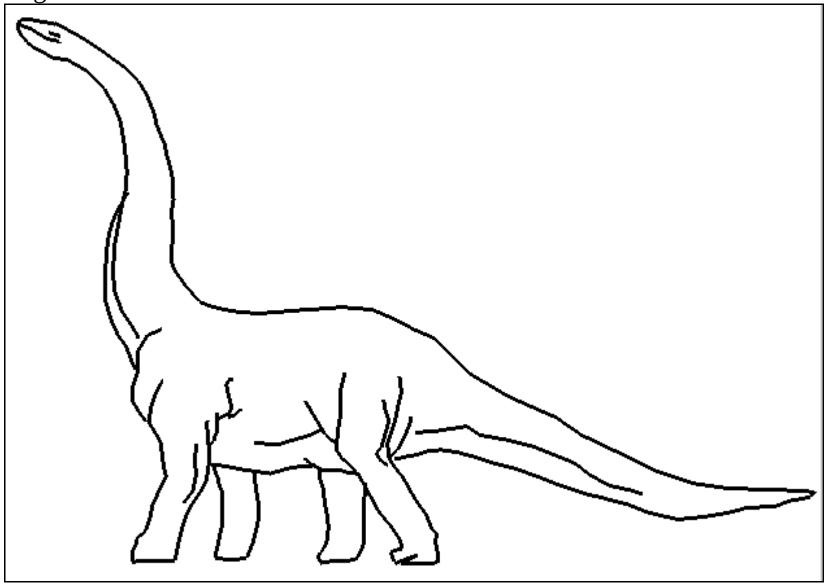

Figure 2.21. Drawing polylines stored in a file. (file: fig2.21.bmp)

Suppose the file `dino.dat` contains a collection of polylines, in the following format (the comments are not part of the file):

```
21                            number of polylines in the file
4                             number of points in the first polyline
169 118                       first point of first polyline
174 120                       second point of first polyline
179 124
178 126
5                             number of points in the second polyline
298 86                        first point of second polyline
304 92
310 104
314 114
314 119
29
32 435
10 439
. . .                         etc.
```

(The entire file is available on the web site for this book. See the preface.) Figure 2.22 shows a routine in C++ that will open such a file, and then draw each of the polylines it contains. The file having the name contained in the string `fileName` is read in and each polyline is drawn. The routine could be used in place of `myDisplay`() in Figure 2.17 as the callback function for the redraw event. The values of *A*, *B*, *C* and *D* would have to be chosen judiciously to scale the polylines properly. We develop a general approach to do this in Chapter 3.

```
void drawPolyLineFile(char * fileName)
```

```
{
     fstream inStream;
     inStream.open(fileName, ios ::in);  // open the file
     if(inStream.fail())
        return;
     glClear(GL_COLOR_BUFFER_BIT);      // clear the screen
     GLint numpolys, numLines, x ,y;
     inStream >> numpolys;              // read the number of polylines
     for(int j = 0; j < numpolys; j++)  // read each polyline
     {
        inStream >> numLines;
        glBegin(GL_LINE_STRIP);         // draw the next polyline
        for (int i = 0; i < numLines; i++)
        {
            inStream >> x >> y;        // read the next x, y pair
            glVertex2i(x, y);
        }
        glEnd();
     }
     glFlush();
     inStream.close();
}
```
Figure 2.22. Drawing polylines stored in a file.

This version of `drawPolyLineFile()`does very little error checking. If the file cannot be opened — perhaps the wrong name is passed to the function — the routine simply returns. If the file contains bad data, such as real values where integers are expected, the results are unpredictable. The routine as given should be considered only as a starting point for developing a more robust version.

Example 2.3.3. Parameterizing Figures.
Figure 2.23 shows a simple house consisting of a few polylines. It can be drawn using code shown partially in Figure 2.24. (What code would be suitable for drawing the door and window?)

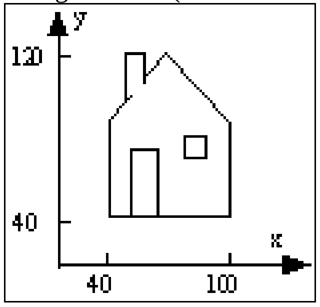

Figure 2.23. A House.

```
void hardwiredHouse(void)
{
   glBegin(GL_LINE_LOOP);
      glVertex2i(40, 40);     // draw the shell of house
      glVertex2i(40, 90);
      glVertex2i(70, 120);
      glVertex2i(100, 90);
      glVertex2i(100, 40);
   glEnd();
   glBegin(GL_LINE_STRIP);
      glVertex2i(50, 100);    // draw the chimney
      glVertex2i(50, 120);
      glVertex2i(60, 120);
      glVertex2i(60, 110);
   glEnd();
      . . . // draw the door
      . . . // draw the window
}
```
Figure 2.24. Drawing a house with “hard-wired” dimensions.

This is not a very flexible approach. The position of each endpoint is hard-wired into this code, so `hardwirededHouse`() can draw only one house in one size and one location. More flexibility is achieved if we **parameterize** the figure, and pass the parameter values to the routine. In this way we can draw **families** of objects, which are distinguished by different parameter values. Figure 2.25 shows this approach. The parameters specify the location of the peak of the roof, the width of the house, and its height. The details of drawing the chimney, door, and window are left as an exercise.

```
void parameterizedHouse(GLintPoint peak, GLint width, GLint height)
 // the top of house is at the peak; the size of house is given
 //  by height and width
{
   glBegin(GL_LINE_LOOP);
     glVertex2i(peak.x,             peak.y);  // draw shell of house
     glVertex2i(peak.x + width / 2, peak.y - 3 * height /8);
     glVertex2i(peak.x + width / 2  peak.y -     height);
     glVertex2i(peak.x - width / 2, peak.y -     height);
     glVertex2i(peak.x - width / 2, peak.y - 3 * height /8);
   glEnd();
   draw chimney in the same fashion
   draw the door
   draw the window
}
```

Figure 2.25. Drawing a parameterized house.

This routine may be used to draw a "village" as shown in Figure 2.26, by making successive calls to `parameterizedHouse()` with different parameter values. (How is a house "flipped" upside down? Can *all* of the houses in the figure be drawn using the routine given?)

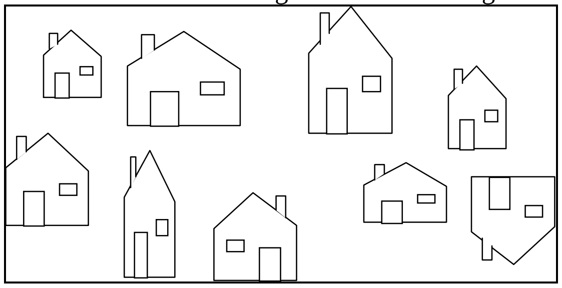

Figure 2.26. A "village" of houses drawn using `parameterizedHouse()`.

Example 2.3.4. Building a Polyline Drawer.
As we shall see, some applications compute and store the vertices of a polyline in a list. It is natural, therefore, to add to our growing toolbox of routines a function that accepts the list as a parameter and draws the corresponding polyline. The list might be in the form of an array, or a linked list. We show here the array form, and define the class to hold it in Figure 2.27.

```
class GLintPointArray{
   const int MAX_NUM = 100;
   public:
      int num;
      GLintPoint pt[MAX_NUM];
};
```

Figure 2.27. Data type for a linked list of vertices.

Figure 2.28 shows a possible implementation of the polyline drawing routine. It also takes a parameter `closed`: if `closed` is nonzero the last vertex in the polyline is connected to the first vertex. The value of `closed` sets the argument of `glBegin`(). The routine simply sends each vertex of the polyline to OpenGL.

```
void drawPolyLine(GlintPointArray poly, int closed)
{
      glBegin(closed ? GL_LINE_LOOP : GL_LINE_STRIP);
        for(int i = 0; i < poly.num; i++)
              glVertex2i(poly.pt[i].x, poly.pt[i].y);
      glEnd();
      glFlush();
}
```

Figure 2.28. A linked list data type, and drawing a polyline or polygon.

2.3.3. Line Drawing using `moveto()` and `lineto()`.

As we noted earlier a number of graphics systems provide line drawing tools based on the functions `moveto`() and `lineto`(). These functions are so common it is important to be familiar with their use. We shall fashion our own `moveto`() and `lineto`() that operate by calling OpenGL tools. In Chapter 3 we shall also dive "under the hood" to see how you would build `moveto`() and `lineto`() based on first principles, if a powerful library like OpenGL were not available.

