FACULTY OF COMPUTER AND INFORMATION SCIENCES, AIN SHAMS UNIVERSITY

Lab 1

Using OpenGL with C++

OBJECTIVES:

1. **Be able to prepare the development environment.**
2. **USING OPENGL IN A C++ PROJECT.**
3. **Be able to draw OpenGL primitives.**

1. PREPARING THE DEVELOPMENT ENVIRONMENT

In this section, we describe how to prepare the development environment that we will use to create OpenGL programs. The steps described in this section need to be done only once on a given computer system to build OpenGL programs. You do not need to redo them every time you create a new OpenGL program.

We will be developing our OpenGL programs in C++ using the Visual Studio IDE. So, make sure that you have Visual Studio 2008 installed on your computer along with Visual C++.

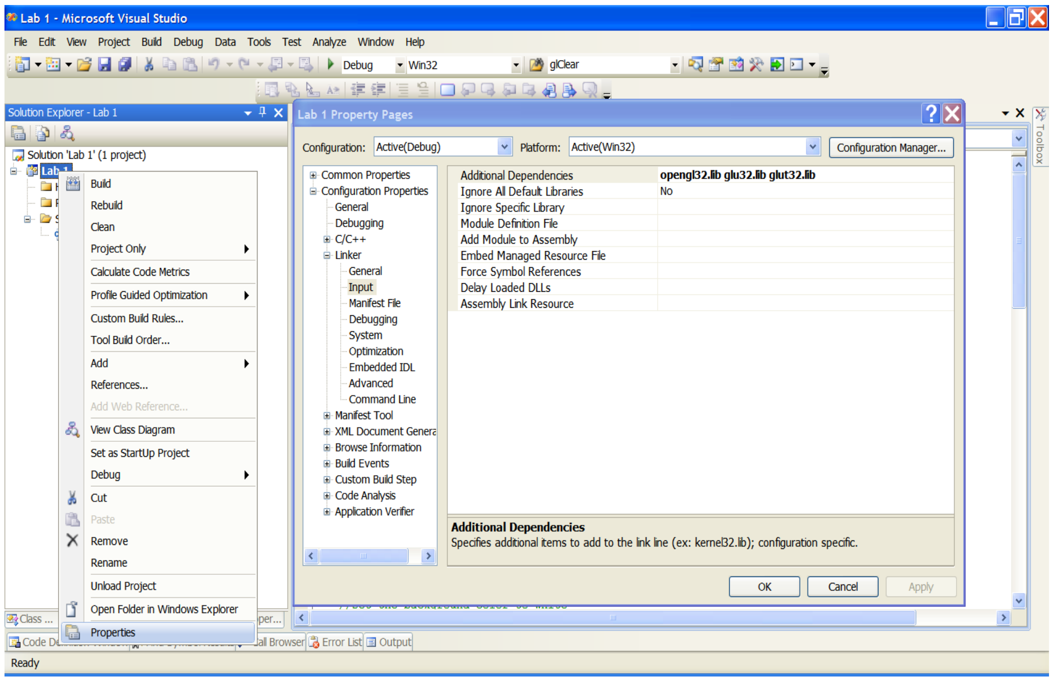
The next step is to install the GLUT library. GLUT is a platform-independent library that allows you to create and display windows and handle keyboard/mouse input. GLUT is not a part of the standard OpenGL. You can download this library from http://www.xmission.com/~nate/glut/glut-3.7.6-bin.zip. Let's assume that Visual Studio was installed at the directory "C:\Program Files\Microsoft Visual Studio 9.0\". Extract *glut-3.7.6-bin.zip* and place its contents as follows:

- Put *glut.h* inside "C:\Program Files\Microsoft Visual Studio 9.0\VC\include\gl\" (you may need to create the directory **gl** yourself).
- Put *glut32.lib* inside "C:\Program Files\Microsoft Visual Studio 9.0\VC\lib\".
- **Windows 32-Bit Users**: Put *glut32.dll* inside "C:\Windows\System32\".
- **Windows 64-Bit Users**: Put *glut32.dll* inside "C:\Windows\SysWOW64\".

