An Introduction to 3D Computer Graphics, Stereoscopic Image, and Animation in OpenGL and C/C++

Fore June

Chapter 2 OpenGL Basics

2.1 What Is OpenGL?

OpenGL, which stands for **Open Graphics Library**, is the computer industry's standard graphics application program interface (API) described in the C programming language. The interface defines over 700 distinct "commands" which are actually C-type functions. Out of the 700 functions, about 650 are in the core OpenGL and 50 in the OpenGL Utility Library. The functions allow users to create interactive two dimensional and three dimensional graphics applications. OpenGL is designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms. We use the words "commands" and "functions" interchangeably in describing OpenGL functions.

Prior to OpenGL, any company developing a graphical application typically had to rewrite the graphics part of it for each operating system platform and had to customize for the specific graphics hardware platform. Nowadays, an application can use OpenGL to create the same graphics look-and-feel in any platform that supports OpenGL.

Each OpenGL command (functon) directs a drawing action or causes special effects. The outputs of an OpenGL command do not depend on the windowing characteristics of any operating system; OpenGL provides special functions for each operating system to enable OpenGL to work in that system's windowing environment. OpenGL comes with a large number of built-in graphics processing functions, including hidden surface removal, alpha blending, antialiasing, texture mapping, pixel operations, viewing and modeling transformations, and creating atmospheric effects.

Silicon Graphics, a company which makes advanced graphics workstations, first started the development of OpenGL. Other companies on the industry-wide Architecture Review Board (ARB) including DEC, Intel, IBM, Microsoft, and Sun Microsystems contribute to its later development. OpenGL is not a development "toolkit". However, such toolkits are available. For example, Silicon Graphics's Open Inventor is an object-oriented programming 3D graphics toolkit for development.

OpenGL is an open-source API; one does not need to pay any royalty to any party when using it. The libraries for various platforms are also free for download. For example, Mesa is an open-source implementation of the OpenGL specification. A variety of device drivers allows Mesa to be used in many different environments ranging from software emulation to complete hardware acceleration for modern graphics processing units (GPUs). One can download the Mesa package from the site *http://www.mesa3d.org/* The site *http://www.opengl.org/* provides documentation, tutorials and examples on OpenGL.

To maintain the hardware-independent characteristics, core OpenGL does not provide commands for performing windowing tasks or obtaining user inputs. One may use the OpenGL Utility Library or other means to perform these tasks. Also, OpenGL does **not** provide high-level commands for three-dimensional modelling, which are commonly used in specifying relatively complicated shapes such as automobiles, parts of the body, airplanes, or molecules. To create complicated objects using OpenGL, one must build up a desired model from a small set of geometric primitive such as points, lines, and polygons. On the other hand, OpenGL Utility Library provides some commands to create some common graphics object models such as spheres, teapots, and cubes.

2.2 OpenGL Programming

2.2.1 Basics of OpenGL Code

The OpenGL functions are contained in two libraries usually called **gl**, and **glu** (or GL and GLU). The first library, gl, is the core OpenGL library; it contains all the required OpenGL functions. The second library is the OpenGL Utility Library (GLU), which contains functions that are written using the core OpenGL functions to provide many helpful functions for modeling. Most OpenGL releases also include the **OpenGL Utility Toolkit** (GLUT), which contains window management functions. Both GLU and GLUT include a number of built-in graphical elements that we can use. To use the functions, we have to include the following header statements in our programs:

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

Names of functions in the GL library begin with **gl** (e.g. $glColor3f()$). On the other hand, names of functions in the GLU library begin with **glu** (e.g. $gluLookAt()$), and those in GLUT begin with **glut** (e.g. $glutMainLoop()$).

The goal of most OpenGL applications is to create images from models using computers, a process known as **rendering**. We construct models, or objects from geometric primitives such as points, lines, and polygons which are defined by one or more vertices. The final rendered image consists of pixels drawn on the screen. **Pixel** is the abbreviation of **picture element** which is the smallest visible element the display hardware can put on the screen. We may organize pixels into bitplanes. A bitplane is a 2D array that holds one bit of information for every pixel of the screen. Therefore, if each pixel holds 24 bits of information, the screen consists of 24 bitplanes. We may then organize bitplanes into a framebuffer, which holds all the information that the graphics display needs to control the color and intensity of all the pixels on the screen. We can imagine that a framebuffer is a 2D array with each location holding the information of a pixel. Bitplanes are artificial and are not necessary when developing graphics programs. However, the concept of framebuffer is fundamental. We can always consider that framebuffer is a memory buffer with each location corresponding to a pixel on the screen; when we change the value at a specified location of the framebuffer, the appearance of the corresponding pixel on the screen will be changed.

OpenGL function calls are enclosed between the functions **glBegin**() and **glEnd**(). For example, the following code draws three points:

```
glBegin( GL_POINTS );
  glVertex2i( 100, 40 );
  glVertex2i( 120, 50 );
  glVertex2i( 140, 80 );
glEnd();
```

Each OpenGL command starts with the prefix **gl** followed by a capital letter just like the command **glVertex2i**(). Also, OpenGL-defined constants begin with **GL_**, consisting of all capital letters. Words in the defined constants are separated by underscores (e.g. GL_POINTS and GL_LINES).

