Workbook

Computer Graphics
(SE – 202)
Second Year

Name	
Roll No	
Batch	
Year	
Department	

Department of Computer Science & Information Technology
NED University of Engineering & Technology

Workbook

Computer Graphics
(SE – 202)
Second Year

Prepared by

Noor Afshan Vasty

Approved by

Chairman
Department of Computer Science & Information Technology

Department of Computer Science & Information Technology
NED University of Engineering & Technology

CONTENTS

Lab Session 01

OBJECT

Introduction to OpenGL, OpenGL Command Syntax

Computer Graphics involve technology to accept, process, transform and present information in a visual form that also concerns with producing images (or animations) using a computer.

OpenGL is a low-level graphics library specification. It makes available to the programmer a small set of geomteric primitives - points, lines, polygons, images, and bitmaps. OpenGL provides a set of commands that allow the specification of geometric objects in two or three dimensions, using the provided primatives, together with commands that control how these objects are rendered (drawn).

OpenGL Command Syntax

OpenGL commands use the prefix **gl** and initial capital letters for each word making up the command name (**glClearColor()**, for example). Similarly, OpenGL defined constants begin with GL_, use all capital letters, and use underscores to separate words (like GL_COLOR_BUFFER_BIT).

Data Types

Note the following commands
glColor3f() and glVertex3f()

In these commands, 3 shows the number of arguments the command contains, and the f part of the suffix indicates that the arguments are floating-point numbers.

Thus, the two commands **glVertex2i(1, 3); glVertex2f(1.0, 3.0);** are equivalent, except that the first specifies the vertex's coordinates as 32-bit integers, and the second specifies them as single-precision floating-point numbers.

Some OpenGL commands accept as many as 8 different data types for their arguments. The letters used as suffixes to specify these data types for ISO C implementations of OpenGL are shown in Table 1-1, along with the corresponding OpenGL type definitions. The particular implementation of OpenGL that you're using might not follow this scheme exactly; an implementation in C++ or Ada, for example, wouldn't need to.

Table 1-1 : Command Suffixes and Argument Data Types

Suffix	Data Type	Typical Corresponding C-Language Type	OpenGL Type Definition
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 integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int or unsigned long	GLuint, GLenum, GLbitfield

OpenGL-Related Libraries

OpenGL provides a powerful but primitive set of rendering commands, and all higher-level drawing must be done in terms of these commands. Also, OpenGL programs have to use the underlying mechanisms of the windowing system. A number of libraries exist to allow you to simplify your programming tasks, including the following:

- The OpenGL Utility Library (GLU) contains several routines that use lower-level OpenGL commands to perform such tasks as setting up matrices for specific viewing orientations and projections, performing polygon tessellation, and rendering surfaces. This library is provided as part of every OpenGL implementation. GLU routines use the prefix **glu**.
- For every window system, there is a library that extends the functionality of that window system to support OpenGL rendering. For machines that use the X Window System, the OpenGL Extension to the X Window System (GLX) is provided as an adjunct to OpenGL. GLX routines use the prefix **glX**. For

Microsoft Windows, the WGL routines provide the Windows to OpenGL interface. All WGL routines use the prefix **wgl**.

- The OpenGL Utility Toolkit (GLUT) is a window system-independent toolkit, written by Mark Kilgard, to hide the complexities of differing window system APIs. GLUT routines use the prefix **glut.**

Include Files

For all OpenGL applications, you want to include the gl.h header file in every file. Almost all OpenGL applications use GLU, the aforementioned OpenGL Utility Library, which requires inclusion of the glu.h header file. So almost every OpenGL source file begins with

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

If you are directly accessing a window interface library to support OpenGL, such as GLX, AGL, PGL, or WGL, you must include additional header files. For example, if you are calling GLX, you may need to add these lines to your code

```
#include <X11/Xlib.h>
#include <GL/glx.h>
```

If you are using GLUT for managing your window manager tasks, you should include

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

Note that glut.h includes gl.h, glu.h, and glx.h automatically, so including all three files is redundant. GLUT for Microsoft Windows includes the appropriate header file to access WGL.

GLUT, the OpenGL Utility Toolkit

As you know, OpenGL contains rendering commands but is designed to be independent of any window system or operating system. Consequently, it contains no commands for opening windows or reading events from the keyboard or mouse. Unfortunately, it's impossible to write a complete graphics program without at least opening a window, and most interesting programs require a bit of user input or other services from the operating system or window system. In many cases, complete programs make the most interesting examples, so this book uses GLUT to simplify opening windows, detecting input, and so on. If you have an implementation of OpenGL and GLUT on your system, the examples in this book should run without change when linked with them.

In addition, since OpenGL drawing commands are limited to those that generate simple geometric primitives (points, lines, and polygons), GLUT includes several routines that create more complicated three-dimensional objects such as a sphere, a torus, and a teapot. This way, snapshots of program output can be interesting to look at. (Note that the

OpenGL Utility Library, GLU, also has quadrics routines that create some of the same three-dimensional objects as GLUT, such as a sphere, cylinder, or cone.)

Exercise

Answer the following Questions.

1. What syntax does openGL follow for its commands.
2. How the type of data is set in OpenGL.
3. Describe OpenGL-Related Libraries.

Lab Session 02

OBJECT

Specifying Colors for Objects /Basic OpenGL Geometric Primitive Types

Specifying a Color

With OpenGL, the description of the shape of an object being drawn is independent of the description of its color. Whenever a particular geometric object is drawn, it's drawn using the currently specified coloring scheme. The coloring scheme might be as simple as "draw everything in fire-engine red," or might be as complicated as "assume the object is made out of blue plastic, that there's a yellow spotlight pointed in such and such a direction, and that there's a general low-level reddish-brown light everywhere else." In general, an OpenGL programmer first sets the color or coloring scheme and then draws the objects. Until the color or coloring scheme is changed, all objects are drawn in that color or using that coloring scheme. This method helps OpenGL achieve higher drawing performance than would result if it didn't keep track of the current color.

For example, the pseudocode

```
set_current_color(red);
draw_object(A);
draw_object(B);
set_current_color(green);
set_current_color(blue);
draw_object(C);
```

draws objects A and B in red, and object C in blue. The command on the fourth line that sets the current color to green is wasted.

To set a color, use the command **glColor3f()**. It takes three parameters, all of which are floating-point numbers between 0.0 and 1.0. The parameters are, in order, the red, green, and blue *components* of the color. You can think of these three values as specifying a "mix" of colors: 0.0 means don't use any of that component, and 1.0 means use all you can of that component. Thus, the code

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

makes the brightest red the system can draw, with no green or blue components. All zeros makes black; in contrast, all ones makes white. Setting all three components to 0.5 yields gray (halfway between black and white). Here are eight commands and the colors they would set.

```
glColor3f(0.0, 0.0, 0.0);          black
glColor3f(1.0, 0.0, 0.0);           red
glColor3f(0.0, 1.0, 0.0);           green
glColor3f(1.0, 1.0, 0.0);           yellow
glColor3f(0.0, 0.0, 1.0);           blue
glColor3f(1.0, 0.0, 1.0);           magenta
```

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

You might have noticed earlier that the routine to set the clearing color, **glClearColor()**, takes four parameters, the first three of which match the parameters for **glColor3f()**. The fourth parameter is the alpha value.

Describing Points, Lines, and Polygons

This section explains how to describe OpenGL geometric primitives. All geometric primitives are eventually described in terms of their *vertices* - coordinates that define the points themselves, the endpoints of line segments, or the corners of polygons. The next section discusses how these primitives are displayed and what control you have over their display.

What Are Points, Lines, and Polygons?

You probably have a fairly good idea of what a mathematician means by the terms *point*, line, and polygon. The OpenGL meanings are similar, but not quite the same.

One difference comes from the limitations of computer-based calculations. In any OpenGL implementation, floating-point calculations are of finite precision, and they have round-off errors. Consequently, the coordinates of OpenGL points, lines, and polygons suffer from the same problems.

Points

A point is represented by a set of floating-point numbers called a vertex. All internal calculations are done as if vertices are three-dimensional. Vertices specified by the user as two-dimensional (that is, with only *x* and *y* coordinates) are assigned a *z* coordinate equal to zero by OpenGL.

Lines

In OpenGL, the term *line* refers to a *line segment*, not the mathematician's version that extends to infinity in both directions. There are easy ways to specify a connected series of line segments, or even a closed, connected series of segments (see Figure 2.1). In all cases, though, the lines constituting the connected series are specified in terms of the vertices at their endpoints.

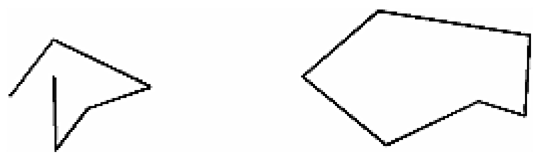


Figure 2.1 : Two Connected Series of Line Segments

Polygons

Polygons are the areas enclosed by single closed loops of line segments, where the line segments are specified by the vertices at their endpoints. Polygons are typically drawn with the pixels in the interior filled in, but you can also draw them as outlines or a set of points.

In general, polygons can be complicated, so OpenGL makes some strong restrictions on what constitutes a primitive polygon. First, the edges of OpenGL polygons can't intersect (a mathematician would call a polygon satisfying this condition a *simple polygon*). Second, OpenGL polygons must be *convex*, meaning that they cannot have indentations. Stated precisely, a region is convex if, given any two points in the interior, the line segment joining them is also in the interior. See Figure 2.2 for some examples of valid and invalid polygons. OpenGL, however, doesn't restrict the number of line segments making up the boundary of a convex polygon. Note that polygons with holes can't be described. They are nonconvex, and they can't be drawn with a boundary made up of a single closed loop. Be aware that if you present OpenGL with a nonconvex filled polygon, it might not draw it as you expect. For instance, on most systems no more than the convex hull of the polygon would be filled. On some systems, less than the convex hull might be filled.

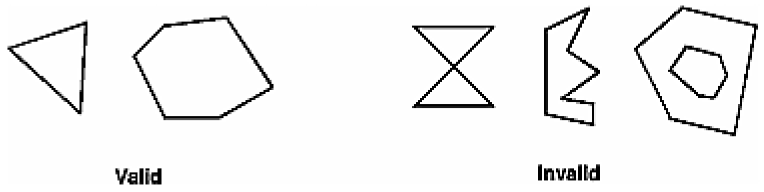


Figure 2.2 : Valid and Invalid Polygons

The reason for the OpenGL restrictions on valid polygon types is that it's simpler to provide fast polygon-rendering hardware for that restricted class of polygons. Simple polygons can be rendered quickly. The difficult cases are hard to detect quickly. So for maximum performance, OpenGL crosses its fingers and assumes the polygons are simple.

Many real-world surfaces consist of nonsimple polygons, nonconvex polygons, or polygons with holes. Since all such polygons can be formed from unions of simple convex polygons, some routines to build more complex objects are provided in the GLU library. These routines take complex descriptions and tessellate them, or break them down into groups of the simpler OpenGL polygons that can then be rendered

Since OpenGL vertices are always three-dimensional, the points forming the boundary of a particular polygon don't necessarily lie on the same plane in space. (Of course, they do in many cases - if all the *z* coordinates are zero, for example, or if the polygon is a triangle.) If a polygon's vertices don't lie in the same plane, then after various rotations in space, changes in the viewpoint, and projection onto the display screen, the points might no longer form a simple convex polygon. For example, imagine a four-point *quadrilateral* where the points are slightly out of plane, and look at it almost edge-on. You can get a nonsimple polygon that resembles a bow tie, as shown in Figure 2.3, which isn't guaranteed to be rendered correctly. This situation isn't all that unusual if you approximate curved surfaces by quadrilaterals made of points lying on the true surface. You can always avoid the problem by using triangles, since any three points always lie on a plane.