Recall that `moveto`() and `lineto`() manipulate the position of a hypothetical pen, whose position is called the **current position**, or *CP*. We can summarize the effects of the two functions as:

`moveto`(x, y): set *CP* to (x, y)
`lineto`(x, y): draw a line from *C2970P* to (*x*, *y*), and then update *CP* to (*x*, *y*)

A line from (x_1, y_1) to (x_2, y_2) is therefore drawn using the two calls `moveto(x1, y1);lineto(x2, y2)`. A polyline based on the list of points (x_0, y_0), $(x_1, y1)$, ... , (x_{n-1}, y_{n-1}) is easily drawn using:

```
moveto(x[0], y[0]);
for(int i = 1; i < n; i++)
   lineto(x[i], y[i]);
```

It is straightforward to build `moveto`() and `lineto`() on top of OpenGL. To do this we must define and maintain our own *CP*. For the case of integer coordinates the implementation shown in Figure 2.29 would do the trick.

```
GLintPoint CP;         // global current position

//<<<<<<<<<<<<< moveto >>>>>>>>>>>>>>
void moveto(GLint x, GLint y)
{
   CP.x = x; CP.y = y; // update the CP
}
//<<<<<<<<<<<< lineTo >>>>>>>>>>>>>>>>
void lineto(GLint x, GLint y)
{
   glBegin(GL_LINES);  // draw the line
      glVertex2i(CP.x, CP.y);
      glVertex2i(x, y);
   glEnd();
   glFlush();
   CP.x = x; CP.y = y; // update the CP
}
```

Figure 2.29. Defining `moveto`() and `lineto`() in OpenGL.

2.3.4. Drawing Aligned Rectangles.

A special case of a polygon is the **aligned rectangle,** so called because its sides are aligned with the coordinate axes. We could create our own function to draw an aligned rectangle (how?), but OpenGL provides the ready-made function:

```
glRecti(GLint x1, GLint  y1, GLint x2, GLint y2);
// draw a rectangle with opposite corners (x1, y1) and (x2, y2);
// fill it with the current color;
```

This command draws the aligned rectangle based on two given points. In addition the rectangle is filled with the current color. Figure 2.30 shows what is drawn by the code:

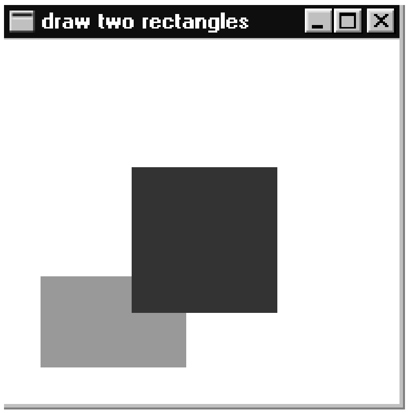

Figure 2.30. Two aligned rectangles filled with colors.

```
  glClearColor(1.0,1.0,1.0,0.0); // white background
  glClear(GL_COLOR_BUFFER_BIT);  // clear the window
  glColor3f(0.6,0.6,0.6);             // bright gray
  glRecti(20,20,100,70);
  glColor3f(0.2,0.2,0.2);             // dark gray
  glRecti(70, 50, 150, 130);
  glFlush();
```

Notice that the second rectangle is "painted over" the first one. We examine other "drawing modes" in Chapter 10.

Figure 2.31 shows two further examples. Part a) is a "flurry" of randomly chosen aligned rectangles, that might be generated by code such as[10]:

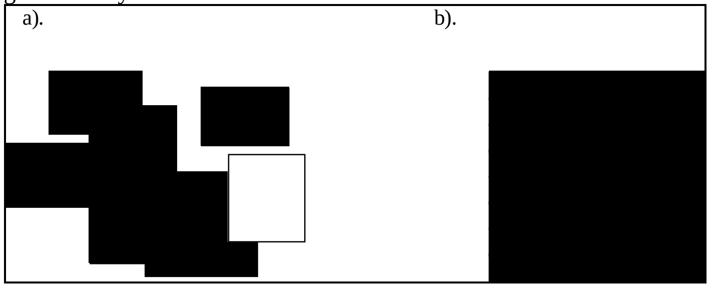

Figure 2.31. a). Random Flurry of rectangles. b). a checkerboard.

```
void drawFlurry(int num, int numColors, int Width, int Height)
// draw num random rectangles in a Width by Height rectangle
{
  for (int i = 0; i < num; i++)
  {
    GLint x1 = random(Width);           // place corner randomly
    GLint y1 = random(Height);
    GLint x2 = random(Width);           // pick the size so it fits
    GLint y2 = random(Height);
    GLfloat lev = random(10)/10.0;      // random value, in range 0 to 1
    glColor3f(lev,lev,lev);             // set the gray level
    glRecti(x1, y1, x2, y2);            // draw the rectangle
  }
  glFlush();

}
```

Part b) is the familiar checkerboard, with alternating gray levels. The exercises ask you to generate it.

2.3.5. Aspect Ratio of an Aligned Rectangle.

[10] Recall that `random(N)` returns a randomly-chosen value between 0 and N - 1 (see Appendix 3).

The principal properties of an aligned rectangle are its size, position, color, and "shape". Its shape is embodied in its **aspect ratio**, and we shall be referring to the aspect ratios of rectangles throughout the book. The **aspect ratio** of a rectangle is simply the ratio of its width to its height[11]:

$$aspect\,ratio = \frac{width}{height} \tag{2.2}$$

Rectangles with various aspect ratios are shown in Figure 2.32..

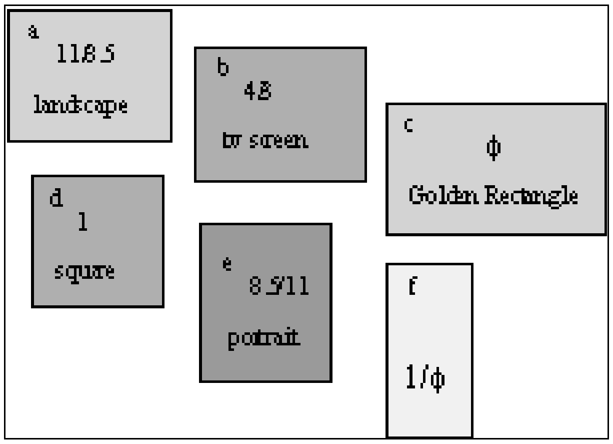

Figure 2.32. Examples of aspect ratios of aligned rectangles.

Rectangle *A* has the shape of a piece of 8.5 by 11 inch paper laid on its side in the so-called **landscape** orientation (i.e. width larger than height). It has an aspect ratio of 1.294. Rectangle *B* has the aspect ratio of a television screen, 4/3, and *C* is the famous **golden rectangle** described in Case Study 2.3. Its aspect ratio is close to ϕ = 1.618034. Rectangle *D* is a square with aspect ratio equal to 1, and *E* has the shape of a piece of standard paper in **portrait** orientation, with an aspect ratio of .7727. Finally, *F* is tall and skinny with an aspect ratio of 1/ϕ.

Practice Exercises.

2.3.1. Drawing the checkerboard. (Try your hand at this before looking at the answers.)
Write the routine `checkerboard(int size)` that draws the checkerboard shown in Figure 2.31b. Place the checkerboard with its lower left corner at (0,0). Each of the 64 squares has length `size` pixels. Choose two nice colors for the squares. **Solution:** The ij-th square has lower left corner at (i*size, j*size) for i = 0,..,7 and j = 0..7. The color can be made to alternate between (r_1, g_1, b_1) and (r_2, g_2, b_2) using

```
if((i + j)%2 ==0) // if i + j is even
  glColor3f( r1, g1, b1);
else
  glColor3f(r2, g2, b2);
```

2.3.2. Alternative ways to specify a rectangle. An aligned rectangle can be described in other ways than by two opposite corners. Two possibilities are:
* its center point, height, and width;
* its upper left corner, width, and aspect ratio.
Write functions `drawRectangleCenter()` and `drawRectangleCornerSize()` that pass these alternative parameters.

2.3.3. Different Aspect Ratios. Write a short program that draws a filled rectangle of aspect ratio *R*, where *R* is specified by the user. Initialize the display to a drawing space of 400 by 400. Arrange the size of the rectangle so that it is as large as possible. That is, if *R* > 1 it spans across the drawing space, and if *R* < 1 it spans from top to bottom.

2.3.4. Drawing the Parametrized house. Fill in the details of `parametrizedHouse`() in Figure 2.25 so that the door, window, and chimney are drawn in their proper proportions for given values of `height` and `width`.