In the above example, the function **glVertex2i**() can be decomposed into several parts as shown in the following table:

glVertex2i(..)

gl	**Vertex**	**2**	**i**
gl library	basic command	number of arguments	type of argument

In this example, the number of arguments is 2, indicating the two dimensional x-y coordinates of the vertex, and the data type of argument is i (integer). If the suffix **i** in the command is replaced by **f**, it would indicate that the argument type is floating-point numbers. Having different formats allows an OpenGL program to accept a user's data in a preferred specified data format.

Some OpenGL commands accept as many as 8 different data types for their arguments. Table 2-1 shows the suffix letters used to specify these data types for ISO C implementations of OpenGL, along with the corresponding OpenGL type definitions. Variations of OpenGL implementations exist and some implementations might not follow this scheme exactly. For example, an implementation of OpenGL in C++ or Ada may differ from this scheme. Therefore, using OpenGL defined types such as **GLbyte** and **GLshort** instead of C-specific types **char** and **short** to write OpenGL programs may ease the difficulties of porting the programs from one platform to another.

Table 2-1 OpenGL Command Suffixes and Argument Data Types

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

Therefore, the two commands

```
glVertex2i(6, 4);
glVertex2f(6.0, 4.0);
```

are equivalent, except that the first function specifies the vertex's coordinates as 32-bit integers, and the second specifies them as single-precision floating-point numbers. The number of arguments of glVertex*() and many OpenGL commands can be 2, 3, or 4; in our notation here, the asterisk * stands for all variations of the command that we can use to specify a vertex.

Some OpenGL functions can take a final letter **v**, which stands for "vector" to indicate that the command takes a pointer (i.e. a vector or an array) rather than a series of individual arguments. The absence of the suffix "v" indicates a scalar format. Many commands have both vector and scalar versions, but some commands accept only scalar formats and others require that at least some of the arguments be specified in vector formats. For example, the following glColor*() commands are equivalent:

```
glColor3f( 1.0, 0.0, 0.0 );

GLfloat color_array[] = {1.0, 0.0, 0.0};
glColor3fv( color_array );
```

2.2.2 The Event Loop

OpenGL programs often run in an event loop. After initialization, the program falls into an infinite loop, waiting for events to happen. The application program has to specify how it handles various events, such as displaying, mouse movement, key pressing and window reshaping. The events are placed in an **event queue** and are processed sequentially. The application only needs to use a set of **callback functions** to specify how the program should react to specific events. Typically, an OpenGL program has a display callback function, which is invoked whenever the OpenGL program discovers that the window needs to be redrawn, including the first time when the window is created and opened. Consequently, if we put our rendering commands in the display callback function, it is guaranteed that they will be executed at least once.

After registering the callback functions, the program typically executes **glutMainLoop**() to get into an infinite loop to wait for events to happen. The following code is a typical piece of code that registers callback functions and puts the program in an infinite loop:

```
glutMouseFunc( myMouse );
glutMotionFunc( myMovedMouse );
glutKeyboardFunc( myKeyboard );
glutDisplayFunc( display );
glutReshapeFunc( reshape );
glutMainLoop();                  //go into perpetual loop
```

The above code has used the following callback functions to invoke various events:

1. **glutMouseFunc**(void (*func)(int button, int state, int x, int y)) associates a mouse button with a routine, which will be called when the mouse button is pressed or released.
2. **glutMotionFunc**(void (*func)(int x, int y)) associates a mouse motion to a routine, which will be called when the mouse is moved while a mouse button is also pressed.
3. **glutKeyboardFunc**(void (*func)(unsigned char key, int x, int y) links a keyboard key with a routine, which will be called when the key is pressed or released.
4. **glutDisplayFunc**(void (*func)(int w, int h)) links window display to a routine, which will be invoked each time the window needs to be redrawn.
5. **glutReshapeFunc**(void (*func)(int w, int h)) links window reshaping to a routine, which will be called when the window is resized.

Besides these functions, another popularly used callback function is **glutIdleFunc**(void (*func)(void)), which is particularly useful in animation graphics. This callback specifies a function that's to be executed when no other events are pending (i.e. when the event loop would otherwise be idle). This routine takes a pointer to the function as its only argument.

2.2.3 OpenGL as a State Machine

We can consider a running OpenGL program as a state machine consisting of state variables such as object color, line width, and object position. We may briefly classify state variables into three kinds:

1. *On-Off state variables* can be set to GL_FALSE (off) or GL_TRUE (on). We can use the functions glEnable() and glDisable() to set or reset these variables. For example, if GL_LINE_SMOOTH is enabled, lines are drawn with antiliasing, which makes lines look smooth, otherwise lines are drawn with aliasing.

2. *Mode state variables* require a function specific to the state variable in order to change its state. For example, glShadeModel (GL_SMOOTH) enables smooth shading, and glShadeModel (GL_FLAT) tells the program to draw with flat shading, which is the default.
3. *Value state variables* require commands specific to the state variables in order to change them. For example, the command "glColor3f(0.0, 1.0, 0.0)" changes the color state to green.