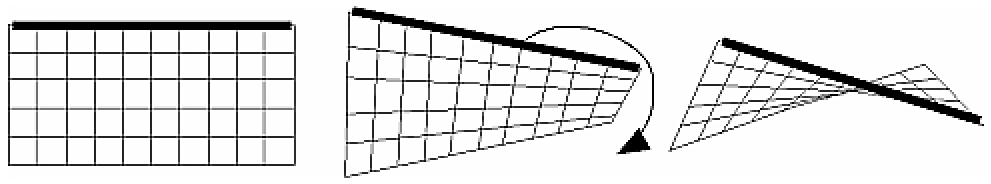


Figure 2.3 : Nonplanar Polygon Transformed to Nonsimple Polygon

Rectangles

Since rectangles are so common in graphics applications, OpenGL provides a filled-rectangle drawing primitive, **glRect*()**. You can draw a rectangle as a polygon, but your particular implementation of OpenGL might have optimized **glRect*()** for rectangles.

void ***glRect****{sifd}(TYPEx1, TYPEy1, TYPEx2, TYPEy2);*
void ***glRect****{sifd}****v****(TYPE*v1, TYPE*v2);*

> *Draws the rectangle defined by the corner points (x1, y1) and (x2, y2). The rectangle lies in the plane z=0 and has sides parallel to the x- and y-axes. If the vector form of the function is used, the corners are given by two pointers to arrays, each of which contains an (x, y) pair.*

Note that although the rectangle begins with a particular orientation in three-dimensional space (in the *x-y* plane and parallel to the axes), you can change this by applying rotations or other transformations.

Curves and Curved Surfaces

Any smoothly curved line or surface can be approximated - to any arbitrary degree of accuracy - by short line segments or small polygonal regions. Thus, subdividing curved lines and surfaces sufficiently and then approximating them with straight line segments or flat polygons makes them appear curved (see Figure 2.4). If you're skeptical that this really works, imagine subdividing until each line segment or polygon is so tiny that it's smaller than a pixel on the screen.

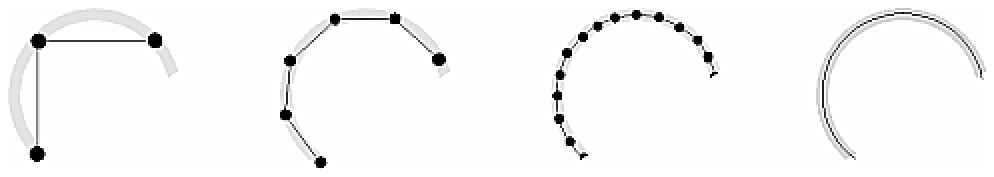


Figure 2.4 : Approximating Curves

Even though curves aren't geometric primitives, OpenGL does provide some direct support for subdividing and drawing them.

Specifying Vertices

With OpenGL, all geometric objects are ultimately described as an ordered set of vertices. You use the **glVertex*()** command to specify a vertex.

void ***glVertex****{234}{sifd}[v](TYPEcoords);*

> *Specifies a vertex for use in describing a geometric object. You can supply up to four coordinates (x, y, z, w) for a particular vertex or as few as two (x, y) by selecting the appropriate version of the command. If you use a version that doesn't explicitly specify z or w, z is understood to be 0 and w is understood to be 1. Calls to* ***glVertex*()*** *are only effective between a* ***glBegin()*** *and* ***glEnd()*** *pair.*

Example 2.1 provides some examples of using **glVertex*()**.

Example 2.1 : Legal Uses of glVertex*()

```
glVertex2s(2, 3);
glVertex3d(0.0, 0.0, 3.1415926535898);
glVertex4f(2.3, 1.0, -2.2, 2.0);
GLdouble dvect[3] = {5.0, 9.0, 1992.0};
glVertex3dv(dvect);
```

The first example represents a vertex with three-dimensional coordinates (2, 3, 0). (Remember that if it isn't specified, the *z* coordinate is understood to be 0.) The coordinates in the second example are (0.0, 0.0, 3.1415926535898) (double-precision floating-point numbers). The third example represents the vertex with three-dimensional coordinates (1.15, 0.5, -1.1). (Remember that the *x, y*, and *z* coordinates are eventually divided by the *w* coordinate.) In the final example, *dvect* is a pointer to an array of three double-precision floating-point numbers.

On some machines, the vector form of **glVertex*()** is more efficient, since only a single parameter needs to be passed to the graphics subsystem. Special hardware might be able to send a whole series of coordinates in a single batch. If your machine is like this, it's to your advantage to arrange your data so that the vertex coordinates are packed sequentially in memory. In this case, there may be some gain in performance by using the vertex array operations of OpenGL.

OpenGL Geometric Drawing Primitives

Now that you've seen how to specify vertices, you still need to know how to tell OpenGL to create a set of points, a line, or a polygon from those vertices. To do this, you bracket each set of vertices between a call to **glBegin()** and a call to **glEnd()**. The argument passed to **glBegin()** determines what sort of geometric primitive is constructed from the vertices. For example, Example 2.2> specifies the vertices for the polygon shown in Figure 2-6.

Example 2.2 : Filled Polygon

```
glBegin(GL_POLYGON);
   glVertex2f(0.0, 0.0);
   glVertex2f(0.0, 3.0);
   glVertex2f(4.0, 3.0);
   glVertex2f(6.0, 1.5);
   glVertex2f(4.0, 0.0);
glEnd();
```

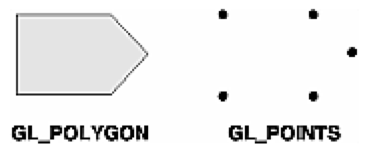


Figure 2.5 : Drawing a Polygon or a Set of Points

If you had used GL_POINTS instead of GL_POLYGON, the primitive would have been simply the five points shown in Figure 2.5. Table 2.1 in the following function summary for **glBegin()** lists the ten possible arguments and the corresponding type of primitive.

*void **glBegin**(GLenum mode);*

Marks the beginning of a vertex-data list that describes a geometric primitive. The type of primitive is indicated by mode, which can be any of the values shown in Table 2.1.

Table 2.1 : Geometric Primitive Names and Meanings

Value	**Meaning**
GL_POINTS	individual points
GL_LINES	pairs of vertices interpreted as individual line segments
GL_LINE_STRIP	series of connected line segments
GL_LINE_LOOP	same as above, with a segment added between last and first vertices

GL_TRIANGLES	triples of vertices interpreted as triangles
GL_TRIANGLE_STRIP	linked strip of triangles
GL_TRIANGLE_FAN	linked fan of triangles
GL_QUADS	quadruples of vertices interpreted as four-sided polygons
GL_QUAD_STRIP	linked strip of quadrilaterals
GL_POLYGON	boundary of a simple, convex polygon

void ***glEnd****(void);*

Marks the end of a vertex-data list.

Figure 2.6 shows examples of all the geometric primitives listed in Table 2.1. The paragraphs that follow the figure describe the pixels that are drawn for each of the objects. Note that in addition to points, several types of lines and polygons are defined. Obviously, you can find many ways to draw the same primitive. The method you choose depends on your vertex data.

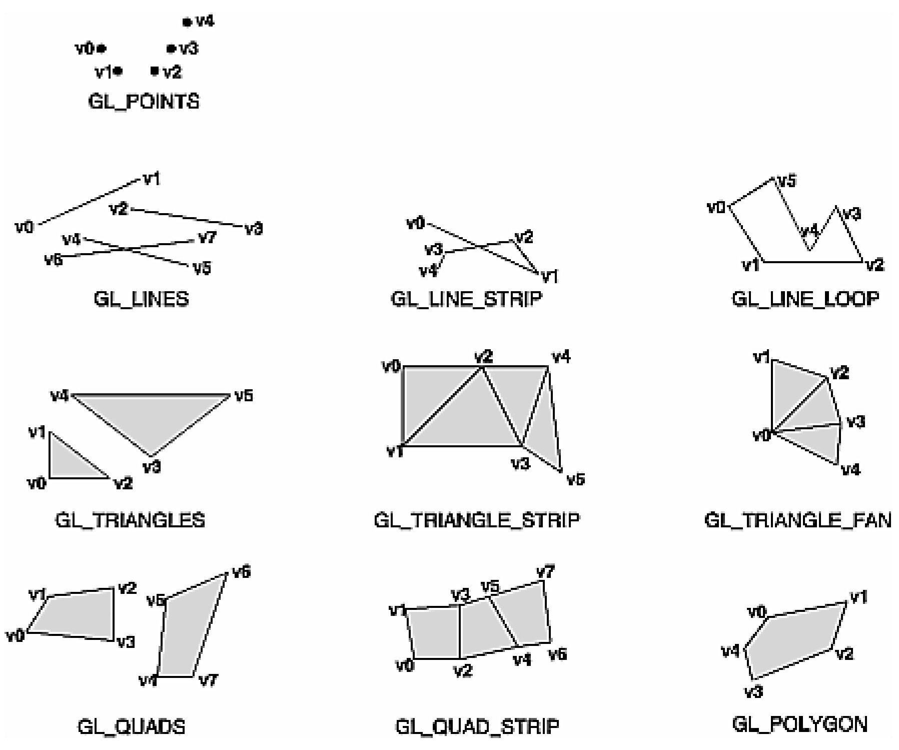


Figure 2.6 : Geometric Primitive Types

As you read the following descriptions, assume that *n* vertices (v0, v1, v2, ... , vn-1) are described between a **glBegin()** and **glEnd()** pair.

GL_POINTS	Draws a point at each of the *n* vertices.
GL_LINES	Draws a series of unconnected line segments. Segments are drawn between v0 and v1, between v2 and v3, and so on. If *n* is odd, the last segment is drawn between vn-3 and vn-2, and vn-1 is ignored.
GL_LINE_STRIP	Draws a line segment from v0 to v1, then from v1 to v2, and so on, finally drawing the segment from vn-2 to vn-1. Thus, a total of *n-1* line segments are drawn. Nothing is drawn unless *n* is larger than 1. There are no restrictions on the vertices describing a line strip (or a line loop); the lines can intersect arbitrarily.
GL_LINE_LOOP	Same as GL_LINE_STRIP, except that a final line segment is drawn from vn-1 to v0, completing a loop.
GL_TRIANGLES	Draws a series of triangles (three-sided polygons) using vertices v0, v1, v2, then v3, v4, v5, and so on. If *n* isn't an exact multiple of 3, the final one or two vertices are ignored.
GL_TRIANGLE_STRIP	Draws a series of triangles (three-sided polygons) using vertices v0, v1, v2, then v2, v1, v3 (note the order), then v2, v3, v4, and so on. The ordering is to ensure that the triangles are all drawn with the same orientation so that the strip can correctly form part of a surface. Preserving the orientation is important for some operations, such as culling.
GL_TRIANGLE_FAN	Same as GL_TRIANGLE_STRIP, except that the vertices are v0, v1, v2, then v0, v2, v3, then v0, v3, v4, and so on (see Figure 2-7).
GL_QUADS	Draws a series of quadrilaterals (four-sided polygons) using vertices v0, v1, v2, v3, then v4, v5, v6, v7, and so on. If *n* isn't a multiple of 4, the final one, two, or three vertices are ignored.
GL_QUAD_STRIP	Draws a series of quadrilaterals (four-sided polygons) beginning with v0, v1, v3, v2, then v2, v3, v5, v4, then v4, v5, v7, v6, and so on (see Figure 2-7). *n* must be at least 4 before anything is drawn. If *n* is odd, the final vertex is ignored.
GL_POLYGON	Draws a polygon using the points v0, ... , vn-1 as vertices. *n* must be at least 3, or nothing is drawn. In addition, the polygon

	specified must not intersect itself and must be convex. If the vertices don't satisfy these conditions, the results are unpredictable.

Restrictions on Using glBegin() and glEnd()

The most important information about vertices is their coordinates, which are specified by the **glVertex*()** command. You can also supply additional vertex-specific data for each vertex - a color, a normal vector, texture coordinates, or any combination of these - using special commands. In addition, a few other commands are valid between a **glBegin()** and **glEnd()** pair. Table 2-3 contains a complete list of such valid commands.