[11]Alert! Some authors define it as height / width.

2.3.5. Scaling and positioning a figure using parameters. Write the function `void drawDiamond(GLintPoint center, int size)` that draws the simple diamond shown in Figure 2.33, centered at `center`, and having size `size`.

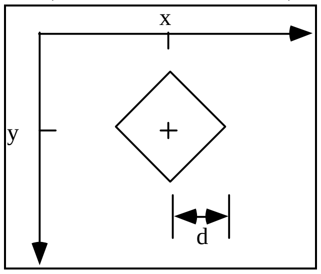

Figure 2.33. A simple diamond.

Use this function to draw a "flurry" of diamonds as suggested in Figure 2.34.

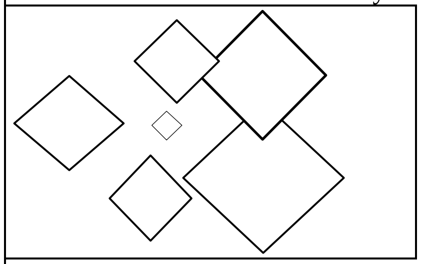
Figure 2.34. A flurry of diamonds.

2.3.6. Filling Polygons.

So far we can draw unfilled polygons in OpenGL, as well as aligned rectangles filled with a single solid color. OpenGL also supports filling more general polygons with a pattern or color. The restriction is that the polygons must be *convex*.

Convex polygon:
a polygon is convex if a line connecting any two points of the polygon lies entirely within the polygon.

Several polygons are shown in Figure 2.35. Of these only *D*, *E*, and *F* are convex. (Check that the definition of convexity is upheld for each of these polygons.) *D* is certainly convex: all triangles are. *A* is not even simple (recall Chapter 1) so it cannot be convex. Both *B* and *C* "bend inward" at some point. (Find two points on *B* such that the line joining them does not lie entirely inside *B*.)

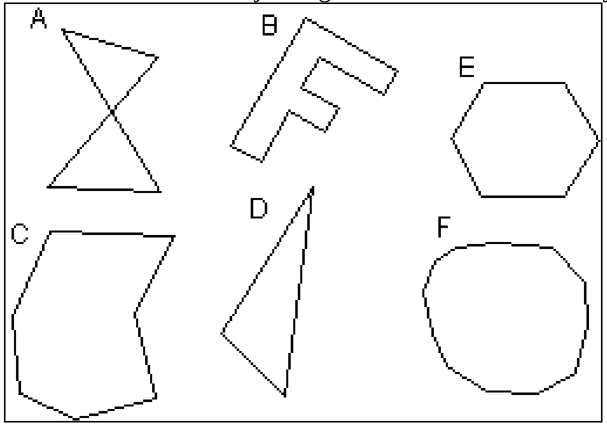

Figure 2.35. Convex and non-convex polygons.

To draw a convex polygon based on vertices (x_0, y_0), (x_1, y_1), …, (x_n, y_n) use the usual list of vertices, but place them between a `glBegin(GL_POLYGON)` and an `glEnd()`:

```
glBegin(GL_POLYGON);
   glVertex2f(x0, y0);
   glVertex2f(x1, y1);
   . . .
   glVertex2f(xn, yn);
 glEnd();
```

It will be filled in the current color. It can also be filled with a stipple pattern – see Case Study 2.5, and later we will paint images into polygons as part of applying a texture.

Figure 2.36 shows a number of filled convex polygons. In Chapter 10 we will examine an algorithm for filling any polygon, convex or not.

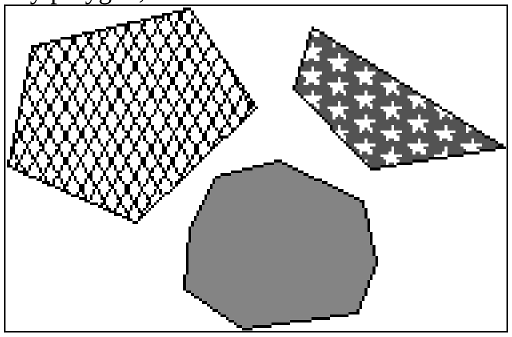
Figure 2.36. Several filled convex polygons.

2.3.7. Other Graphics Primitives in OpenGL.

OpenGL supports the drawing of five other objects as well. Figure 2.37 shows examples of each of them. To draw a particular one the constant shown with it is used in `glBegin`().

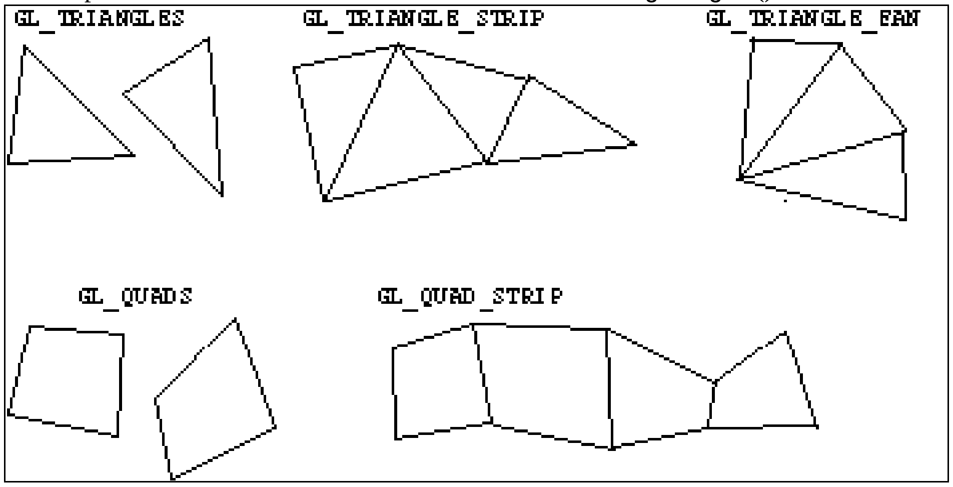

Figure 2.37. Other geometric primitive types.

The following list explains the function of each of the five constants:

- GL_TRIANGLES: takes the listed vertices three at a time, and draws a separate triangle for each;
- GL_QUADS: takes the vertices four at a time and draws a separate quadrilateral for each;
- GL_TRIANGLE_STRIP: draws a series of triangles based on triplets of vertices: v_0, v_1, v_2, then v_2, v_1, v_3, then v_2, v_3, v_4, etc. (in an order so that all triangles are "traversed" in the same way; e.g. counterclockwise).
- GL_TRIANGLE_FAN: draws a series of connected triangles based based on triplets of vertices: v_0, v_1, v_2, then v_0, v_2, v_3, then v_0, v_3, v_4, etc.
- GL_QUAD_STRIP: draws a series of quadrilaterals based on foursomes of vertices: first v_0, v_1, v_3, v_2, then v_2, v_3, v_5, v_4, then v_4, v_5, v_7, v_6 (in an order so that all quadrilaterals are "traversed" in the same way; e.g. counterclockwise).

2.4. Simple Interaction with the mouse and keyboard.

Interactive graphics applications let the user control the flow of a program by natural human motions: pointing and clicking the mouse, and pressing various keyboard keys. The mouse position at the time of the click, or the identity of the key pressed, is made available to the application program and is processed as appropriate.

Recall that when the user presses or releases a mouse button, moves the mouse, or presses a keyboard key, an event occur. Using the OpenGL Utility Toolkit (GLUT) the programmer can register a callback function with each of these events by using the following commands:

- `glutMouseFunc(myMouse)` which registers `myMouse`() with the event that occurs when the mouse button is pressed or released;
- `glutMotionFunc(myMovedMouse)` which registers `myMovedMouse`() with the event that occurs when the mouse is moved while one of the buttons is pressed;
- `glutKeyboardFunc(myKeyboard)` which registers `myKeyBoard`() with the event that occurs when a keyboard key is pressed.

We next see how to use each of these.

2.4.1. Mouse interaction.

How is data about the mouse sent to the application? You must design the callback function `myMouse`() to take four parameters, so that it has the prototype:

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

When a mouse event occurs the system calls the registered function, supplying it with values for these parameters. The value of `button` will be one of:

```
GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, or GLUT_RIGHT_BUTTON,
```

with the obvious interpretation, and the value of `state` will be one of: `GLUT_UP` or `GLUT_DOWN`. The values `x` and `y` report the position of the mouse at the time of the event. **Alert**: The `x` value is the number of pixels from the left of the window as expected, but the `y` value is the number of pixels *down* from the top of the window!