OpenGL also provides functions that can query the current state of the state machine. At any instance, we can use these functions to query the system for any variable's current value. Typically, we use one of these four commands to make the query: **glGetBooleanv**(), **glGetDoublev**(), **glGetFloatv**(), or **glGetIntegerv**(). For example, the following code returns the current color state (RGBA values) to the array *color_state* and prints out the values:

```
glColor3f( 0.6, 0.4, 0.2 );
float color_state[4];
glGetFloatv( GL_CURRENT_COLOR, color_state );
char *colors[4] = { "Red", "Green", "Blue", "Alpha"};
for ( int i = 0; i < 4; i++ )
     printf("%s=%4.1f, ", colors[i], color_state[i]);
```

The code gives the following outputs:

```
Red= 0.6, Green= 0.4, Blue= 0.2, Alpha= 1.0
```

The following is another state-query example which prints out the current version of OpenGL used in the program:

```
const char *version;
version = (const char *) glGetString(GL_VERSION);
printf("\nOpenGL Version: %s\n", version );
```

This code may typically print out the following:

```
OpenGL Version: 2.1 Mesa 7.0.3
```

Besides querying, we can also save the current state in a stack. We can save a collection of state variables on an attribute stack using the command **glPushAttrib**(), and retrieve the attributes later by **glPopAttrib**().

2.2.4 Demo Programs

In this section we present a couple of very simple but complete programs. Each of them can be compiled independently to an executable file. The first program, **draw.cpp** presented in Listing 2-1 shows you how to write an OpenGL program to draw some primitive objects such as points, lines, and rectangles.

Program Listing 2-1: Draw Demo

```
//draw.cpp : demo program for drawing 3 dots,lines,ploylines,rectangles
#include <GL/glut.h>
#include <stdio.h>
```

```
//initialization
void init( void )
{
  glClearColor( 1.0, 1.0, 1.0, 0.0 );   //get white background color
  glColor3f( 0.0f, 0.0f, 0.0f );        //set drawing color to black
  glPointSize( 4.0 );                   //a dot is 4x4 pixels
  glMatrixMode( GL_PROJECTION );        //matrix for projection
  glLoadIdentity();                     //load identity matrix
  gluOrtho2D( 0.0, 300.0, 0.0, 300.0 ); //2D 300x300 world window
}

void display( void )
{
  glClear( GL_COLOR_BUFFER_BIT );       //clear screen

  glBegin( GL_POINTS );                 //draw points
    glVertex2i( 100, 50 );              //draw a point at (100, 50)
    glVertex2i( 100, 150 );             //draw a point
    glVertex2i( 200, 200 ); //draw a point
  glEnd();

  //Two points specify a line.
  glBegin( GL_LINES );                  //draw lines
    glVertex2i( 20, 20 );               //horizontal line
    glVertex2i( 250, 20 );
    glVertex2i( 20, 10 );               //vertical line
    glVertex2i( 20, 250 );
  glEnd();

  //draw line connecting all points
  glBegin( GL_LINE_STRIP );             //draw polyline
    glVertex2i( 100, 100 );
    glVertex2i( 200, 50 );
    glVertex2i( 250, 100 );
    glVertex2i( 100, 50 );
  glEnd();

  glColor3f( 0.6, 0.6, 0.6 );           //bright grey color

  glRecti( 200, 200, 250, 280 );        //draw rectangle

  glFlush();                            //send all output to screen
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGB display mode.
 */
int main(int argc, char** argv)
{
  glutInit(&argc, argv);                //initialize toolkit
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB ); //set display mode
  glutInitWindowSize(300, 300);         //set window size on screen
  glutInitWindowPosition( 100, 150 );   //set window position on screen
  glutCreateWindow(argv[0]);            //open screen widow
  init();                               //customized initialization
  glutDisplayFunc (display);            //points to display function
  glutMainLoop();                       //go into perpetual loop
  return 0;
```

```
}
```

Figure 2-1 shows the display output of the program.

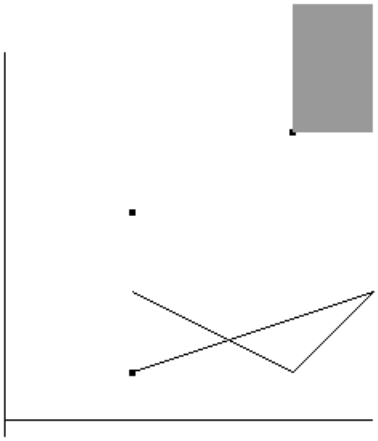

Figure 2-1 Display Output of Program **draw.cpp**.

The following is a sample **Makefile** to compile the program.

```
#sample Makefile for using OpenGL
PROG   = draw
TOP    = /apps/opengl/Mesa-7.0.3
XLIBS  = -lX11 -lXext -lXmu -lXext -lXmu -lXt -lXi -lSM -lICE
LIBS   = -lglut -lGLU -lGL
INCLS  = -I/usr/X11R/include -I$(TOP)/include
LIBDIR = -L/usr/X11/lib -L/usr/X11R6/lib -L$(TOP)/lib

#source codes
SRCS   = $(PROG).cpp

#substitute .cpp by .o to obtain object filenames
OBJS   = $(SRCS:.cpp=.o)

#$< evaluates to the target's dependencies,
#$@ evaluates to the target

$(PROG): $(OBJS)
        g++ -o $@ $(OBJS)  $(LIBDIR) $(LIBS) $(XLIBS)
$(OBJS):
        g++ -c  $*.cpp $(INCLS)
clean:
        rm $(OBJS)
```

2.3 OpenGL Drawing Primitives

OpenGL supports several basic primitive types, including points, lines, quadrilaterals, and general polygons. We can specify a primitive using a sequence of vertices. OpenGL does not have any primitive for drawing curves or circles. However, we can always approximate any curve using a sequence of small line segments. Figure 2-2 below shows some of the OpenGL primitives.