Table 2.2 : Valid Commands between glBegin() and glEnd()

Command	**Purpose of Command**	
glVertex*()	set vertex coordinates	
glColor*()	set current color	
glIndex*()	set current color index	
glNormal*()	set normal vector coordinates	
glTexCoord*()	set texture coordinates	
glEdgeFlag*()	control drawing of edges	
glMaterial*()	set material properties	
glArrayElement()	extract vertex array data	
glEvalCoord*(), glEvalPoint*()	generate coordinates	
glCallList(), glCallLists()	execute display list(s)	

No other OpenGL commands are valid between a **glBegin()** and **glEnd()** pair, and making most other OpenGL calls generates an error. Some vertex array commands, such as **glEnableClientState()** and **glVertexPointer()**, when called between **glBegin()** and **glEnd()**, have undefined behavior but do not necessarily generate an error. (Also, routines related to OpenGL, such as **glX*()** routines have undefined behavior between **glBegin()** and **glEnd()**.) These cases should be avoided, and debugging them may be more difficult.

Note, however, that only OpenGL commands are restricted; you can certainly include other programming-language constructs (except for calls, such as the aforementioned **glX*()** routines). For example, Example 2.3 draws an outlined circle.

Example 2.3 : Other Constructs between glBegin() and glEnd()

```
#define PI 3.1415926535898
GLint circle_points = 100;
glBegin(GL_LINE_LOOP);
for (i = 0; i < circle_points; i++) {
   angle = 2*PI*i/circle_points;
   glVertex2f(cos(angle), sin(angle));
}
glEnd();
```

Note: This example isn't the most efficient way to draw a circle, especially if you intend to do it repeatedly. The graphics commands used are typically very fast, but this code calculates an angle and calls the **sin()** and **cos()** routines for each vertex; in addition, there's the loop overhead. If you need to draw lots of circles, calculate the coordinates of the vertices once and save them in an array and create a display list, or use vertex arrays to render them.

Unless they are being compiled into a display list, all **glVertex*()** commands should appear between some **glBegin()** and **glEnd()** combination. (If they appear elsewhere, they don't accomplish anything.) If they appear in a display list, they are executed only if they appear between a **glBegin()** and a **glEnd()**.

Although many commands are allowed between **glBegin()** and **glEnd()**, vertices are generated only when a **glVertex*()** command is issued. At the moment **glVertex*()** is called, OpenGL assigns the resulting vertex the current color, texture coordinates, normal vector information, and so on. To see this, look at the following code sequence. The first point is drawn in red, and the second and third ones in blue, despite the extra color commands.

```
glBegin(GL_POINTS);
   glColor3f(0.0, 1.0, 0.0);                /* green */
   glColor3f(1.0, 0.0, 0.0);                /* red */
   glVertex(...);
   glColor3f(1.0, 1.0, 0.0);                /* yellow */
   glColor3f(0.0, 0.0, 1.0);                /* blue */
   glVertex(...);
   glVertex(...);
glEnd();
```

You can use any combination of the 24 versions of the **glVertex*()** command between **glBegin()** and **glEnd()**, although in real applications all the calls in any particular instance tend to be of the same form. If your vertex-data specification is consistent and repetitive (for example, **glColor***, **glVertex***, **glColor***, **glVertex***,...), you may enhance your program's performance by using vertex arrays.

Exercise:

Write code to draw a point on output window. Specify blue as its color.

Lab Session 03

OBJECT

Window Management/ My first Program

Five routines perform tasks necessary to initialize a window.

- **glutInit**(int **argc*, char ***argv*) initializes GLUT and processes any command line arguments (for X, this would be options like -display and -geometry). **glutInit()** should be called before any other GLUT routine.
- **glutInitDisplayMode**(unsigned int *mode*) specifies whether to use an *RGBA* or color-index color model. You can also specify whether you want a single- or double-buffered window. (If you're working in color-index mode, you'll want to load certain colors into the color map; use **glutSetColor()** to do this.) Finally, you can use this routine to indicate that you want the window to have an associated depth, stencil, and/or accumulation buffer. For example, if you want a window with double buffering, the RGBA color model, and a depth buffer, you might call **glutInitDisplayMode**(*GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH*).
- **glutInitWindowPosition**(int *x*, int *y*) specifies the screen location for the upper-left corner of your window.
- **glutInitWindowSize**(int *width*, int *size*) specifies the size, in pixels, of your window.
- int **glutCreateWindow**(char **string*) creates a window with an OpenGL context. It returns a unique identifier for the new window. Be warned: Until **glutMainLoop()** is called (see next section), the window is not yet displayed.

The Display Callback

glutDisplayFunc(void (**func*)(void)) is the first and most important event callback function you will see. Whenever GLUT determines the contents of the window need to be redisplayed, the callback function registered by **glutDisplayFunc()** is executed. Therefore, you should put all the routines you need to redraw the scene in the display callback function.

If your program changes the contents of the window, sometimes you will have to call **glutPostRedisplay**(void), which gives **glutMainLoop()** a nudge to call the registered display callback at its next opportunity.

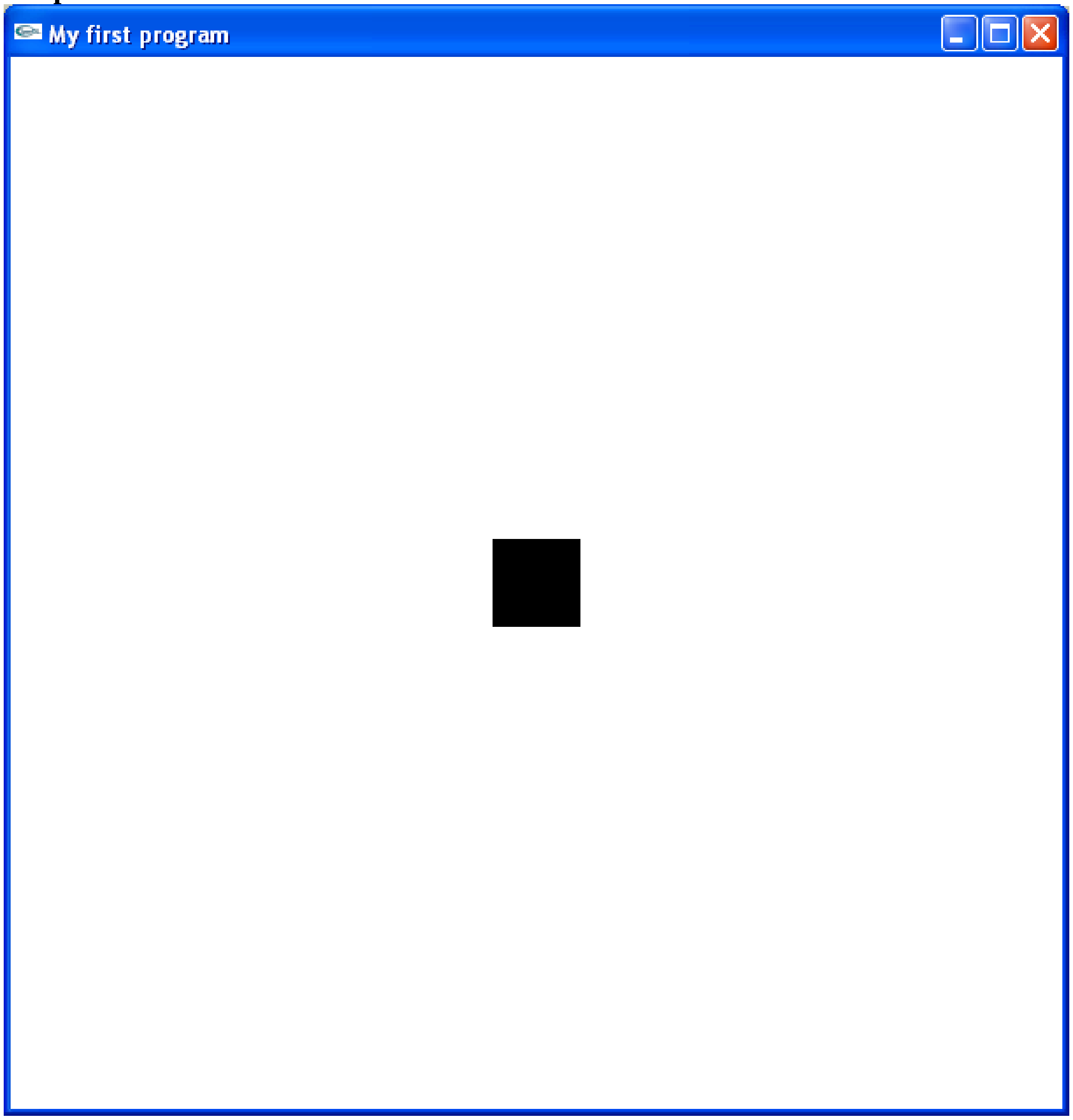
Running the Program

The very last thing you must do is call **glutMainLoop**(void). All windows that have been created are now shown, and rendering to those windows is now effective. Event processing begins, and the registered display callback is triggered. Once this loop is entered, it is never exited!

Example 2.1 shows how you might use GLUT to create the simple program

```
//this program draws a simple point in the center of output window.

#include<windows.h>
#include<gl/Gl.h>
#include<gl/glut.h>

void myInit(void);
void myDisplay(void);

void main(int argc,char** argv)
{
        glutInit(&argc,argv);

       //argc= arg count, specifies no. of arguments
       //argv= arg values, specifies values of those arguments

        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutInitWindowSize(600,600);
        glutInitWindowPosition(100,100);
        glutCreateWindow("My first program");
        glutDisplayFunc(myDisplay);
        myInit();
        glutMainLoop();
}

void myInit(void)
{
        glClearColor(1.0,1.0,1.0,1.0);
        glColor3f(0.0,0.0, 0.0);
        glPointSize(50.0);
        gluOrtho2D(0,600,0,600);
}

void myDisplay(void)
{
        glClear(GL_COLOR_BUFFER_BIT);
        glBegin(GL_POINTS);
        glVertex2i(300,300);
        glEnd();
        glFlush();
}
```

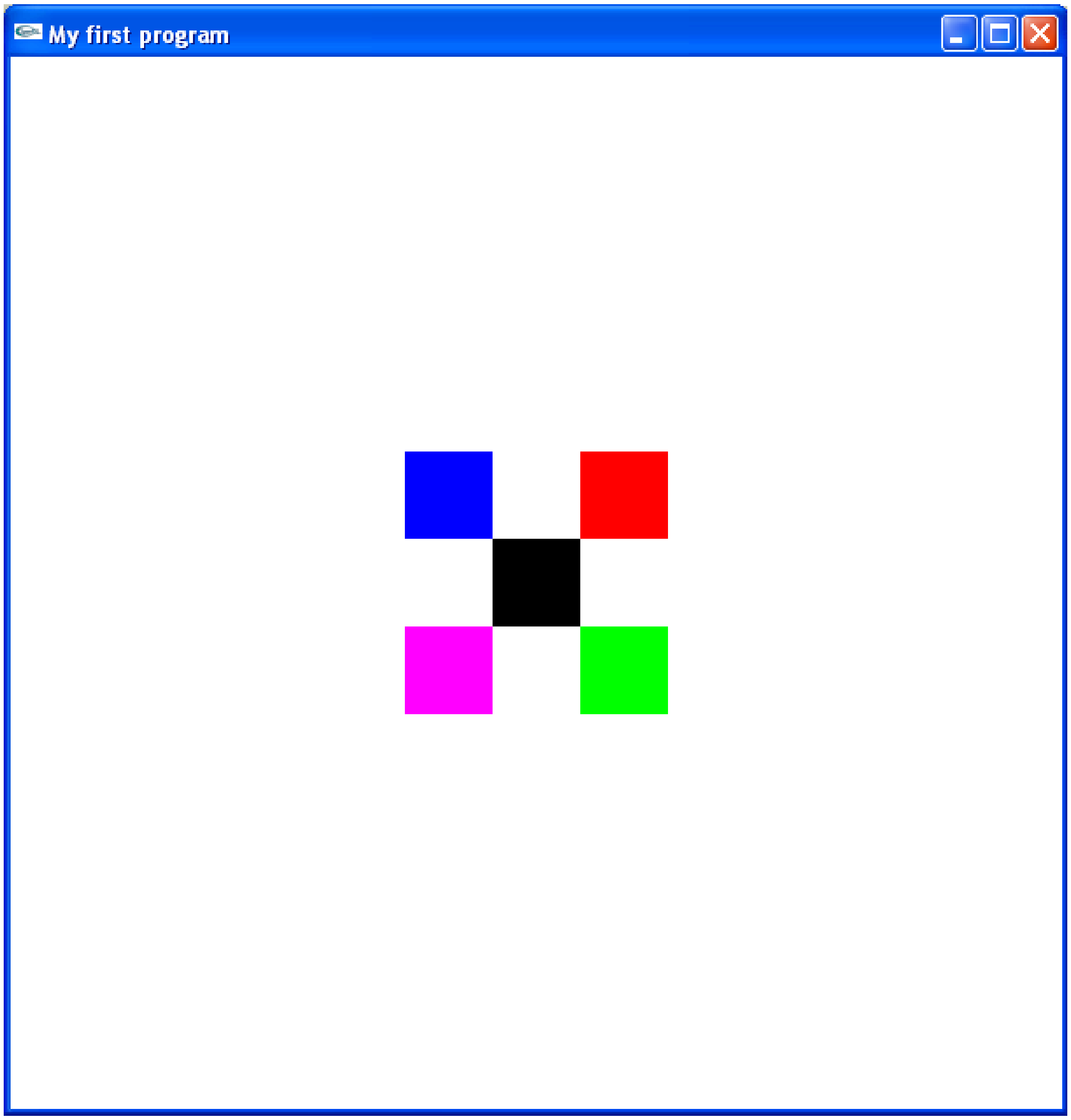
Output:



From Line 1 to 3are Library files that must be included in start of every openGL program.
Lines 4 and 5 are function declaration for two functions, which will be defined later in this program.
Lines through 6 to 14 show the contents of main function, in which basic properties of a window are set, like windows width, height, windows title, and the call to other functions included in the program.
Lines through 15 to 19 indicate the definition of myInit function. This function sets the color, size etc of the objects that are drawn in the main output window.
Lines through 20 to 26 are contents of the function myDisplay, in which object itself is drawn.

Exercises:

Write a program, which draws five points on output window. Each point having a different color. The output window should look like as under:



Lab Session 04

OBJECT

Drawing a triangle

In lab Session 03, we have learnt how to draw a simple point on output screen.

```
#include<windows.h>
#include<gl/Gl.h>
#include<gl/glut.h>

void myInit(void);
void myDisplay(void);

void main(int argc,char** argv)
{
        glutInit(&argc,argv);

       //argc= arg count, specifies no. of arguments
       //argv= arg values, specifies values of those arguments

        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutInitWindowSize(600,600);
        glutInitWindowPosition(100,100);
        glutCreateWindow("My first program");
        glutDisplayFunc(myDisplay);
        myInit();
        glutMainLoop();
}

void myInit(void)
{
        glClearColor(1.0,1.0,1.0,1.0);
        glColor3f(0.0,0.0, 0.0);
        glPointSize(50.0);
        gluOrtho2D(0,600,0,600);
}

void myDisplay(void)
{
        glClear(GL_COLOR_BUFFER_BIT);
        glBegin(GL_POINTS);
        glVertex2i(300,300);
        glEnd();
        glFlush();
}
```

Lines through 20 to 26 is the code, where any of the object is drawn. To draw a triangle, we simply replace this code by the respective code of triangle.

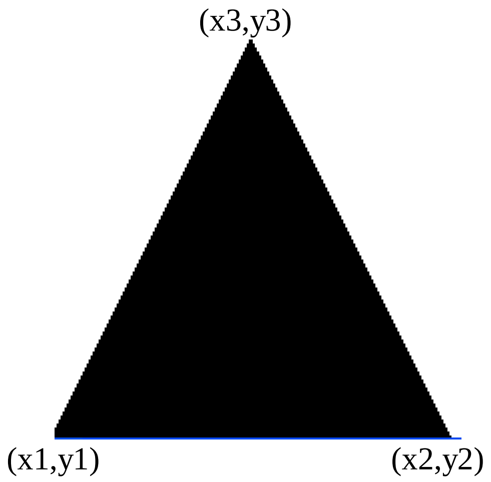
Hence we replace:

```
glBegin(GL_POINTS);
 glVertex2i(300,300);
 glEnd();
```

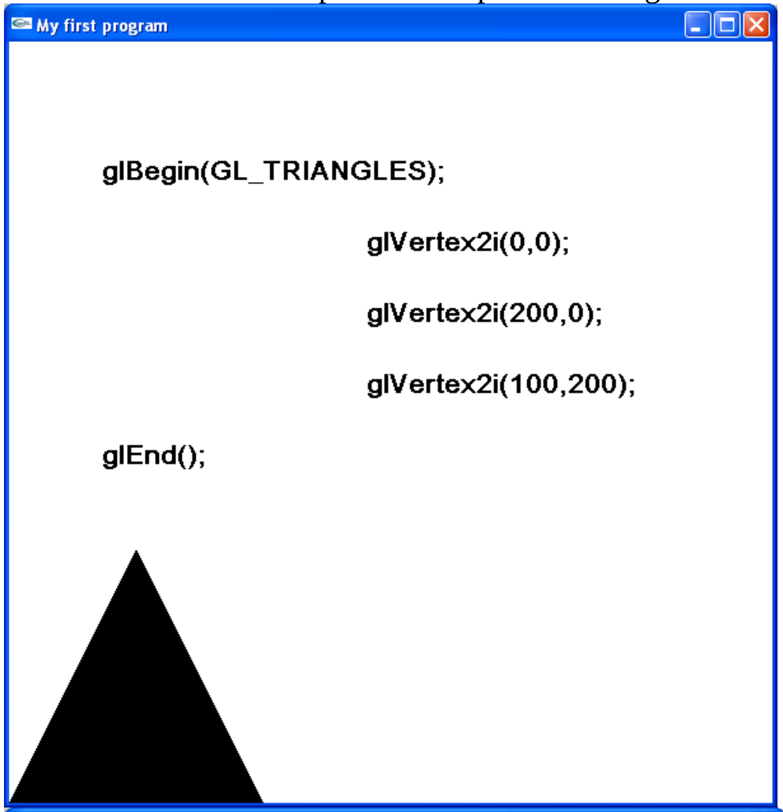
by:

```
glBegin(GL_TRIANGLES)
glVertex2i(x1,y1);
glVertex2i(x2,y2);
glVertex2i(x3,y3);
glEnd();
```

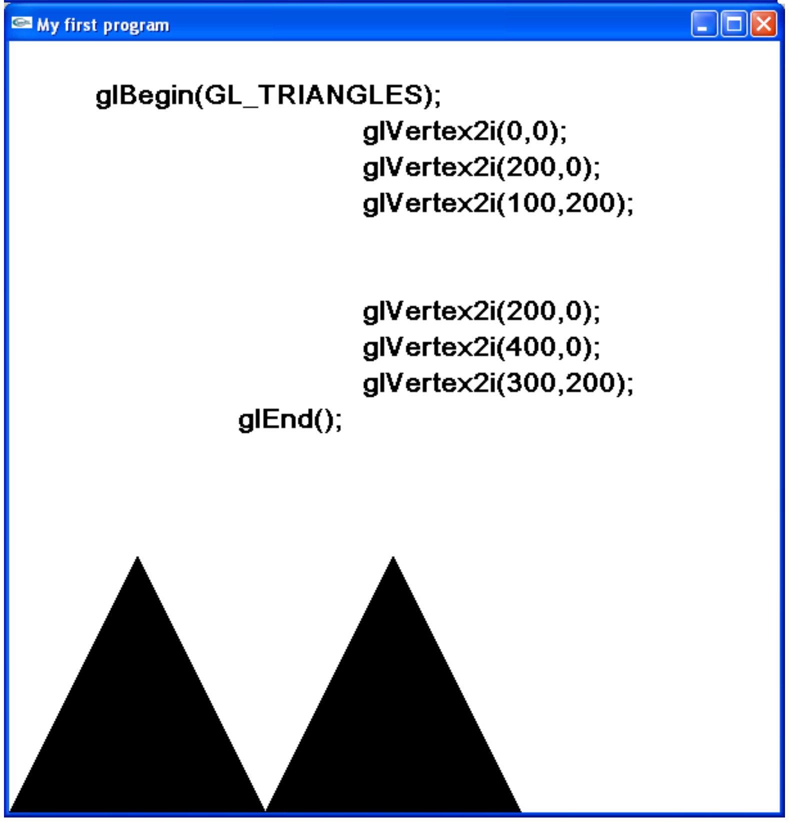
Note that simply code between geBegin and glEnd is changed.
This draws a triangle as below.



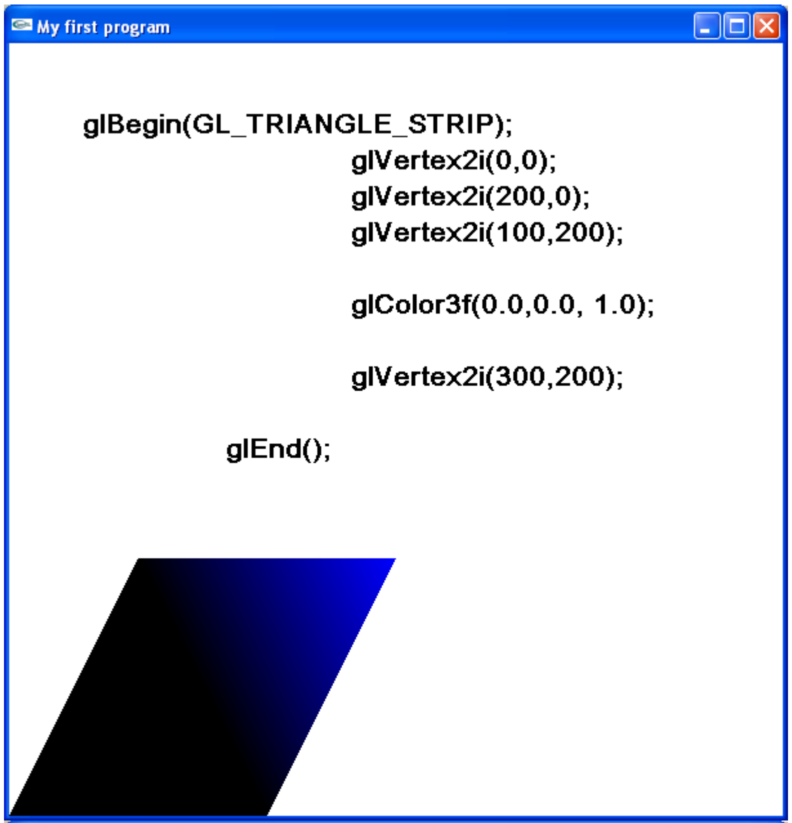
We now define some stepwise description of triangle construction:



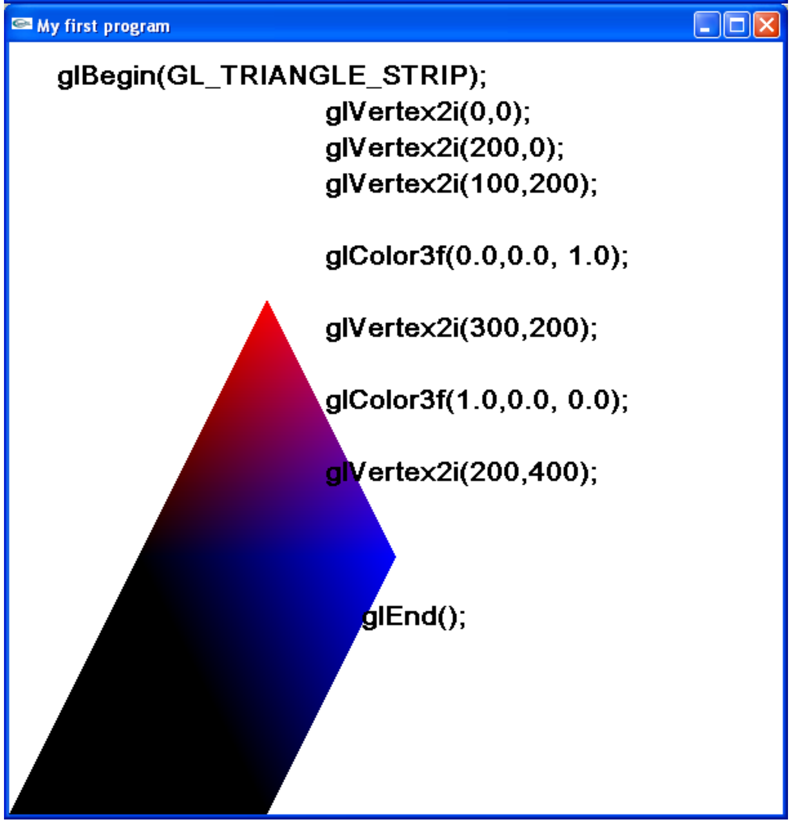
```
glBegin(GL_TRIANGLES);

                    glVertex2i(0,0);

                    glVertex2i(200,0);

                    glVertex2i(100,200);

glEnd();
```



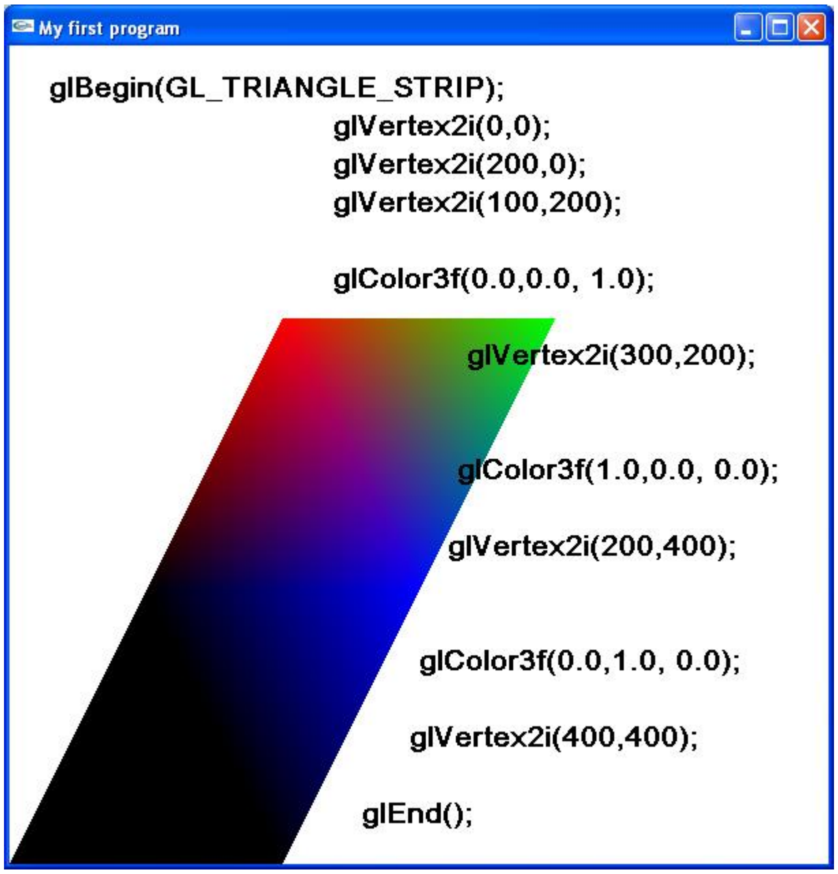
```
glBegin(GL_TRIANGLES);
                    glVertex2i(0,0);
                    glVertex2i(200,0);
                    glVertex2i(100,200);


                    glVertex2i(200,0);
                    glVertex2i(400,0);
                    glVertex2i(300,200);
          glEnd();
```



```
glBegin(GL_TRIANGLE_STRIP);
        glVertex2i(0,0);
        glVertex2i(200,0);
        glVertex2i(100,200);

        glColor3f(0.0,0.0, 1.0);

        glVertex2i(300,200);

    glEnd();
```



```
glBegin(GL_TRIANGLE_STRIP);
        glVertex2i(0,0);
        glVertex2i(200,0);
        glVertex2i(100,200);

        glColor3f(0.0,0.0, 1.0);

        glVertex2i(300,200);

        glColor3f(1.0,0.0, 0.0);

        glVertex2i(200,400);


         glEnd();
```

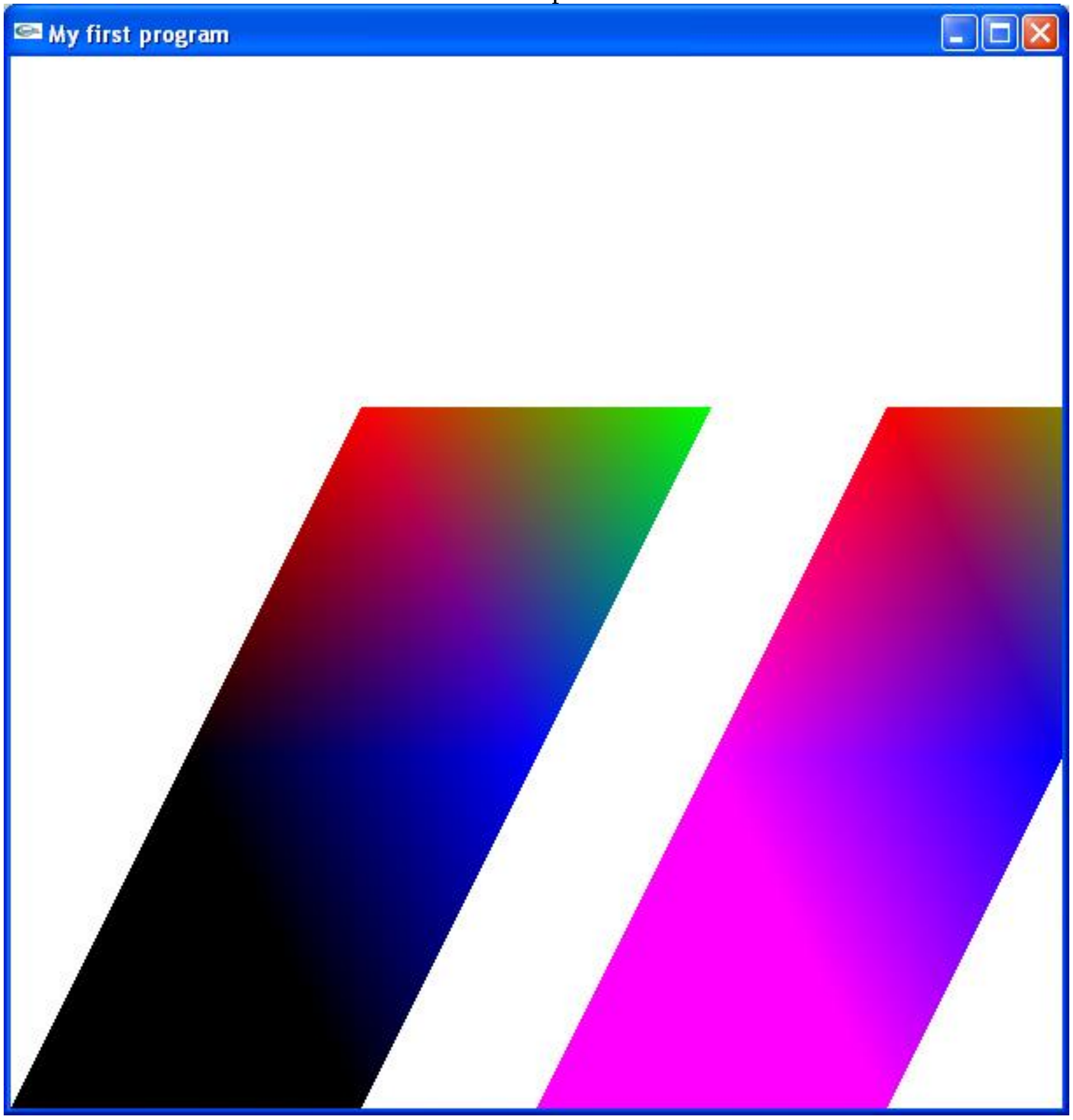


```
glBegin(GL_TRIANGLE_STRIP);
        glVertex2i(0,0);
        glVertex2i(200,0);
        glVertex2i(100,200);

        glColor3f(0.0,0.0, 1.0);

        glVertex2i(300,200);

        glColor3f(1.0,0.0, 0.0);

        glVertex2i(200,400);

        glColor3f(0.0,1.0, 0.0);

        glVertex2i(400,400);

        glEnd();
```

Note that the routine GL_TRIANGLE_STRIP draws a series of connected triangles, each having its respective defined color.

Exercise

Write code to show the results as under on output window.



Lab Session 05

OBJECT

Drawing Lines

In lab Session 03, we have learnt how to draw a simple point on output screen.

```
#include<windows.h>
#include<gl/Gl.h>
#include<gl/glut.h>

void myInit(void);
void myDisplay(void);

void main(int argc,char** argv)
{
        glutInit(&argc,argv);

        //argc= arg count, specifies no. of arguments
        //argv= arg values, specifies values of those arguments

        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutInitWindowSize(600,600);
        glutInitWindowPosition(100,100);
        glutCreateWindow("My first program");
        glutDisplayFunc(myDisplay);
        myInit();
        glutMainLoop();
}

void myInit(void)
{
        glClearColor(1.0,1.0,1.0,1.0);
        glColor3f(0.0,0.0, 0.0);
        glPointSize(50.0);
        gluOrtho2D(0,600,0,600);
}

void myDisplay(void)
{
        glClear(GL_COLOR_BUFFER_BIT);
        glBegin(GL_POINTS);
        glVertex2i(300,300);
        glEnd();
        glFlush();
}
```

To draw a line we simply need to edit the function myDisplay().
We simply replace:

```
glBegin(GL_POINTS);
 glVertex2i(300,300);
 glEnd();
```

by:

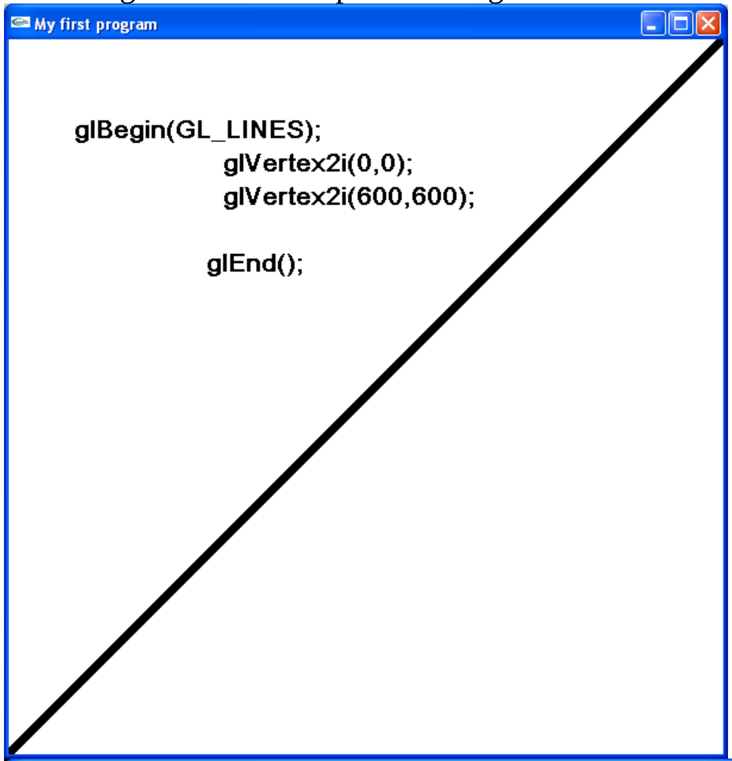
```
glBegin(GL_LINES);
 glVertex2i(x1,y1);
glVertex2i(x2,y2);
 glEnd();
```

We also need to make some change in function myInit().
Since we are making a live, so we don’t need the glPointSize( ), we now need function indicating width of line: i.e. glLineWidth(), which indicates the width of line in pixels.