Example 2.4.1. Placing dots with the mouse.
We start with an elementary but important example. Each time the user presses down the left mouse button a dot is drawn in the screen window at the mouse position. If the user presses the right button the program terminates. The version of `myMouse`() shown next does the job. Because the *y*-value of the mouse position is the number of pixels from the top of the screen window, we draw the dot, not at (`x, y`), but at (`x, screenHeight` – *y*), where `screenHeight` is assumed here to be the height of the window in pixels.

```
void myMouse(int button, int state, int x, int y)
{
      if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
            drawDot(x, screenHeight -y);
      else if(button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
      exit(-1);}
```

The argument of –1 in the standard function `exit`() simply returns –1 back to the operating system: It is usually ignored.

Example 2.4.2. Specifying a rectangle with the mouse.
Here we want the user to be able to draw rectangles whose dimensions are entered with the mouse. The user clicks the mouse at two points which specify opposite corners of an aligned rectangle, and the rectangle is drawn. The data for each rectangle need not be retained (except through the picture of the rectangle itself): each new rectangle replaces the previous one. The user can clear the screen by pressing the right mouse button.

The routine shown in Figure 2.38 stores the corner points in a `static` array `corner[]`. It is made `static` so values are retained in the array between successive calls to the routine. Variable `numCorners` keeps track of how many corners have been entered so far: when this number reaches two the rectangle is drawn, and `numCorners` is reset to 0.

```
void myMouse(int button, int state, int x, int y)
{
    static GLintPoint corner[2];
    static int numCorners = 0;                    // initial value is 0
    if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    {
       corner[numCorners].x = x;
       corner[numCorners].y = screenHeight - y;    // flip y coordinate
       numCorners++;                               // have another point
       if(numCorners == 2)
       {
          glRecti(corner[0].x, corner[0].y, corner[1].x, corner[1].y);
          numCorners = 0;                          // back to 0 corners
       }
    }
    else if(button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
            glClear(GL_COLOR_BUFFER_BIT);          // clear the window
    glFlush();
}
```

Figure 2.38. A callback routine to draw rectangles entered with the mouse.

An alternative method for designating a rectangle uses a **rubber rectangle** that grows and shrinks as the user moves the mouse. This is discussed in detail in Section 10.3.3.

Example 2.4.3. Controlling the Sierpinski gasket with the mouse.
It is simple to extend the Sierpinski gasket routine described earlier so that the user can specify the three vertices of the initial triangle with the mouse. We use the same process as in the previous example: gather the three points in an array `corners`[], and when three points are available draw the Sierpinski gasket. The meat of the `myMouse`() routine is therefore:

```
    static GLintPoint corners[3];
    static int numCorners = 0;
    if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    {
       corner[numCorners].x = x;
       corner[numCorners].y = screenHeight - y;       // flip y coordinate
       if(++numCorners == 3)
       {
          Sierpinski(corners);                  // draw the gasket
          numCorners = 0;                       // back to 0 corners
       }
   }
```

where `Sierpinski()` is the same as in Figure 2.15 except the three vertices of the triangle are passed as parameters.

Example 2.4.4. Create a polyline using the mouse.
Figure 2.39 shows a polyline being created with mouse clicks. Here, instead of having each new point replace the previous one we choose to retain all the points clicked for later use. The user enters a succession of points with the mouse, and each point is stored in the next available position of the array. If the array becomes full no further points are accepted. After each click of the mouse the window is cleared and the entire current polyline is redrawn. The polyline is reset to empty if the right mouse button is pressed.

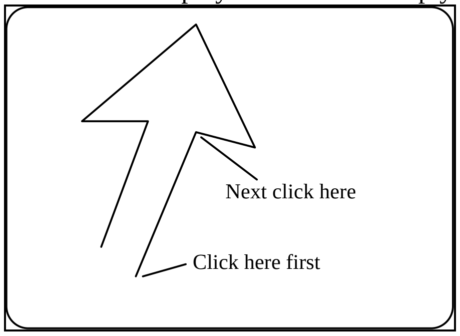

Figure 2.39. Interactive creation of a polyline.

Figure 2.40 shows one possible implementation. Note that `last` keeps track of the last index used so far in the array `List[]`; it is incremented as each new point is clicked, and set to –1 to make the lists empty. If it were desirable to make use of the points in `List` outside of `myMouse()`, the variable `List` could be made global.

```
void myMouse(int button, int state, int x, int y)
{
 #define NUM 20
 static GLintPoint List[NUM];
    static int last = -1;                  // last index used so far

   // test for mouse button as well as for a full array
   if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN && last < NUM -1)
   {
      List[++last].x = x;                // add new point to list
      List[  last].y = screenHeight - y; // window height is 480
      glClear(GL_COLOR_BUFFER_BIT);      // clear the screen
      glBegin(GL_LINE_STRIP);            // redraw the polyline
        for(int i = 0; i <= last; i++)
         glVertex2i(List[i].x, List[i].y);
      glEnd();
      glFlush();
 }
 else if(button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
last = -1;            // reset the list to empty
}
```

Figure 2.40. A polyline drawer based on mouse clicks.

Mouse motion.
An event of a different type is generated when the mouse is moved (more than some minimal distance) while some button is held down. The callback function, say `myMovedMouse()`, is registered with this event using

```
glutMotionFunc(myMovedMouse);
```

The callback function must take two parameters and have the prototype: `myMovedMouse(int x, int y);` The values of `x` and `y` are of course the position of the mouse when the event occurred.

Example 2.4.4. "Freehand" drawing with a fat brush.
Suppose we want to create a curve by sweeping the mouse along some trajectory with a button held down. In addition we want it to seem that the drawing "brush" has a square shape. This can be accomplished by designing `myMovedMouse()` to draw a square at the current mouse position:

```
void myMovedMouse(int mouseX, int mouseY)
{
   GLint x = mouseX;                //grab the mouse position
   GLint y = screenHeight – mouseY;       // flip it as usual
   GLint brushSize = 20;
   glRecti(x,y, x + brushSize, y + brushSize);
   glFlush();
}
```

2.4.2. Keyboard interaction.
As mentioned earlier, pressing a key on the keyboard queues a keyboard event. The callback function `myKeyboard()` is registered with this type of event through `glutKeyboardFunc(myKeyboard)`. It must have prototype:

```
void myKeyboard(unsigned int key, int x, int y);
```

The value of key is the ASCII value[12] of the key pressed. The values `x` and `y` report the position of the mouse at the time that the event occurred. (As before *y* measures the number of pixels down from the top of the window.)

The programmer can capitalize on the many keys on the keyboard to offer the user a large number of choices to invoke at any point in a program. Most implementations of `myKeyboard`() consist of a large `switch` statement, with a `case` for each key of interest. Figure 2.41 shows one possibility. Pressing 'p' draws a dot at the mouse position; pressing the left arrow key adds a point to some (global) list, but does no drawing[13]; pressing 'E' exits from the program. Note that if the user holds down the 'p' key and moves the mouse around a rapid sequence of points is generated to make a "freehand" drawing.

```
void myKeyboard(unsigned char theKey, int mouseX, int mouseY)
{
  GLint x = mouseX;
  GLint y = screenHeight - mouseY; // flip the y value as always
  switch(theKey)
  {
    case ‘p’:
       drawDot(x, y);   // draw a dot at the mouse position
       break;
    case GLUT_KEY_LEFT: List[++last].x = x; // add a point
                        List[  last].y = y;
       break;
    case ‘E’:
       exit(-1);        //terminate the program
    default:
     break;             // do nothing
 }
}
```

Figure 2.41. An example of the keyboard callback function.

2.5. Summary

The hard part in writing graphics applications is getting started: pulling together the hardware and software ingredients in a program to make the first few pictures. The OpenGL application programmer interface (API) helps enormously here, as it provides a powerful yet simple set of routines to make drawings. One of its great virtues is device independence, which makes it possible to write programs for one graphics environment, and use the same program without changes in another environment.

Most graphics applications are written today for a windows-based environment. The program opens a window on the screen that can be moved and resized by the user, and it responds to mouse clicks and key strokes. We saw how to use OpenGL functions that make it easy to create such a program.

Primitive drawing routines were applied to making pictures composed of dots, lines, polylines, and polygons, and were combined into more powerful routines that form the basis of one's personal graphics toolkit. Several examples illustrated the use of these tools, and described methods for interacting with a program using the keyboard and mouse. The Case studies presented next offer additional programming examples that explore deeper into the topics discussed so far, or branch out to interesting related topics.