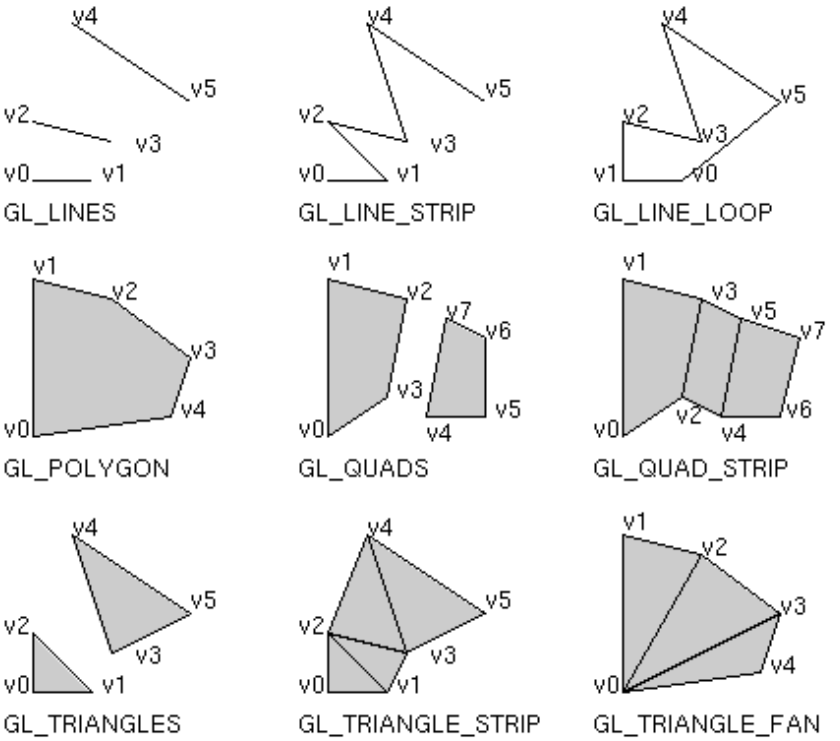

Figure 2-2 OpenGL Drawing Primitives

Table 2-2 describes the meanings of the OpenGL primitives. Usaully there are several ways to draw the same primitive. The way we choose depends on our vertex data, which specify the coordinates of the vertex and are parameters of the **glVertex***() command. We can also use special commands to provide additional vertex-specific data such as a color, a normal vector, texture coordinates, or a combination of these for each vertex. For example, the following piece of code specifies the normal of the vertices.

```
int v[3][3]={{100, 200,0},{200, 100,0},{200, 200, 0}};
int n[3] =  {0, 0, 1};

glBegin ( GL_TRIANGLES );
  glNormal3iv ( n );
  glVertex3iv( v[0] );
```

```
    glVertex3iv( v[1] );
    glVertex3iv( v[2] );
glEnd();
```

Providing the data of a normal vector to a plane is important in generating realistic scenes when light sources exist; OpenGL uses the normal vector to calculate the color at each pixel when filling the polygon.

Table 2-2 OpenGL Primitives Names and Meanings

Value	**Meaning**
GL_POINTS	individual points
GL_LINES	independent line segments, each joining two vertices
GL_LINE_STRIP	connected line segments
GL_LINE_LOOP	connected line segments that form a loop
GL_TRIANGLES	triangles, each specified by 3 vertices
GL_TRIANGLE_STRIP	linked strip of triangles, a vertex may be shared by 2 or more triangles
GL_TRIANGLE_FAN	linked fan of triangles, all having a common vertex
GL_QUADS	four-sided polygons, each joining four vertices
GL_QUAD_STRIP	connected strip of quadrilaterals, shared vertices
GL_POLYGON	simple bounded convex polygon

2.4 Displaying Points, Lines and Polygons

By default, a point is drawn as a single pixel on the screen; a line is drawn solid and one pixel wide, and a polygon is drawn with solidly-filled in color. We can change these default display modes using OpenGL functions as described below.

2.4.1 Point and Line Details

Points

We can change point attributes using the function **void glPointSize**(GLfloat *size*), which works as follows.

1. It sets the width of a rendered point in pixels with *size* > 0.0; the default *size* is 1.0.
2. The default is that antialiasing is disabled. In this case fractional widths are rounded to integer widths. For example, if *size* = 2, the dot is 2×2 pixels and looks like a small square.
3. We can enable antialiasing using the function **glEnable(GL_POINT_SMOOTH)**. In this case, a circular group of pixels is drawn and a point looks like a filled circle.