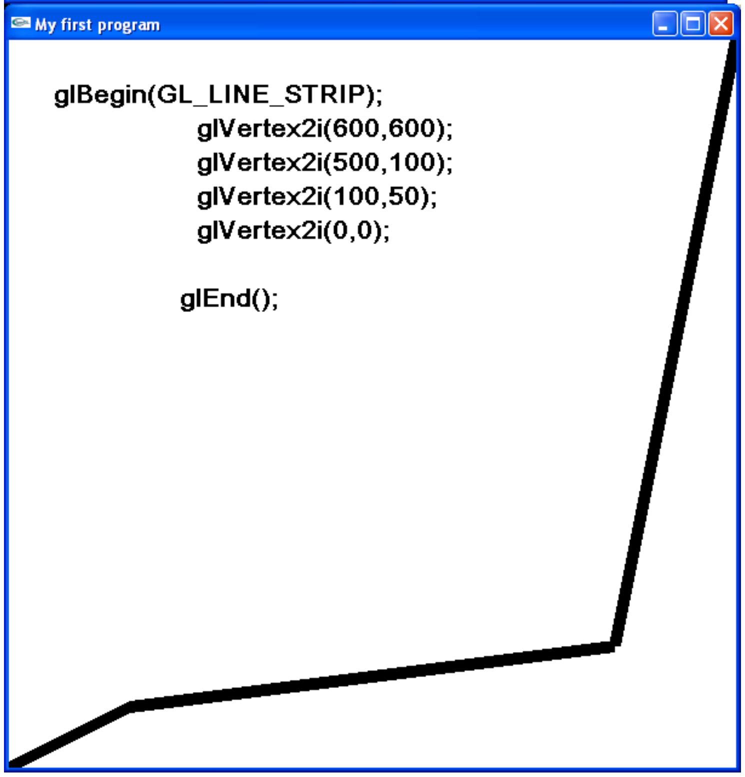
Hence the function myInit() becomes:

```
void myInit(void)
        {
                glClearColor(1.0,1.0,1.0,1.0);
                glColor3f(0.0,0.0, 0.0);
                glLineWidth(10.0);
                gluOrtho2D(0,600,0,600);
        }
```

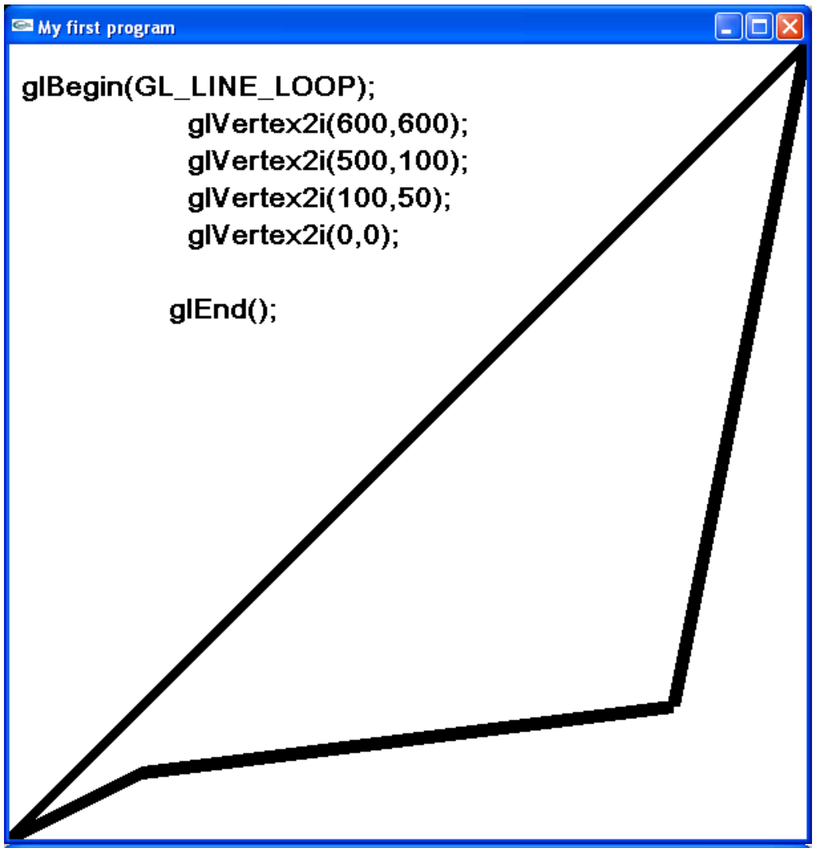
Following are some examples showing different codes with their respective outputs:



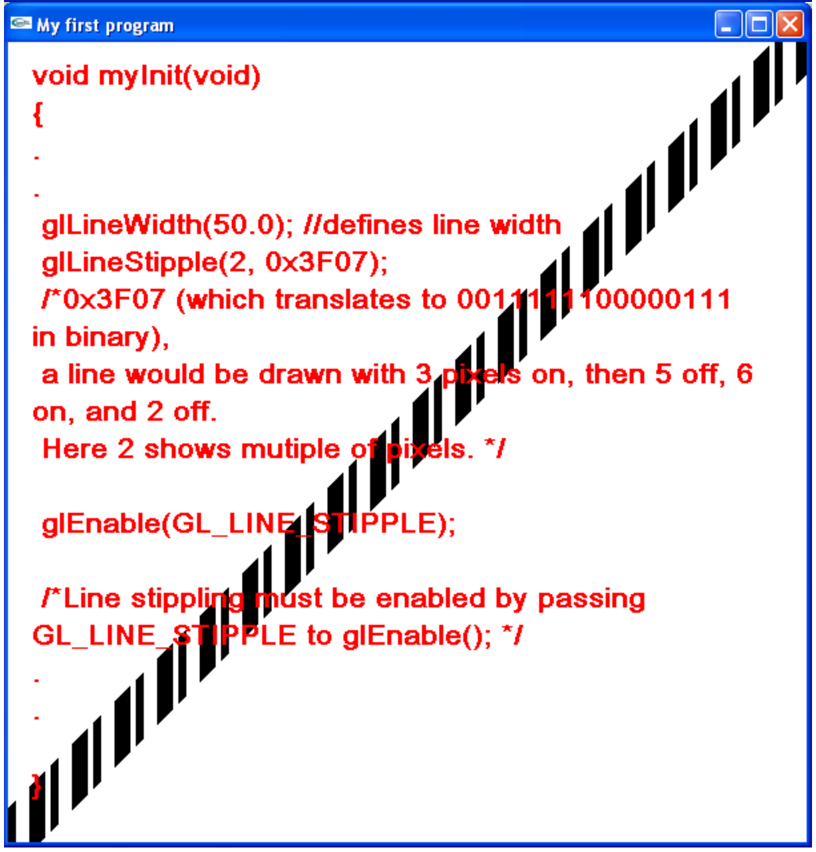
```
glBegin(GL_LINES);
        glVertex2i(0,0);
        glVertex2i(600,600);

       glEnd();
```



```
glBegin(GL_LINE_STRIP);
        glVertex2i(600,600);
        glVertex2i(500,100);
        glVertex2i(100,50);
        glVertex2i(0,0);

       glEnd();
```

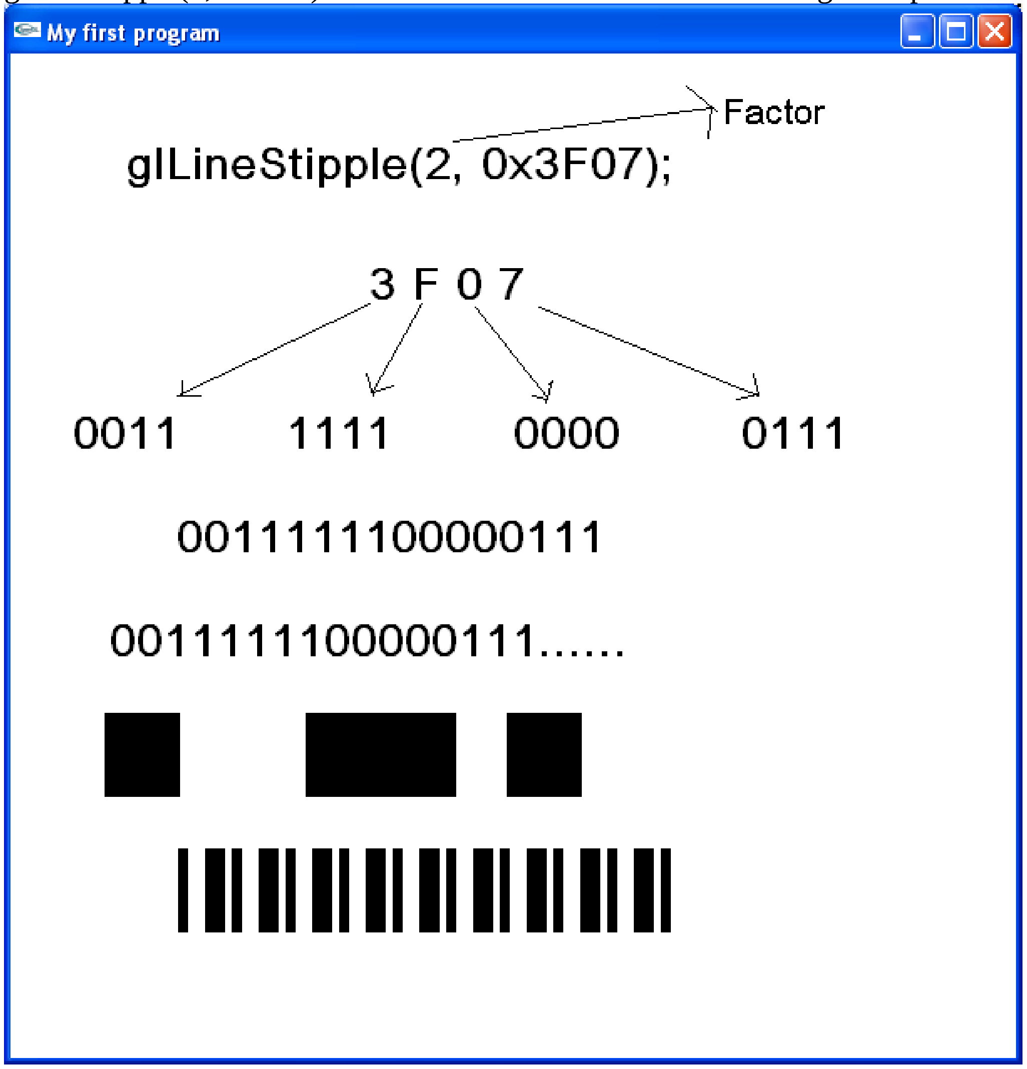


```
glBegin(GL_LINE_LOOP);
                glVertex2i(600,600);
                glVertex2i(500,100);
                glVertex2i(100,50);
                glVertex2i(0,0);

              glEnd();
```

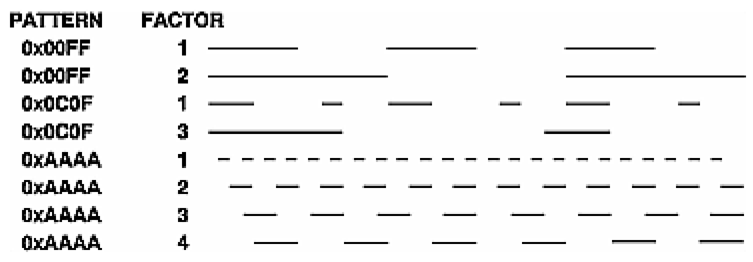


```
void myInit(void)
{
.
.
 glLineWidth(50.0); //defines line width
 glLineStipple(2, 0x3F07);
 /*0x3F07 (which translates to 0011111100000111
in binary),
 a line would be drawn with 3 pixels on, then 5 off, 6
on, and 2 off.
 Here 2 shows mutiple of pixels. */

 glEnable(GL_LINE_STIPPLE);

 /*Line stippling must be enabled by passing
GL_LINE_STIPPLE to glEnable(); */
.
.

}
```