2.6. Case Studies.

It is best while using this text to try out new ideas as they are introduced, to solidify the ideas presented. This is particularly true in the first few chapters, since getting started with the first graphics programs often presents a hurdle. To focus this effort, each chapter ends with some **Case Studies** that describe programming projects that are both interesting in themselves, and concentrate on the ideas developed in the chapter.

[12] ASCII stands for American Standard Code for Information Interchange. Tables of ASCII values are readily available on the internet. Also see `ascii.html` in the web site for this book.

[13] Names for the various "special" keyboard keys, such as the function keys, arrow keys, and "home", may be found in the include file `glut.h`.

Some of the Case Studies are simple exercises that only require fleshing out some pseudocode given in the text, and then running the program through its paces. Others are much more challenging, and could be the basis of a major programming project within a course. It is always difficult to judge how much time someone else will need to accomplish any project. The "**Level of Effort**" that accompanies each Case Study is a rough guess at best.

Level of Effort:
I: a simple exercise. It could be assigned for the next class.
II: an intermediate exercise. It probably needs several days for completion[14].
III: An advanced exercise. It would probably be assigned for two weeks or so ahead.

2.6.1. Case Study 2.1. Pseudo random Clouds of Dots.

(Level of Effort: II) The random number generator (RNG) `random(N)` (see Appendix 3) produces a value between 0 and N-1 each time it is called. It uses the standard C++ function `rand`() to generate values. Each value appears to be randomly selected, and to have no relation to its predecessors.

In fact the successive numbers that `rand`() produces are not generated randomly at all, but rather through a very regular mechanism where each number n_i is determined from its predecessor n_{i-1} by a specific formula. A typical formula is:

$$n_i = (n_{i-1} * A + B) \bmod N \qquad (2.3)$$

where A, B, and N are suitably chosen constants. One set of numbers that works fairly well is: A = 1103515245, B = 12345, and N = 32767. Multiplying n_{i-1} by A and adding B forms a large value, and the modulo operation brings the value into the range 0 to N-1. The process begins with some "seed" value chosen for n_0.

Because the numbers only give an appearance of randomness they are called **pseudo random** numbers. The choices of the values for A, B, and N are very important, and slightly different values give rise to very different characteristics in the sequence of numbers. More details can be found in [knuth, weiss98] .
Scatter Plots.
Some experiments yield data consisting of many pairs of numbers (a_i, b_i), and the goal is to infer visually how the a-values and b-values are "related". For instance, a large number of people are measured, and one wonders if there is a strong correlation between a person's height and weight.

A scatter plot can be used to give visual insight into the data. The data for each person is plotted as a dot at position (*height*, *weight*) so only the `drawDot()` tool is needed. Figure 2.42 shows an example. It suggests that a person's height and weight are roughly linearly related, although some people (such as A) are idiosyncratic, being very tall yet quite light.

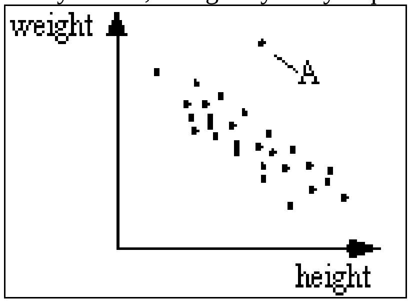

Figure 2.42. A scatter plot of people's height versus weight.

[14] A "day of programming" means several two hour "sessions", with plenty of thinking (and resting) time between sessions. It also assumes a reasonably skilled programmer (with at least two semesters of programming in hand), who is familiar with the idiosyncrasies of the language and the platform being used. It does *not* allow for those dreadful hours we all know too well of being stuck with some obscure bug that presents a brick wall of frustration until it is ferreted out and squashed.

Here we use scatter plots to visually test the quality of a random number generator. Each time the function `random(N)`is called it returns a value in the range 0..*N* - 1 that is apparently chosen at random, unrelated to values previously returned from `random(N).` But are successive values truly unrelated?

One simple test builds a scatter plot based on pairs of successive values returned by `random(N)`. It calls `random(N)` twice in succession, and plots the first value against the second. This can be done using `drawDot`():

```
for(int i = 0; i < num; i++)
   drawDot(random(N), random(N));
```

or in "raw" OpenGL by placing the `for` loop between `glBegin`() and `glEnd`():

```
glBegin(GL_POINTS);
  for(int i = 0; i < num; i++)            // do it num times
     glVertex2i(random(N), random(N));
glEnd();
```

It is more efficient to do it the second way, which avoids the overhead associated with making many calls to `glBegin`() and `glEnd`().

Figure 2.43 shows a typical plot that might result. There should be a “uniform” density of dots throughout the square, to reassure you that the values 0..*N*-1 occur with about equal likelihood, and that there is no discernible dependence between one value and its successor.

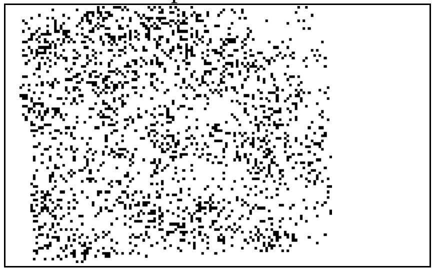
Figure 2.43. A constellation of 500 random dots.

Figure 2.44 shows what can happen with an inferior RNG. In a) there is too high a density of certain values, so the distribution is not uniform in 0..*N*-1. In b) there is high correlation between a number and its successor: when one number is large the other tends to be small. And c) shows perhaps the worst situation of all: after a few dozen values have been generated the pattern *repeats*, and no new dots are generated!

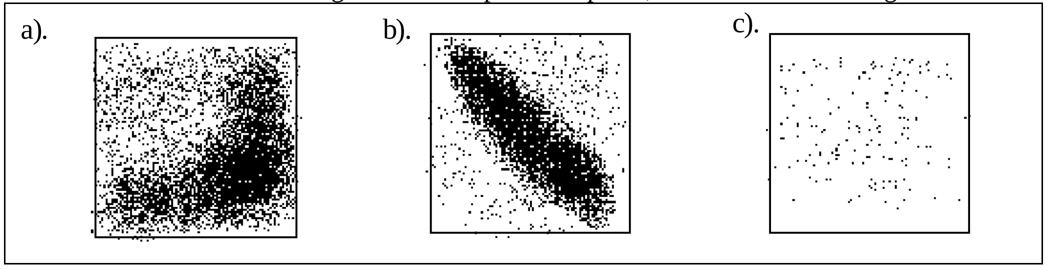

Figure 2.44. Scatter plots for inferior random number generators.

Plotting dot constellations such as these can provide a rough first check on the uniformity of the numbers generated. It is far from a thorough test, however [knuth].

Write a program that produces random dot plots, using some different RNG’s to produce the (*x*, *y*) pairs. Try different constants *A*, *B*, and *N* in the basic RNG and see the effect this has on the dot constellations. (Warning: if the dot constellation suddenly “freezes” such that no new dots appear, it may be that the pattern of numbers is simply repeating.)

2.6.2. Case Study 2.2. Introduction to Iterated Function Systems.

From his paradise no one shall ever evict us

(Level of Effort: II.) The repetitive operation of drawing the Sierpinski gasket is an example of an **iterated function system (IFS)**, which we shall encounter a surprising number of times throughout the book. Many interesting computer-generated figures (fractals, the Mandelbrot set, etc.) are based on variations of it.

A hand calculator provides a tool for experimenting with a simple IFS: Enter some (positive) number *num* and press the square root key. This produces a new number $\sqrt{num}$. Press the square root key again to take its square root, yielding $\sqrt{\sqrt{num}}$. Keep doing this forever..., or until satisfied. We are *iterating* with the square root function, and each result is used as the input for the next square root. An initial value of *num* = 64 yields the sequence: 64, 8, 2.8284, 1.68179,... (Is there a value to which this sequence converges?)

Figure 2.45 presents the system schematically, showing that each output value is *fed back* to have its square root formed, again and again.

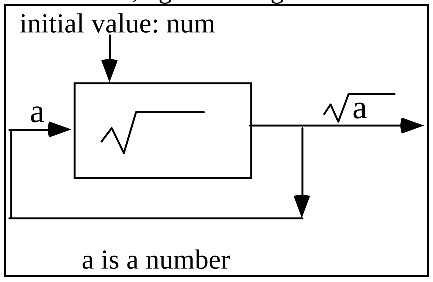

Figure 2.45. Taking the square root repetitively.