Line Stipple

We can draw lines with different widths using the function **glLineWidth**(GLFloat *width*), where *width* sets the line width in pixels for the rendered line and its value must be larger than 0.0. The default line width value is 1.0. The actual rendering of lines is affected by the antialiasing mode, in the same way as for points.

We can also draw lines that are stippled in various ways such as dotted, dashed, dotted-dash and so on. To make stippled lines, we use the function **glLineStipple**() to define the stipple pattern, and enable line stippling with glEnable():

void **glLineStipple** (GLint *factor*, GLushort *pattern*) sets the current stippling pattern for lines *pattern* is a 16-bit number that sets the pattern with 1 implying drawing and 0 implying no drawing *factor* ranges 1 to 256 specifying the repeating pattern

For example, the hexadecimal number $0\text{x}3F07$ can be expressed in binary as

$$0011111100000111$$

If we repeat the binary pattern twice, the binary number becomes

$$00001111111111110000000000111111$$

Therefore, the command *glLineStipple(2, 0x3F07)* would draw a line with 6 pixels on, 10 off, 12 on, and 4 off.

2.4.2 Polygon Details

A **polygon** is a plane figure specified by a set of three or more vertices (coordinate positions). A **simple** (standard) polygon is a polygon whose edges do not cross each other. Each polygon in a scene is contained within a plane of infinite extent, which can be described by an equation like the following,

$$Ax + By + Cz + D = 0 \tag{2.1}$$

where A, B, C and D are constants and (x, y, z) is any point on the plane. The plane divides the three dimensional space into three regions; a point can be behind the plane, on the plane, or in front of the plane. For an arbitrary given point (x, y, z), if

$$Ax + By + Cz + D < 0$$

the point (x, y, z) is **behind** the plane. If

$$Ax + By + Cz + D > 0$$

the point (x, y, z) is **before** the plane.

Polygons are typically drawn by coloring all the pixels enclosed within the boundary of the polygon, but we can also draw only the outlines of them or simply draw only the points at the vertices of the polygons. A filled polygon might be solidly filled or stippled with a certain pattern.

Back and Front Faces

A polygon has two sides, **front face** and **back face**. The back face of a polygon is the side that faces into the object interior and the front face is the outward side that is visible to the viewer.

We can render the back face, the front face or both faces of a polygon. We can obtain a cutaway view of an object by appropriately rendering the faces of the polygons of an object. By default, both front and back faces are drawn in the same way. We may use the function **glPolygonMode**() to change this, or to draw only outlines or vertices:

void **glPolygonMode**(GLenum *face*, GLenum *mode*):

- Controls the drawing mode of drawing a polygon.
- Parameter *face* can be GL_FRONT, or GL_BACK, or GL_FRONT_AND_BACK to indicate whether we draw only the front faces, only the back faces or both front faces and back faces of the polygons.
- Parameter *mode* can be GL_POINT, GL_LINE, or GL_FILL to indicate whether we draw a polygon as points, or draw its outline or draw it as a filled polygon. By default, both the front and back faces are drawn filled.

For example, we can draw a polygon with its front face filled and its back face outlined by making the following two calls to this function:

```
glPolygonMode(GL_FRONT, GL_FILL);
glPolygonMode(GL_BACK, GL_LINE);
```

By convention, when we specify the vertices of a polygon in counterclockwise order, the side that appears on the screen is called **front-facing**. . However, we can also make clockwise order as front-facing by the following command:

```
glFrontFace( GL_CW );  //clockwise as front-facing
```

We can change the facing to counterclockwise by

```
glFrontFace( GL_CCW ); //counterclockwise as front-facing
```

The orientation vertices is also known as **winding**.

Culling

Culling is the discarding (ignoring) of invisible polygons during rendering. This feature is helpful in drawing an object that is composed of polygons. For example, we can cull the back faces of an object enclosed by opaque polygons. The feature can be enabled by the command **glEnable**(GL_CULL_FACE). We can also specify whether we want to cull the back face, front face or both back and front faces by the commands,

```
glCullFace ( GL_FRONT );
glCullFace ( GL_BACK );
glCullFace ( GL_FRONT_AND_BACK );
```

As an example, consider the following piece of code. Assuming that everything has been initialized properly, *what do you expect to see on the screen when the code is executed?*

```
glEnable( GL_CULL_FACE );
glCullFace ( GL_BACK );     //discard back faces
glColor3f( 1.0, 0.0, 0.0 ); //red
glFrontFace ( GL_CCW );     //counterclockwise as front facing
glPolygonMode( GL_FRONT, GL_FILL );
glBegin( GL_POLYGON );      //draw solid polygon
  glVertex2i( 0, 0 );
```

```
    glVertex2i( 0, 100 );
    glVertex2i( 100, 100 );
    glVertex2i( 100, 0 );
  glEnd();
```

The answer to the above question is that you should see nothing. This is because the code discards the back face of any polygon and uses counterclockwise as front facing; the polygon it specifies is a square whose vertices are specified in counterclockwise order $((0,0),(0,100),(100,100),(100,0))$ and therefore is a back face and will be discarded while rendering.