Above output window shows a textured line, with its respective code. To describe line texture, we first enable this feature by placing routine glEnable(GL_LINE_STIPPLE), followed by glLineStipple(2,0*3F07)
glLineStipple(2,0*3F07) is more described in detail in following example:



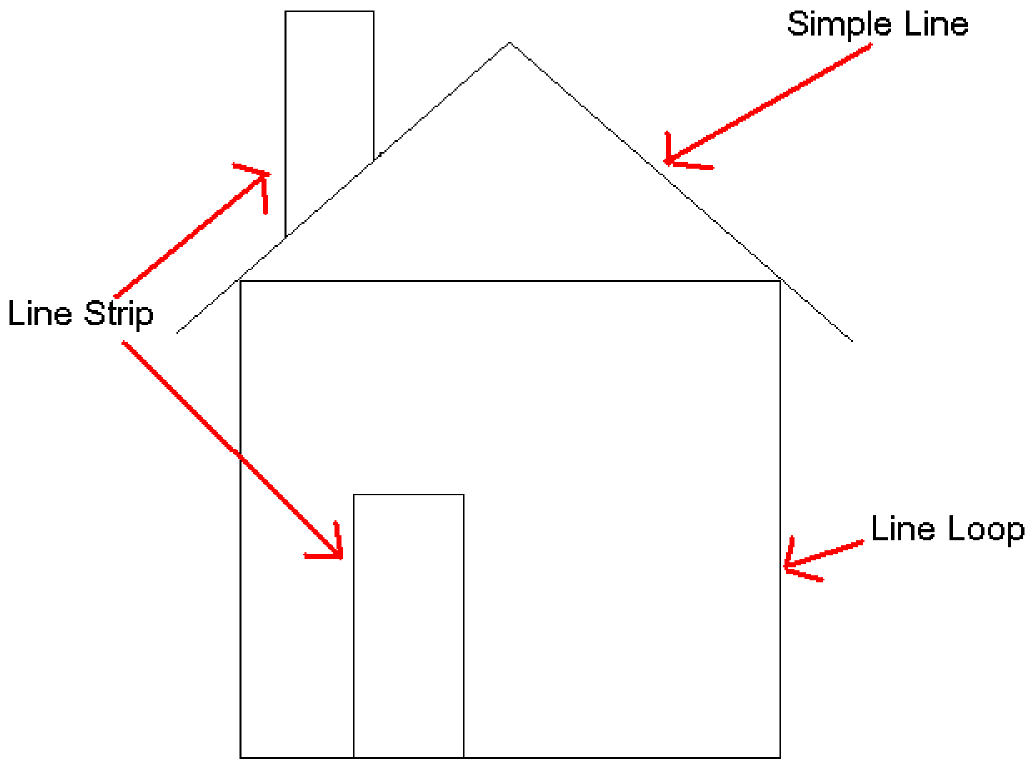
```
glLineStipple(2, 0x3F07);
```



PATTERN	FACTOR
0x00FF	1
0x00FF	2
0x0C0F	1
0x0C0F	3
0xAAAA	1
0xAAAA	2
0xAAAA	3
0xAAAA	4

Exercise:

Draw a simple house as shown below on your output window, use respective line functions as stated in figure below:



Lab Session 06

OBJECT

Drawing Polygons and Quad

Polygons

Polygons are the areas enclosed by single closed loops of line segments, where the line segments are specified by the vertices at their endpoints. Polygons are typically drawn with the pixels in the interior filled in, but you can also draw them as outlines or a set of points.

In general, polygons can be complicated, so OpenGL makes some strong restrictions on what constitutes a primitive polygon. First, the edges of OpenGL polygons can't intersect (a mathematician would call a polygon satisfying this condition a *simple polygon*). Second, OpenGL polygons must be *convex,* meaning that they cannot have indentations. Stated precisely, a region is convex if, given any two points in the interior, the line segment joining them is also in the interior. See Figure 2.2 for some examples of valid and invalid polygons. OpenGL, however, doesn't restrict the number of line segments making up the boundary of a convex polygon. Note that polygons with holes can't be described. They are nonconvex, and they can't be drawn with a boundary made up of a single closed loop. Be aware that if you present OpenGL with a nonconvex filled polygon, it might not draw it as you expect. For instance, on most systems no more than the convex hull of the polygon would be filled. On some systems, less than the convex hull might be filled.

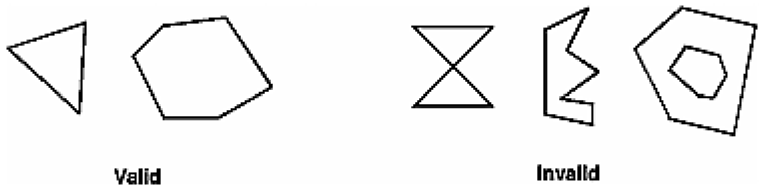


Figure 6.1 : Valid and Invalid Polygons

The reason for the OpenGL restrictions on valid polygon types is that it's simpler to provide fast polygon-rendering hardware for that restricted class of polygons. Simple polygons can be rendered quickly. The difficult cases are hard to detect quickly. So for maximum performance, OpenGL crosses its fingers and assumes the polygons are simple.

Many real-world surfaces consist of nonsimple polygons, nonconvex polygons, or polygons with holes. Since all such polygons can be formed from unions of simple convex polygons, some routines to build more complex objects are provided in the GLU library. These routines take complex descriptions and tessellate them, or break them down into groups of the simpler OpenGL polygons that can then be rendered

Since OpenGL vertices are always three-dimensional, the points forming the boundary of a particular polygon don't necessarily lie on the same plane in space. (Of course, they do in many cases - if all the *z* coordinates are zero, for example, or if the polygon is a triangle.) If a polygon's vertices don't lie in the same plane, then after various rotations in space, changes in the viewpoint, and projection onto the display screen, the points might no longer form a simple convex polygon. For example, imagine a four-point *quadrilateral* where the points are slightly out of plane, and look at it almost edge-on. You can get a nonsimple polygon that resembles a bow tie, as shown in Figure 6.2, which isn't guaranteed to be rendered correctly. This situation isn't all that unusual if you approximate curved surfaces by quadrilaterals made of points lying on the true surface. You can always avoid the problem by using triangles, since any three points always lie on a plane.

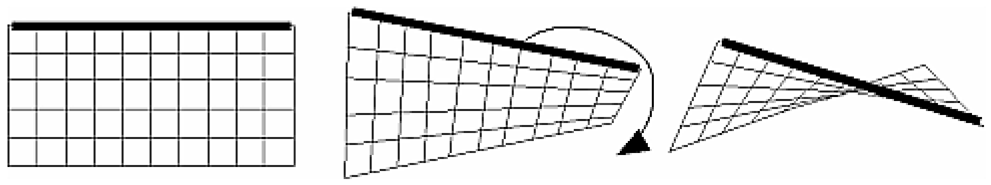


Figure 6.2 : Nonplanar Polygon Transformed to Nonsimple Polygon

Rectangles

Since rectangles are so common in graphics applications, OpenGL provides a filled-rectangle drawing primitive, **glRect*()**. You can draw a rectangle as a polygon, but your particular implementation of OpenGL might have optimized **glRect*()** for rectangles.

void ***glRect****{sifd}(TYPEx1, TYPEy1, TYPEx2, TYPEy2);*
void ***glRect****{sifd}****v****(TYPE*v1, TYPE*v2);*

> *Draws the rectangle defined by the corner points (x1, y1) and (x2, y2). The rectangle lies in the plane z=0 and has sides parallel to the x- and y-axes. If the vector form of the function is used, the corners are given by two pointers to arrays, each of which contains an (x, y) pair.*

Note that although the rectangle begins with a particular orientation in three-dimensional space (in the *x-y* plane and parallel to the axes), you can change this by applying rotations or other transformations.

Curves and Curved Surfaces

Any smoothly curved line or surface can be approximated - to any arbitrary degree of accuracy - by short line segments or small polygonal regions. Thus, subdividing curved lines and surfaces sufficiently and then approximating them with straight line segments or flat polygons makes them appear curved (see Figure 6.3). If you're skeptical that this really works, imagine subdividing until each line segment or polygon is so tiny that it's smaller than a pixel on the screen.

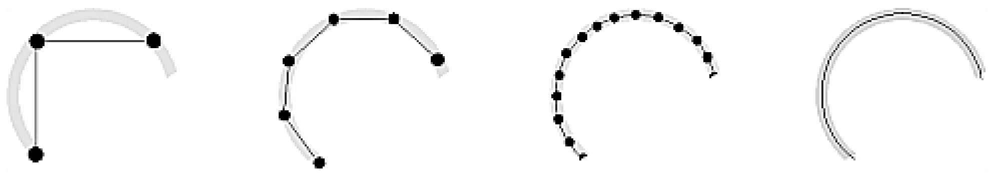


Figure 6.3 : Approximating Curves

Even though curves aren't geometric primitives, OpenGL does provide some direct support for subdividing and drawing them.

To draw a polygon on output window, we simply make some amendments in our first program “making a point”. We simply make changes in myDisplay() function.

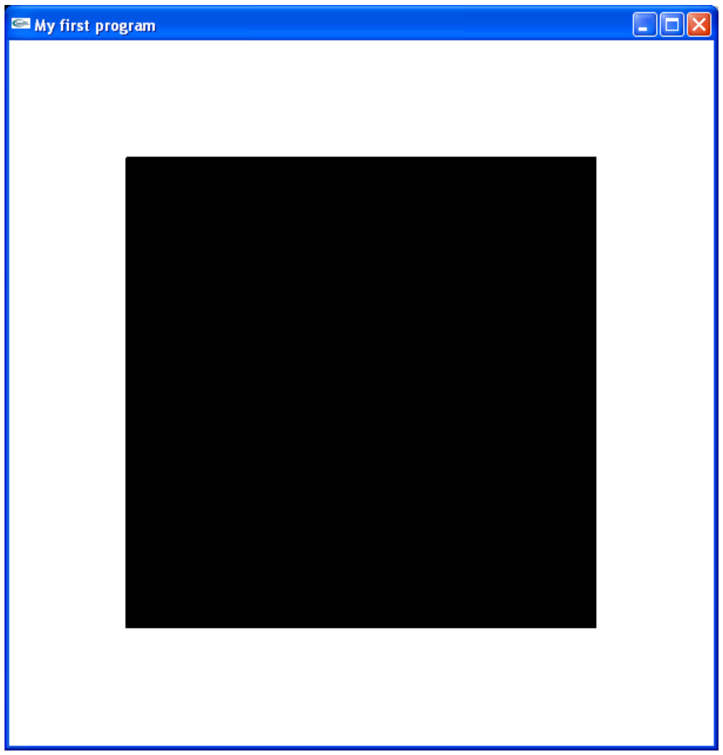
Since myDisplay function of “making a point” looks like:

```
void myDisplay(void)
{
        glClear(GL_COLOR_BUFFER_BIT);
        glBegin(GL_POINTS);
        glVertex2i(300,300);
        glEnd();
        glFlush();
}
```

To draw a polygon, we write this function as:

```
void myDisplay(void)
{
        glClear(GL_COLOR_BUFFER_BIT);
        glBegin(GL_POLYGON);
        glVertex2i(x1,y1);
        glVertex2i(x2,y2);
        glVertex2i(x3,y3);
        glVertex2i(x4,y4);
        glEnd();
        glFlush();
}
```