In this example the function being iterated is $f(x) = \sqrt{x}$, or symbolically $f(.) = \sqrt{.}$, the "square-rooter". Other functions $f(.)$ can be used instead, such as:

• $f(.) = 2(.)$;	the "doubler" doubles its argument;
• $f(.) = cos(.)$;	the "cosiner";
• $f(.) = 4\,(.)\,(1 - (.))$	the "logistic" function, used in Chaos theory (see Chapter 3)**.**
• $f(.) = (.)^2 + c$ for a constant c;	used to define the Mandelbrot set (see Chapter 8);

It is sometimes helpful to give a name to each number that emerges from the IFS. We call the *k*-th such number d_k, and say that the process begins at $k = 0$ by "injecting" the initial value d_0 into the system. Then the sequence of values generated by the IFS is:

d_0
$d_1 = f(d_0)$
$d_2 = f(f(d_0))$
$d_3 = f(f(f(d_0)))$
...
so d_3 is formed by applying function $f(.)$ three times. This is called **the third iterate** of $f()$ applied to the initial value d_0. More succinctly we can denote the **k-th iterate** of $f()$ by

$$d_k = f^{[k]}(d_0) \tag{2.4}$$

meaning the value produced after $f(.)$ has been applied *k* times to d_0. (Note: it does *not* mean the value $f(d_0)$ is raised to the *k*-th power.) We can also use the recursive form and say:

$d_k = f(d_{k-1})$ for $k = 1,2,3,...,$ for a given value of d_0.

This sequence of values $d_0, d_1, d_2, d_3, d_4,\ldots$ is called "the **orbit** of d_0" for the system.

Example: The orbit of 64 for the function $f(.) = \sqrt{.}$ is 64, 8, 2.8284, 1.68179,..., and the orbit of 10000 is 100, 10, 3.162278, 1.77828,.... (What is the orbit of 0? What is the orbit of 0.1?)

Example: The orbit of 7 for the "doubler" $f(.) = 2\cdot(.)$ is: 7, 14, 28, 56, 112, ... The k-th iterate is $7 * 2^k$.

Example: The orbit of 1 for $f(.) = sin(.)$ can be found using a hand calculator: 1, .8414, .7456, .6784, ... , which *very* slowly approaches the value 0. (What is the orbit of 1 for *cos*(.)? In particular, to what value does the orbit converge?)

Project 1: Plotting the Hailstone sequence.

Consider iterating the intriguing function *f*(.):

$$f(x) = \begin{cases} \frac{x}{2} & \text{if x is even} \\ 3\,x + 1 & \text{if x is odd} \end{cases} \tag{2.5}$$

Even valued arguments are cut in half, whereas odd ones are enlarged. For example, the orbit of 17 is the sequence: 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1 . . .. Once a power of 2 is reached, the sequence falls "like a hailstone" to 1 and becomes trapped in a short repetitive cycle (which one?). An unanswered question in mathematics is:

Unanswered Question: Does *every* orbit fall to 1?

That is, does a positive integer exist, that when used as a starting point and iterated with the hailstone function, does *not* ultimately crash down to 1? No one knows, but the intricacies of the sequence have been widely studied (See [Hayes 1984] or numerous sources on the internet, such as www.cecm.sfu.ca/organics/papers/lagarias/.).

Write a program that plots the course of the sequence $y_k = f^{[k]}(y_0)$ versus k. The user gives a starting value y_0 between 1 and 4,000,000,000. (`unsigned long's` will hold values of this size.) Each value y_k is plotted as the point (k, y_k). Each plot continues until y_k reaches a value of 1 (if it does...).

Because the hailstone sequence can be very long, and the values of y_k can grow very large, it is essential to scale the values before they are displayed. Recall from Section 2.2 that appropriate values of *A*, *B*, *C*, and *D* are determined so that when the value (k, y_k) is plotted at screen coordinates:

$$sx = (A * k + B)$$
$$sy = (C * y_k + D) \tag{2.6}$$

the entire sequence fits on the screen.

Note that you don't know how long the sequence will be, nor how large y_k will get, until after the sequence has been generated. A simple solution is to run the sequence invisibly first, keeping track of the largest value `yBiggest` attained by y_k , as well as the number of iterations `kBiggest` required for the sequence to reach 1. These values are then used to determine *A*, *B*, *C*, and *D*. The sequence is then re-run and plotted.

To improve the final plot:
a. Draw horizontal and vertical axes;
b. Plot the logarithm of y_k rather than y_k itself;

A Curious question: What is the largest `yBiggest` and what is the largest `kBiggest` encountered for any hailstone sequence with starting value between 1 and 1,000,000?

Iterating with functions that produce points.

Iterating numbers through some function *f*(.) is interesting enough, but iterating *points* through a function is even more so, since we can use `drawDot()` to build patterns out of the different points that emerge. So we consider a function $f(p)$ that takes one point $p = (x, y)$ as input and produces another point as its output. Each newly formed point is fed back into the same function again to generate yet another new point, as suggested in Figure 2.46. Here p_{k-1} is used to create the k-th iterate $p_k = f^{[k]}(p_0)$, which is then fed back to produce p_{k+1}, etc.

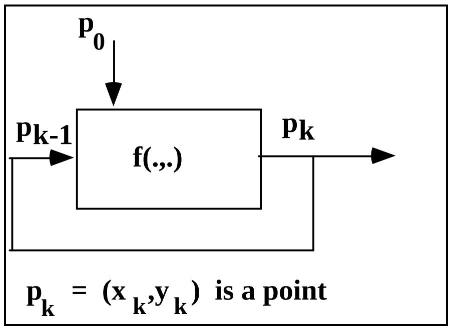

Figure 2.46. Iterated function sequence generator for points.

Once again, we call the sequence of points p_0, p_1, p_2, ... the **orbit of p_0**.

Aside: The Sierpinski gasket seen as an IFS;
In terms of an IFS the *k*-th dot, p_k, of the Sierpinski gasket is formed from p_{k-1} using:

$p_k = (p_{k-1} + T[random(3)]) /2$

where it is understood that the *x* and *y* components must be formed separately. Thus the function that is iterated is:

$f(.) = ((.) + T[random(3)]) / 2$

Project 2: The Gingerbread Man.
The “gingerbread man” shown in Figure 2.47 is based on another IFS, and it can be drawn as a dot constellation. It has become a familiar creature in chaos theory [peitgen88, gleick87, schroeder91] because it is a form of “strange attractor”: the successive dots are “attracted” into a region resembling a gingerbread man, with curious hexagonal holes.

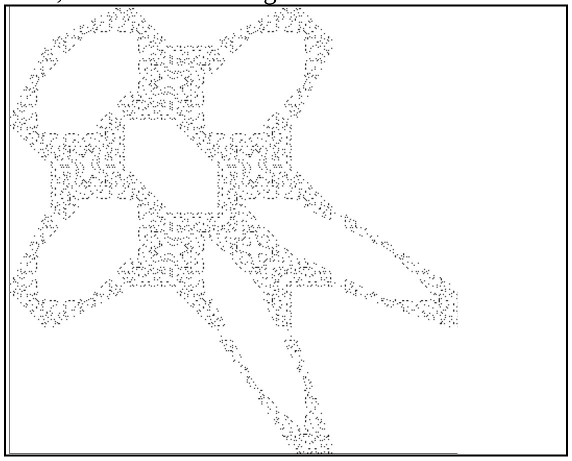
Figure 2.47. A Typical Gingerbread Man.

There is no randomness in the process that generates the gingerbread man; each new point *q* is formed from the previous point *p* according to the rules:

$$q.x = M(1 + 2L) - p.y + | p.x - L M| \qquad (2.7)$$
$$q.y = p.x;$$

where constants *M* and *L* are carefully chosen to scale and position the gingerbread man on the display. (The values *M* = 40 and *L* = 3 might be good choices for a 640 by 480 pixel display.)

Write a program that allows the user to choose the starting point for the iterations with the mouse, and draws the dots for the gingerbread man. (If a mouse is unavailable, one good starting point is (115, 121).) Fix suitable values of *M* and *L* in the routine, but experiment with other values as well.

You will notice that for a given starting point only a certain number of dots appears before the pattern repeats (so it stops changing). Different starting points give rise to different patterns. Arrange your program so that you can add to the picture by inputting additional starting points with the mouse.

Practice Exercise 2.6.1. A Fixed point on the Gingerbread man. Show that this process has "fixed point": ((1+ *L*)*M*, (1 + *L*)*M*). That is, the result of subjecting this point to the process of Equation 2.7 is the same point. (This would be a very uninteresting starting point for generating the gingerbread man!)