The area of an n-sided polygon with vertices $(x_0,y_0),(x_i,y_i),...,(x_{n-1},y_{n-1})$ can be calculated according to the following formula.

$$A = \frac{1}{2}\sum_{i=0}^{n-1} x_i y_{i+1} - x_{i+1} y_i \tag{2.2}$$

where $i+1$ is taken mod n. If we set counterclockwise ordering as front-facing (i.e. GL_CCW), then the side of a polygon with $A > 0$ is a front face, otherwise it is a back face. On the other hand, if we use clockwise ordering as front-facing (i.e. GL_CW), then the side of a polygon with $A < 0$ is a front face, otherwise it is a back face.

Figure 2-3 below shows three polygons, (a), (b), and (c) that are drawn using the following three modes respectively:

(a) glPolygonMode(GL_FRONT, GL_FILL);
(b) glPolygonMode(GL_FRONT, GL_LINE);
(c) glPolygonMode(GL_FRONT, GL_POINT);

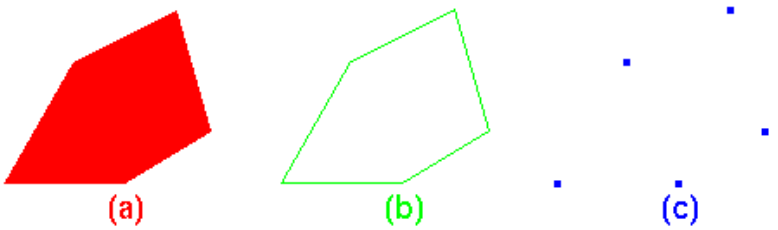

Figure 2-3 Polygons Drawn with Various Modes

2.5 Two Dimensional Graphics

OpenGL was designed primarily for making 3D graphics, but it can be used for 2D drawing as well. In 3D graphics, we need three coordinates, x, y, and z to specify a vertex. In 2D graphics, we just need the two coordinates x and y. The third coordinate, z, usually representing depth is ignored. The depth values (z values) are stored separately in a memory buffer called depth-buffer or z-buffer. The depth values are used to determine which objects are in front of others by depth-test. As we do not need the z coordinate in 2D graphics, it is better for us to turn off the depth-test. This can be done by the command

```
glDisable(GL_DEPTH_TEST);
```

Under this situation, we just need two parameters to specify a vertex. For example,

```
glVertex2f ( 10.0, 30.0 );
glVertex2i ( 100, 200 );
```

are valid commands that specify vertex locations.

The following class **Point2** defines some basic functions of handling a point in 2D graphics.

Listing 2-1: Point2 class

```
class Point2
{
  public:
    float x, y;
    Point2();                       //constructor 1
    Point2(float x0, float y0);     //constructor 2
    void set(float x0, float y0); //set x, y values
    void draw(void);                //draw the point
};
```

Implementation of the Point2 class is straightforward. For example, its **set**() and **draw**() functions can be implemented as follows.

```
void Point2::set(float x0, float y0)
{
  x = x0;  y = y0;
}

void Point2::draw(void)
{
  glBegin(GL_POINTS);   //draw this point
    glVertex2f( (GLfloat)x,  (GLfloat) y );
  glEnd();
}
```

World Window

We can use the command **glutInitWindowSize**() to specify the initial window size. For example, the command

```
glutInitWindowSize (500, 500);
```

sets the initial window size to 500×500 pixels. This window is called the **screen window** and its size is measured in pixels. In the real world, we do not describe the size or coordinates of an object in pixels. For example, the value of the sine function $y = sin(x)$ varies between -1 and 1 and the function is periodic, repeating itself after 2π. Therefore, if we want to display the curve of a sine function, we may want to set the range of x values from 0 to 2π and the range of y values from -1 to 1; the window size for displaying the sine curve is $2\pi \times 2$. This window is referred to as **world window**. We can use the command **gluOrtho2D**(), which defines a 2D orthographic projection matrix, to setup a world window.

For example, the command

```
gluOrtho2D(-3.1416, 3.1416, -1.0, 1.0 );
```

setup the world window for displaying the sine curve from $-\pi$ to $+\pi$.

Even though OpenGL is designed for 3D modeling, at the end each scene is projected on a 2D screen. We shall discuss the projection mechanism in details in next chapter. For 2D drawing, we can use a 2D projection scheme. Using the sine-curve as an example, the following code shows the setup of the projection.

```
const double Xleft=-3,14, Xright=3.14, Ybottom=-1.0, Ytop=1.0;
glMatrixMode (GL_PROJECTION)
glLoadIdentity ();
glOrtho2D(Xleft, Xright, Ybottom, Ytop);
glMatrixMode (GL_MODELVIEW)
```

The following sample code setup the world window and draws the sine curve from $-\pi$ to $+\pi$. The output of the code is shown in Figure 2-4.

```
//Output of Code is shown in Figure 2-4
const double pi = 3.1416;

void init(void)
{
   glClearColor (1.0, 1.0, 1.0, 0.0);
   gluOrtho2D (-pi-0.2, pi+0.2, -1.1, 1.1);
}

void display(void)
{
   glClear (GL_COLOR_BUFFER_BIT);
   glColor3f ( 0.0, 0.0, 0.0 );   //black color
   glLineWidth ( 2 );
   double x, y, incx;
   int N = 100;
   incx = 2 * pi / N;
   //draw the sine curve
   glBegin( GL_LINE_STRIP );
     x = -pi;
     for ( int i = 0; i <= N; i++ ) {
       y = sin ( x );
       glVertex2f ( x, y );
       x += incx;
     }
   glEnd();
   //draw the rectangle
   glBegin( GL_LINE_LOOP );
    glVertex2f(-pi, -1.0);
     glVertex2f( pi, -1.0);
     glVertex2f( pi,  1.0);
     glVertex2f(-pi,  1.0);
   glEnd();
   glFlush ();
}
```