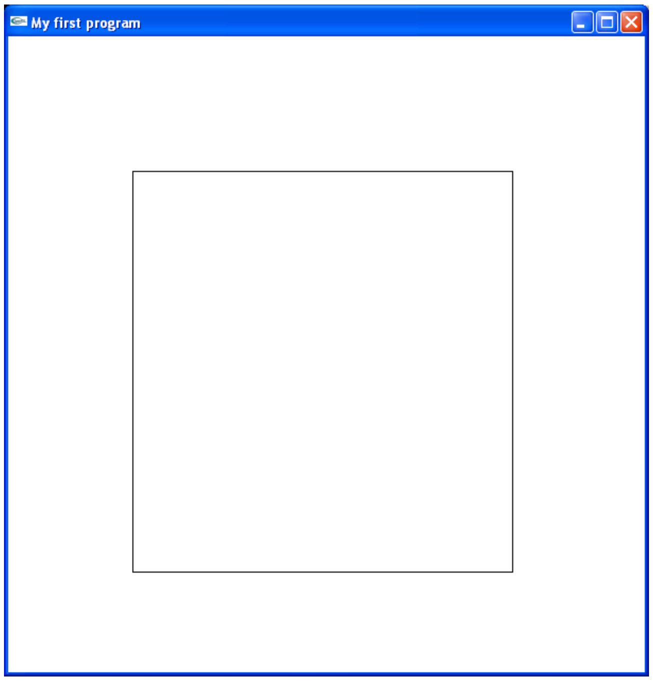
The above code has the following output:



We also can make en empty polygon, with following code:

```
void myDisplay(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glPolygonMode(GL_FRONT_BACK,GL_LINE)
    glBegin(GL_POLYGON);
    glVertex2i(x1,y1);
    glVertex2i(x2,y2);
    glVertex2i(x3,y3);
    glVertex2i(x4,y4);
    glEnd();
    glFlush();
}
```

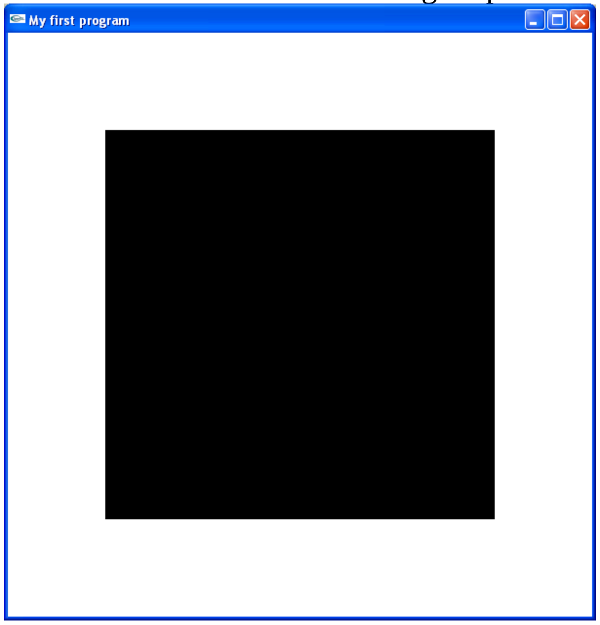
This yields the following output:



Similarly, to draw a quad, we place following code in myDisplay function:

```
glBegin(GL_QUADS);
                glVertex2i(100,100);
                glVertex2i(500,100);
                glVertex2i(500,500);
                glVertex2i(100,500);
glEnd();
```

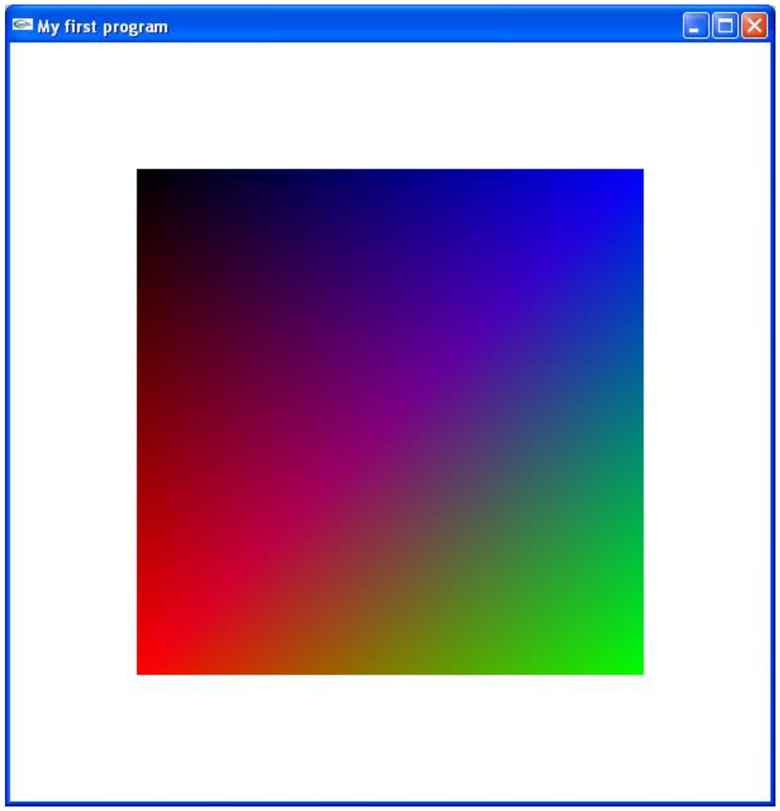
The above code has the following output:



We also can make some colorful quads:

```
glBegin(GL_QUADS);

        glColor3f(1.0,0.0, 0.0);
                glVertex2i(100,100);
                glColor3f(1.0,0.0, 0.0); //Red Color
                glVertex2i(500,100);
                glColor3f(0.0,1.0, 0.0); // Green Color
                glVertex2i(500,500);
                glColor3f(0.0,0.0, 1.0);  // Blue Color
                glVertex2i(100,500);

                glEnd();
```

The above code yields the quad having different color to each vertex.



Exercise:

Write code for showing a polygon and quad both in one output window.

Lab Session 07

OBJECT

Handling objects with keyboard

An important feature of OpenGL is to handle different objects with the help of keyboard. See the following program:

```
#include<windows.h>
#include<gl/Gl.h>
#include<gl/glut.h>

void myInit(void);
void myDisplay(void);

void keyboarda (unsigned char key, int x, int y)
        {
        if(key==112)//112 is the ascci code for alphabet 'p'.
                {
                glColor3f(0.0,0.0, 1.0);
                glBegin(GL_POINTS);
                glVertex2i(x,600-y);
                glEnd();
                glFlush();
                }

        if (key==27)                    //27 is the ascii code for the ESC key
                {exit (0); }            //end the program
        }

void main(int argc,char** argv)
        {
         glutInit(&argc,argv);

                //argc= arg count, specifies no. of arguments
                //argv= arg values, specifies values of those arguments

         glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
         glutInitWindowSize(600,600);
         glutInitWindowPosition(100,100);
         glutCreateWindow("My first program");
         glutDisplayFunc(myDisplay);
         glutKeyboardFunc(keyboarda);
         myInit();
```

```
	glutMainLoop();
}

void myInit(void)
{
	glClearColor(1.0,1.0,1.0,1.0);
	glColor3f(0.0,0.0, 0.0);
	glPointSize(10.0);
	gluOrtho2D(0,600,0,600);
}

void myDisplay(void)
{
	glClear(GL_COLOR_BUFFER_BIT);
	glFlush();
}
```

This code is same, which draws the simple point on a screen, except a simple addition of code, that is shown in bold letters.

Note that we have added a function **keyboarda( )** with two arguments. Which takes three arguments. Unsigned character key, int x, and int y. Unsigned char key is to assign any key of the keyboard in the program, which takes ASCII code of corresponding keyboard key, int x and int y set the x and y position of screen, where the object will be drawn on output window.

In the body of **keyboarda( )** function, there is a condition that if key pressed by the user is having ASCII code 112, which is ofcourse the ASCII code for p, the compiler will draw a point, where mouse pointer is placed.

```
		glBegin(GL_POINTS);
		glVertex2i(x,600-y);
		glEnd();
		glFlush();
```

Above is the code to draw a point on the screen. Note that **glVertex2i(x,600-y);** is the position on the screen, where mouse pointer is placed. Since this program takes the pixel resolution of windows, which is left to right for x-axis, and top to bottom for y-axis. Where as in OpenGL coordinate system, x-axis has the same pixel resolution, but y-axis goes from bottom to top. To cope with this situation, we have subtracted y from 600, which we have set height of the window in main function.

There is another condition, that if user presses Esc key(27 is the ASCII code for Esc key), the program simply exits the output window.

Now the important thing is call to above function in main. In main, we have a routine **glutKeyboardFunc(keyboarda);** which can call the above function. No matter what name is given to the function **keyboarda( ),** but it must be called in main through **glutKeyboardFunc( );.**

The function myInit( ) sets the properties of the object that is drawn, as we did in our previous program, and myDisplay( ) clears the screen, and then forces the program to execute through glFlush( ).

Exercise

Write a program, which when executed draws a triangle on out put screen by pressing t, and exits by pressing Esc key.

}

Lab Session 08

OBJECT

Handling objects with Mouse

Another important feature of OpenGL is to handle different objects with the help of mouse buttons.
See the following program:

```
#include<windows.h>
#include<gl/Gl.h>
#include<gl/glut.h>

void myInit(void);
void myDisplay(void);

void myMouse(int button, int state, int x, int y)
{
        if(button == GLUT_RIGHT_BUTTON)
        exit(0);
        else if(button == GLUT_LEFT_BUTTON)
                {
                glColor3i(0,0,1);
                glBegin(GL_POINTS);
                glVertex2f(x,600-y);
                glEnd();
                glFlush();
                }
}
void main(int argc,char** argv)
        {
        glutInit(&argc,argv);

                //argc= arg count, specifies no. of arguments
                //argv= arg values, specifies values of those arguments

        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutInitWindowSize(600,600);
        glutInitWindowPosition(100,100);
        glutCreateWindow("My first program");
        glutDisplayFunc(myDisplay);
        glutMouseFunc(myMouse);
        myInit();
        glutMainLoop();
}
```

```
void myInit(void)
{
	glClearColor(1.0,1.0,1.0,1.0);
	glColor3f(0.0,0.0, 0.0);
	glPointSize(10.0);
	gluOrtho2D(0,600,0,600);
}

void myDisplay(void)
{
	glClear(GL_COLOR_BUFFER_BIT);
	glFlush();
}
```

This code is same, which draws the simple point on a screen, except a simple addition of code, that is shown in bold letters.
Note that we have added a function **myMmouse( )** with two arguments. Which takes four arguments, int button, int state , int x, and int y. int button is to assign left or right button, which we often call left or right click. int state shows the previous state of the mouse, which is either clicked or released, int x and int y set the x and y position of screen, where the object will be drawn on output window.
In the body of **myMouse( )** function, there is a condition that if we make right mouse click, it simply exits from the output window. Note that **GLUT_RIGHT_BUTTON** indicates right mouse click, and **GLUT_LEFT_BUTTON** indicates left mouse click. Another condition that if we make left mouse click an object is drawn on the screen, where it was clicked.

```
		glBegin(GL_POINTS);
		glVertex2i(x,600-y);
		glEnd();
		glFlush();
```

Above is the code to draw a point on the screen. Note that **glVertex2i(x,600-y);** is the position on the screen, where mouse pointer is placed. Since this program takes the pixel resolution of windows, which is left to right for x-axis, and top to bottom for y-axis. Where as in OpenGL coordinate system, x-axis has the same pixel resolution, but y-axis goes from bottom to top. To cope with this situation, we have subtracted y from 600, which we have set height of the window in main function.

Now the important thing is call to above function in main. In main, we have a routine **glutMouseFunc(myMouse);**which can call the above function. No matter what name is given to the function **myMouse( ),** but it must be called in main through **glutMouseFunc( );.**

The function myInit( ) sets the properties of the object that is drawn, as we did in our previous program, and myDisplay( ) clears the screen, and then forces the program to execute through glFlush( ).

Exercise

Write a program, which when executed draws a polygon on out put screen by left mouse click, and exits by right mouse click.