2.6.3. Case Study 2.3. The Golden Ratio and Other Jewels.

(Level of Effort: I.) The aspect ratio of a rectangle is an important attribute. Over the centuries, one aspect ratio has been particularly celebrated for its pleasing qualities in works of art: that of the golden rectangle. The golden rectangle is considered as the most pleasing of all rectangles, being neither too narrow nor too squat. It figures in the Greek Parthenon (see Figure 2.48), Leonardo da Vinci's Mona Lisa, Salvador Dali's The Sacrament of the Last Supper, and in much of M. C. Escher's works.

old Fig 3.18 (picture of Greek Parthenon inside golden rectangle)

Figure 2.48. The Greek Parthenon fitting within a Golden Rectangle.

The golden rectangle is based on a fascinating quantity, the golden ratio φ = 1.618033989.. The value φ appears in a surprising number of places in computer graphics.

Figure 2.49 shows a golden rectangle, with sides of length φ and 1. Its shape has the unique property that if a square is removed from the rectangle, the piece that remains will again be a golden rectangle! What value must φ have to make this work? Note in the figure that the smaller rectangle has height 1 and so to be golden must have width 1/φ. Thus

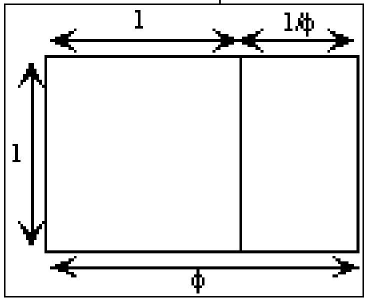

Figure 2.49. The Golden Rectangle.

$$\varphi = 1 + \frac{1}{\varphi} \tag{2.8}$$

which is easily solved to yield

$$\varphi = \frac{1+\sqrt{5}}{2} = 1.618033989... \tag{2.9}$$

This is approximately the aspect ratio of a standard 3-by-5 index card. From Equation 2.8 we see also that if 1 is subtracted from φ the reciprocal of φ is obtained: 1/φ = .618033989. . . . This is the aspect ratio of a golden rectangle lying on its short end.

The number φ is remarkable mathematically in many ways, two favorites being

$$\varphi = \sqrt{1 + \sqrt{1 + \sqrt{1 + \sqrt{1 + ...}}}} \tag{2.10}$$

and

$$\varphi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \ldots}}} \qquad (2.11)$$

These both are easy to prove (how?) and display a pleasing simplicity in the use of the single digit 1.

The idea that the golden rectangle contains a smaller version of itself suggests a form of "infinite regression" of figures…within figures…within figures… *ad infinitum*. Figure 2.50 demonstrates this. Keep removing squares from each remaining golden rectangle.

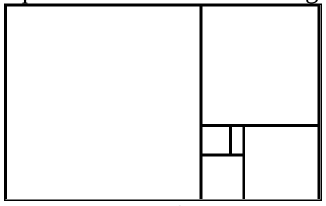
Figure 2.50. Infinite regressions of the golden rectangle.

Write an application that draws the regression of golden rectangles centered in a screen window 600 pixels wide by 400 pixels high. (First determine where and how big the largest golden rectangle is that will fit in this screen window. Your picture should regress down until the smallest rectangle is about one pixel in size.

There is much more to be said about the golden ratio, and many delights can be found in [Gardner 1961], [Hill 1978], [Huntley 1970], and [Ogilvy 1969]. For instance, in the next chapter we see golden pentagrams, and in Chapter 9 we see that two of the platonic solids, the dodecahedron and the icosahedron, contain three mutually perpendicular golden rectangles!

Practice exercises.
2.6.2. Other golden things. Equation 2.10 shows φ as a repeated square root involving the number 1. What is the value of:

$$W = \sqrt{k + \sqrt{k + \sqrt{k + \sqrt{k + \ldots}}}}$$

2.6.3. On φ and Golden Rectangles. a). Show the validity of Equations 2.10 and 2.11.
b). Find the point at which the two dotted diagonals shown in Figure 2.50 lie, and show that this is the point to which the sequence of golden rectangles converges.
c). Use Equation 2.8 to derive the relationship:

$$\varphi^2 + \frac{1}{\varphi^2} = 3 \qquad (2.12)$$

2.6.4. Golden orbits. The expressions in Equations 2.10 and 2.11 show that the golden ratio φ is the limiting value of applying certain functions again and again. The first function is $f(.) = \sqrt{1 + (.)}$. What is the second function? Viewing these expressions in terms of iterated functions systems, φ is seen to be the value to which orbits converge for some starting values. (The starting value is hidden in the "..." of the expressions.) Explore with a hand calculator what starting values one can use and still have the process converge to φ.

2.6.4. Case Study 2.4. Building and Using Polyline Files.

(Level of Effort: II) Complex pictures such as Figure 2.21 are based on a large collection of polylines. The data for the polylines are typically stored in a file, so the picture can be reconstructed at a later time by reading the polylines into a program and redrawing each line. A reasonable format for such a file was described in Section 2.3.1, and the routine `drawPolyLineFile()` was described that does the drawing.

The file, "`dino.dat`", that stores the dinosaur in Figure 2.21 is available as `dino.dat` on the web site for this book (see the preface) Other polyline files are also available there.

a). Write a program that reads polyline data from a file and draws each polyline in turn. Generate at least one interesting polyline file of your own on a text processor, and use your program to draw it.

b). Extend the program in the previous part to accept some other file formats. For instance, have it accept **differentially coded** *x*- and *y*- coordinates. Here the first point (x_1, y_1) of each polyline is encoded as above, but each remaining one (x_i, y_i) is encoded after subtracting the previous point from it: the file contains ($x_i - x_{i-1}$, $y_i - y_{i-1}$). In many cases there are fewer significant digits in the difference than in the original point values, allowing more compact files. Experiment with this format.

c). Adapt the file format above so that a "color value" is associated with each polyline in the file. This color value appears in the file on the same line as the number of points in the associated polyline. Experiment with several polyline files.

---

d). Adjust the polyline drawing routine so that it draws a closed polygon when a minus sign precedes the number of points in a polyline, as in:

```
-3                  <-- negative: so this is a polygon
0        0                      first point in this polygon
35       3                      second point in this polygon
57       8                      also connect this to the first point of the polygon
5                   <-- positive, so leave it open as usual
0        1
12       21
23       34
..  etc
```

The first polyline is drawn as a triangle: its last point is connected to its first..

2.6.5. Case Study 2.5. Stippling of Lines and Polygons.

(Level of Effort: II) You often want a line to be drawn with a dot-dash pattern, or to have a polygon filled with a pattern representing some image. OpenGL provides convenient tools to do this.

Line Stippling.

It is straightforward to define a stipple pattern for use with line drawing. Once the pattern is specified, it is applied to subsequent line drawing as soon as it is enabled with:

```
glEnable(GL_LINE_STIPPLE);
```

and until it is disabled with

```
glDisable(GL_LINE_STIPPLE).
```

The function

```
glLineStipple(GLint factor, GLushort pattern);
```

defines the stipple pattern. The value of `pattern` (which is of type `GLushort` – an unsigned 16-bit quantity) is a sequence of 0's and 1's that define which dots along the line are drawn: a 1 prescribes that a dot is drawn; a 0 prescribes that it is not. The pattern is repeated as many times as necessary to draw the desired line. Figure 2.51 shows several examples. The pattern is given compactly in hexadecimal notation. For example, 0xEECC specifies the bit pattern 1110111011001100. The variable `factor` specifies how much to "enlarge" `pattern`: each bit in the pattern is repeated `factor` times. For instance, the pattern 0xEECC with factor 2 yields: 11111100111111001111000011110000. When drawing a stippled polyline, as with `glBegin(GL_LINE_STRIP); glVertex*(); glVertex*(); glVertex*(); . . . glEnd();`, the pattern continues from the end of one line segment to the beginning of the next, until the `glEnd()`.

pattern	**factor**	**resulting stipple**
0x00FF	1	…….. …….. …….. …….. ……..
0x00FF	2	……………. ……………. …………….
0xAAAA	1	. . . .
0xAAAA	2	.. ..

0xEECC	1	… … .. .. … … .. .. … … .. .. … … .. .. … … .. ..

Figure 2.51. Example stipple patterns.

Write a program that allows the user to type in a `pattern` (in hexadecimal notation) and a value for fac`tor`, and draws stippled lines laid down with the mouse.

Polygon Stippling.
It is also not difficult to define a stipple pattern for filling a polygon: but there are more details to cope with. After the pattern is specified, it is applied to subsequent polygon filling once it is enabled with `glEnable(GL_POLYGON_STIPPLE)`, until disabled with `glDisable(GL_POLYGON_STIPPLE)`.