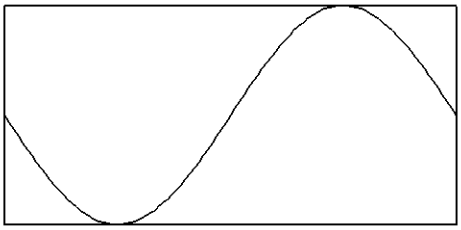

Figure 2-4 Sine Curve in World Window

Viewport

A **viewport** is a rectangular space defined in the screen window. Objects inside the world window are displayed in the viewport. We can display a viewport in anywhere of the screen window. The OpenGL command to set a viewport is **glViewport**():

void glViewport(GLint *x*, GLint *y*, GLint *width*, GLint *height*);

where (*x, y*) specifies the coordinates of the lower-left corner of the viewport, and *width* and *height* are the width and height of the viewport in pixels respectively.

The following sample code displays a sine curve in four different viewports; the output is shown in Figure 2-5.

```
int main(int argc, char** argv)
{
  ....
  glutInitWindowSize (480, 240);
  init();
  glutDisplayFunc(display);
  ....
}

const double pi = 3.1416;
void init(void)
{
   glClearColor (1.0, 1.0, 1.0, 0.0);
   gluOrtho2D (-pi-0.2, pi+0.2, -1.1, 1.1);
}

void drawSine()
{
   double x, y, incx;
   int N = 100;
   incx = 2 * pi / N;
   //draw the sine curve
   glBegin( GL_LINE_STRIP );
     x = -pi;
     for ( int i = 0; i <= N; i++ ) {
       y = sin ( x );
       glVertex2f ( x, y );
       x += incx;
     }
   glEnd();
   //draw the rectangle
```

```
    glBegin( GL_LINE_LOOP );
      glVertex2f(-pi, -1.0);
      glVertex2f( pi, -1.0);
      glVertex2f( pi,  1.0);
      glVertex2f(-pi,  1.0);
    glEnd();
}

void display(void)
{
    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f ( 0.0, 0.0, 0.0 );    //black color
    glLineWidth ( 2 );
    glViewport (0, 0, 240, 120 );   //lower left
    drawSine();
    glViewport (240, 0, 240, 120);  //lower right
    drawSine();
    glViewport (0, 120, 240, 120);  //upper left
    drawSine();
    glViewport (240, 120, 240, 120); //upper right
    drawSine();
    glFlush ();
}
```

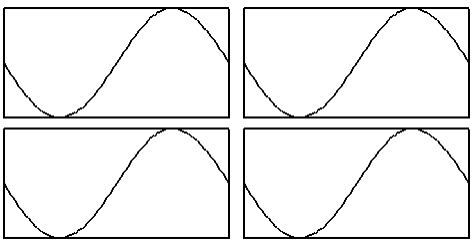

Figure 2-5 Sine Curve Drawn in Four Viewports

One interesting application of 2D graphics is to create turtle graphics, which is a valuable tool for kids to learn graphics, arts, or creating patterns and logos.

2.5.1 Turtle Graphics

Turtle graphics is a style of computer graphics using a relative cursor (the “turtle”) to draw on a 2D plane. It is based on preserved state (position and orientation) and a small number of operations against that state (forward, turn, pen-up and pen-down). Usually the drawing state is referred to as the turtle and programs specify the turtle how to draw. Turtle graphics is easy for kids to pick up and is a key feature of the Logo programming language.

We may imagine that the drawing pen of a plotter is a turtle, and we have control of this little “creature” that exists in a mathematical plane or on a computer display screen. The turtle can respond to a few simple commands: FORWARD moves the turtle in the direction it is facing some number of units. TURN rotates it counterclockwise in its place some number of degrees. The turtle may simply move forward or draws a line while it moves, the exact operation depending on the pen-up and pen-down commands.

If the current position of the "turtle" is given, and we specify the direction (the angle θ with respect to the horizontal axis) and the distance d that we want the turtle to move, then the destination position can be calculated. For example, if the current position of the turtle is (x_0, y_0) and the distance is d as shown in Figure 2-6, we can calculate the destination position according to the following formulas.

$$\begin{aligned} x &= x_0 + d \times \cos\theta \\ y &= y_0 + d \times \sin\theta \end{aligned} \tag{2.3}$$

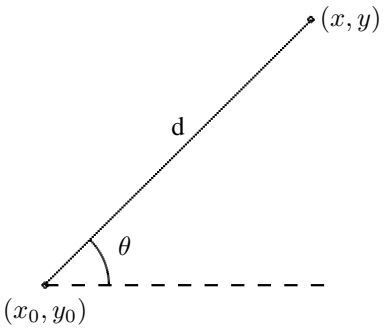

Figure 2-6 Calculating New Position

The following class (Turtle) defines some useful functions of turtle graphics. The **forward**() function is the main function in the class as it instructs the "turtle" to move with "pen" down or up (i.e. *isVisisble* is nonzero or zero).

Listing 2-2: Turtle class

```
class Turtle
{
public:
  Point2 cp;      //current turtle position
  float  angle;   //direction of turtle in degrees
  Turtle();       //constructor
  //move turtle to specified position
  void moveTo(float x, float y);
  void moveTo(Point2 p);
  //move certain displacement
  void moveRel(float dx, float dy);
  //draw line to specified point
  void lineTo(float x, float y);
  void lineTo(Point2 p);
  //turn to direction specified by angle
  void turnTo( float angle );
  //turn to direction pointing from turtle position to p
  void turnTo( Point2 p );
  //turn the specified amount of angle
  void turn( float angle );
  //Move forward by dist. If isVisible != 0, a line is drawn
  //  while moving forward, otherwise no line is drawn.
  void forward ( float dist, int isVisible );
};
```

Implementation of **Turtle** class is also straightforward. For example, we can implement the **turnTo**() and **forward**() functions as follows.

```
//turns to the specific direction
void Turtle::turnTo( float a )
{
  angle = a;   //set current direction
}

//pointing towards p
void Turtle::turnTo( Point2 p )
{
  if ( p.x == cp.x && p.y == cp.y )
    angle = 0;
  else {
    if ( cp.x == p.x ) {
      if ( p.y < cp.y )
        angle = -90;
      else
        angle = 90;
    }else {
        float a=atan((p.y - cp.y) /(p.x - cp.x));//find angle
        angle = 180 * a / 3.1415926;    //convert to degree
    }
  }
}

//move line forward by amount dist;if isVisible nonzero,line is drawn
void Turtle::forward ( float dist, int isVisible )
{
  //radians per degree = 3.14159265389/180
  const float radPerDeg = 0.017453393;
  float x = cp.x + dist * cos ( radPerDeg * angle );
  float y = cp.y + dist * sin ( radPerDeg * angle );
  //special treatment for 0 and 90 degrees for better results
  if ( ( (int) angle % 180 )  == 0 )
    y = cp.y;
  else if ( (int) angle % 90 == 0 )
    x = cp.x;

  if ( isVisible )
    lineTo( x, y );
  else
    moveTo ( x, y );
}
```

Complete listings of all the code discussed in this book can be downloaded from the site *http://www.forejune.com/stereo/*.

The following sample code makes use of turtle graphics to draw a box and then uses the box-drawing routine to draw a window; the output of the code is shown is Figure 2-7.

```
Turtle turtle;

void box()
{
  for ( int i = 0; i < 4; i++ ) {
    turtle.forward( 80, 1);
    turtle.turn( 90 );
  }
```

```
}

void display(void)
{
   int i;
   glClear (GL_COLOR_BUFFER_BIT);
   glLineWidth( 2 );
   Point2 p1 (100, 200);
   turtle.moveTo ( p1 );
   box();    //draw box
   //draw window
   Point2 p2 ( 300, 280 );  //window center
   turtle.moveTo ( p2 );
   for ( int i = 0; i < 4; i++ ){
     box();
     turtle.turn ( 90 );
   }
   glFlush();
}
```

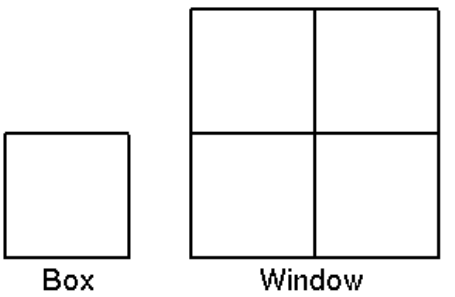

Figure 2-7 Drawing Boxes by Turtle Graphics

Other books by the same author

Windows Fan, Linux Fan

by *Fore June*

Windws Fan, Linux Fan describes a true story about a spiritual battle between a Linux fan and a Windows fan. You can learn from the successful fan to become a successful Internet Service Provider (ISP) and create your own wealth. See *http://www.forejune.com/*

Second Edition, 2002.
ISBN: 0-595-26355-0 Price: $6.86

An Introduction to Digital Video Data Compression in Java

by *Fore June*

The book describes the the principles of digital video data compression techniques and its implementations in java. Topics covered include RBG-YCbCr conversion, macroblocks, DCT and IDCT, integer arithmetic, quantization, reorder, run-level encoding, entropy encoding, motion estimation, motion compensation and hybrid coding. See

http://www.forejune.com/

January 2011

ISBN-10: 1456570870

ISBN-13: 978-1456570873

An Introduction to Video Compression in C/C++

by *Fore June*

The book describes the the principles of digital video data compression techniques and its implementations in C/C++. Topics covered include RBG-YCbCr conversion, macroblocks, DCT and IDCT, integer arithmetic, quantization, reorder, run-level encoding, entropy encoding, motion estimation, motion compensation and hybrid coding.

January 2010
ISBN: 9781451522273