The function

```
glPolygonStipple(const GLubyte * mask);
```

attaches the stipple pattern to subsequently drawn polygons, based on a 128 byte array `mask[]`. These 128 bytes provide the bits for a bitmask that is 32 bits wide and 32 bits high. The pattern is "tiled" throughout the polygon (which is invoked with the usual `glBegin(GL_POLYGON); glVertex*();. . .; glEnd();`).The pattern is specified by an array definition such as:

```
GLubyte mask[] = {0xff, 0xfe, 0x34, ... };
```

The first four bytes prescribe the 32 bits across the bottom row, from left to right; the next 4 bytes give the next row up, etc. Figure 2.52 shows the result of filling a specific polygon with a "fly" pattern specified in the OpenGL "red book" [woo97].

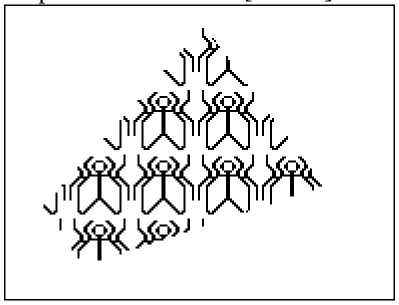

Figure 2.52. An example stippled polygon.

Write a program that defines an interesting stipple pattern for a polygon. It then allows the user to lay down a sequence of convex polygons with the mouse, and each polygon is filled with this pattern.

2.6.6. Case Study 2.6. Polyline Editor.

(Level of Effort: III) Drawing programs often allow one to enter polylines using a mouse, and then to *edit* the polylines until they present the desired picture. Figure 2.53a shows a house in the process of being drawn: the user has just clicked at the position shown, and a line has been drawn from the previous point to the mouse point.

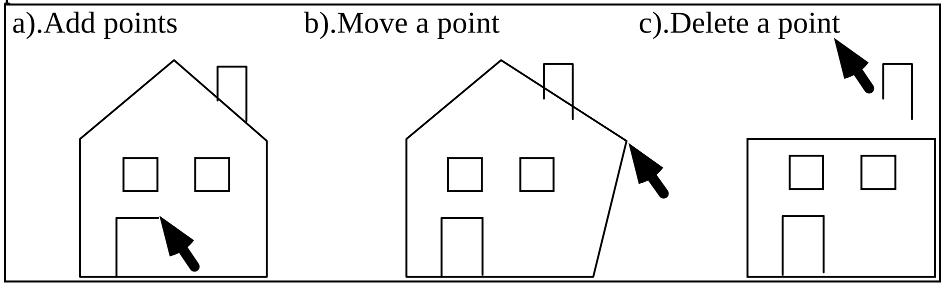

Figure 2.53. Creating and editing polylines.

Figure 2.53b shows the effect of moving a point. The user positions the mouse cursor near some polyline vertex, presses down the mouse button, and "drags" the chosen point to some other location before releasing the mouse button. Upon release, the previous lines connected to this point are erased, and new lines are drawn to it.

Figure 2.53c shows how a point is deleted from a polyline. The user clicks near some polyline vertex, and the two line segments connected to it are erased. Then the two “other” endpoints of the segments just erased are connected with a line segment.

Write and exercise a program that allows the user to enter and edit pictures made up of as many as 60 polylines. The user interacts by pressing keyboard keys and pointing/clicking with the mouse. The functionality of the program should include the “actions”:

• `begin   (‘b’):`	(create a new polyline)
• `delete  (‘d’):`	(delete the next point pointed to)
• `move    (‘m’):`	(drag the point pointed to new location)
• `refresh (‘r’):`	(erase the screen and redraw all the polylines)
• `quit    (‘q’):`	(exit from the program)

A list of polylines can be maintained in an array such as: `GLintPointArray polys[60]`. The verb `begin`, activated by pressing the key ‘`b`’, permits the user to create a new polyline, which is stored in the first available “slot” in array `polys`. The verb `delete` requires that the program identify which point of which polyline lies closest to the current mouse point. Once identified, the “previous” and “next” vertices in the chosen polyline are found. The two line segments connected to the chosen vertex are erased, and the previous and next vertices are joined with a line segment. The verb `move` finds the vertex closest to the current mouse point, and waits for the user to click the mouse a second time, at which point it moves the vertex to this new point.

What other functions might you want in a polyline editor? Discuss how you might save the array of polylines in a file, and read it in later. Also discuss what a reasonable mechanism might be for inserting a new point inside a polyline.

2.6.7. Case Study 2.7. Building and Running Mazes.

(Level of Effort: III) The task of finding a path through a maze seems forever fascinating (see [Ball and Coxeter 1974]). You can generate an elaborate maze on a computer, and use graphics to watch it be traversed. Figure 2.54 shows a rectangular maze having 100 rows and 150 columns. The goal is to find a path from the opening at the left edge to the opening at the right edge. Although you can traverse it manually by trial and error, it's more interesting to develop an algorithm to do it automatically.

use 1st Ed. Figure 3.38

Figure 2.54. A maze.

Write and exercise a program that a). generates and displays a rectangular maze of *R* rows and *C* columns, and b). finds (and displays) the path from start to end. The mazes are generated randomly but must be **proper**; that is, every one of the *R*-by-*C* cells is connected by a unique, albeit tortuous, path to every other cell. Think of a maze as a graph, as suggested by Figure 2.55. A node of the graph corresponds to each cell where either a path terminates or two paths meet, and each path is represented by a branch. So a node occurs at every cell for which there is a choice of which way to go next. For instance, when *Q* is reached there are three choices, whereas at *M* there are only two. The graph of a proper maze is “acyclic” and has a “tree” structure.

use 1st Ed. Figure 3.39

Figure 2.55. A simple maze and its graph.

How should a maze be represented? One way is to state for each cell whether its north wall is intact and its east wall is intact, suggesting the following data structure:

```
char northWall[R][C], eastWall[R][C];
```

If `northwall[i][j]` is 1, the *ij*-th cell has a solid upper wall; otherwise the wall is missing. The 0-th row is a phantom row of cells below the maze whose north walls comprise the bottom edge. Similarly, `eastwall[i][0]` specifies where any gaps appear in the left edge of the maze.

Generating a Maze. Start with all walls intact so that the maze is a simple grid of horizontal and vertical lines. The program draws this grid. An invisible “mouse” whose job is to “eat” through walls to connect adjacent cells,” is initially placed in some arbitrarily chosen cell. The mouse checks the four neighbor cells (above, below, left, and right) and for each asks whether the neighbor has all four walls intact. If not, the cell has previously been visited and so is already on some path. The mouse may detect several candidate cells that haven't been visited: It chooses one randomly and eats through the connecting wall, saving the locations of the other candidates on a stack. The eaten wall is erased, and the mouse repeats the process. When it becomes trapped in a dead end—surrounded by visited cells—it pops an unvisited cell and continues. When the stack is empty, all cells in the maze have been visited. A “start” and “end” cell is then chosen randomly, most likely along some edge of the maze. It is delightful to watch the maze being formed dynamically as the mouse eats through walls. (Question: Might a queue be better than a stack to store candidates? How does this affect the order in which later paths are created?)

Running the Maze. Use a “backtracking” algorithm. At each step, the mouse tries to move in a random direction. If there is no wall, it places its position on a stack and moves to the next cell. The cell that the mouse is in can be drawn with a red dot. When it runs into a dead end, it can change the color of the cell to blue and backtrack by popping the stack. The mouse can even put a wall up to avoid ever trying the dead-end cell again.

Addendum: Proper mazes aren't too challenging because you can always traverse them using the “shoulder-to-the-wall rule.” Here you trace the maze by rubbing your shoulder along the left-hand wall. At a dead end, sweep around and retrace the path, always maintaining contact with the wall. Because the maze is a “tree,” you will ultimately reach your destination. In fact, there can even be cycles in the graph and you still always find the end, as long as both the start and the end cells are on outer boundaries of the maze (why?). To make things more interesting, place the start and end cells in the interior of the maze and also let the mouse eat some extra walls (maybe randomly 1 in 20 times). In this way, some cycles may be formed that encircle the end cell and defeat the shoulder method.

2.9. For Further reading

Several books provide an introduction to using OpenGL. The “OpenGL Programming Guide” by Woo, Neider, and Davis [woo97] is an excellent source. There is also a wealth of information available on the internet. See, for instance, the OpenGL repository at: http://www.opengl.org/, and the complete manual for openGL at http://www.sgi.com/software/opengl/manual.html.