2. USING OPENGL IN A C++ PROJECT

Starting with a new empty Win32 Console Application project, you will need to do the following steps to get OpenGL working:

1. Specify the following lib files as additional linker inputs: *opengl32.lib, glu32.lib, glut32.lib*. This can be performed by opening the properties of your project, and writing the names of the aforementioned files in the field *Configuration Properties>Linker>Input>Additional Dependencies*. This is indicated in the following figure.



Configuration: Active(Debug)   Platform: Active(Win32)

Additional Dependencies	opengl32.lib glu32.lib glut32.lib
Ignore All Default Libraries	No
Ignore Specific Library	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded DLLs	
Assembly Link Resource	

Additional Dependencies
Specifies additional items to add to the link line (ex: kernel32.lib); configuration specific.

Alternatively, add the following lines of code in any .cpp file in your project:

```
#pragma comment(lib, "opengl32.lib")
#pragma comment(lib, "glu32.lib")
#pragma comment(lib, "glut32.lib")
```

2. Create a new .cpp file, say *main.cpp*. Write the following *include*s at the top of your file

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

3. Define a main function that accepts command-line arguments as follows:

```
int main(int argc, char* argv[]) {
      return 0;
   }
```

Now, you are ready to create your OpenGL program. The basic structure of an OpenGL program is simple: a function to do **initialization**, and another to do the actual **rendering**. Rendering is the process by which a computer generates a picture from a description/model. Models are constructed from geometric primitives - points, lines, and polygons - that are specified by their vertices. Initialization is done once at the start of the program. Rendering is done each time the window needs redrawing (aka invalid or damaged window). This happens when a different window hides a part of your window and later reveals it, or when you resize your window. The program itself may initiate redrawing to update the displayed graphics (e.g. cars moving in a game).

On initialization, we use GLUT to create the window onto which OpenGL will put its drawings, and specify the function GLUT should call whenever the window needs redrawing (call it OnDisplay). We shall put all our initialization code inside a function called InitGraphics as follows:

```
/**
Creates the main window, registers event handlers, and
initializes OpenGL stuff.
*/
void InitGraphics(int argc, char *argv[]) {
      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_RGBA);
      //Create an 800x600 window with its top-left corner at
pixel (100, 100)
      glutInitWindowPosition(100, 100); //pass (-1, -1) for
Window-Manager defaults
      glutInitWindowSize(800, 600);
      glutCreateWindow("OpenGL Lab");
      //OnDisplay will handle the paint event
      glutDisplayFunc(OnDisplay);
      glutMainLoop();
}
/**
Sets the logical coordinate system we will use to specify
our drawings.
*/
void SetTransformations() {
      //set up the logical coordinate system of the window: [-
100, 100] x [-100, 100]
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluOrtho2D(-100, 100, -100, 100);
}
/**
Handles the paint event. This event is triggered whenever
our displayed graphics are lost or out-of-date.
ALL rendering code should be written here.
*/
void OnDisplay() {
      //set the background color to white
      glClearColor(1, 1, 1, 1);
      //fill the whole color buffer with the clear color
      glClear(GL_COLOR_BUFFER_BIT);
      SetTransformations();

      //ALL drawing code goes here

      //force previously issued OpenGL commands to begin
execution
      glFlush();
}
```

As you have seen, all OpenGL functions (also called OpenGL commands) start with the prefix *gl*. Similarly, all glut functions begin with the *glut* prefix. All the above code is useless if you forget to call InitGraphics in the main function as follows:

```
int main(int argc, char* argv[]) {
      InitGraphics(argc, argv);
      return 0;
}
```

You can now compile and run the program. All what you see is a white blank window displayed on top of the black console! You will learn in the next section how to draw static primitives on that window using OpenGL.

3. DRAWING OPENGL PRIMITIVES

OpenGL primitives include points, lines (line segments), triangles, quadrilaterals, and polygons. However complex an object is, you can approximate it using these primitives. To draw a primitive, you have to provide the following information to OpenGL:

1. The type of the primitive: This is done with glBegin(primitive_type)/glEnd() pair.
2. The vertices of the primitive: glVertex2f(x, y) specifies the coordinates of a vertex.
3. The color of each vertex: glColor3f(r, g, b) specifies the current drawing color.

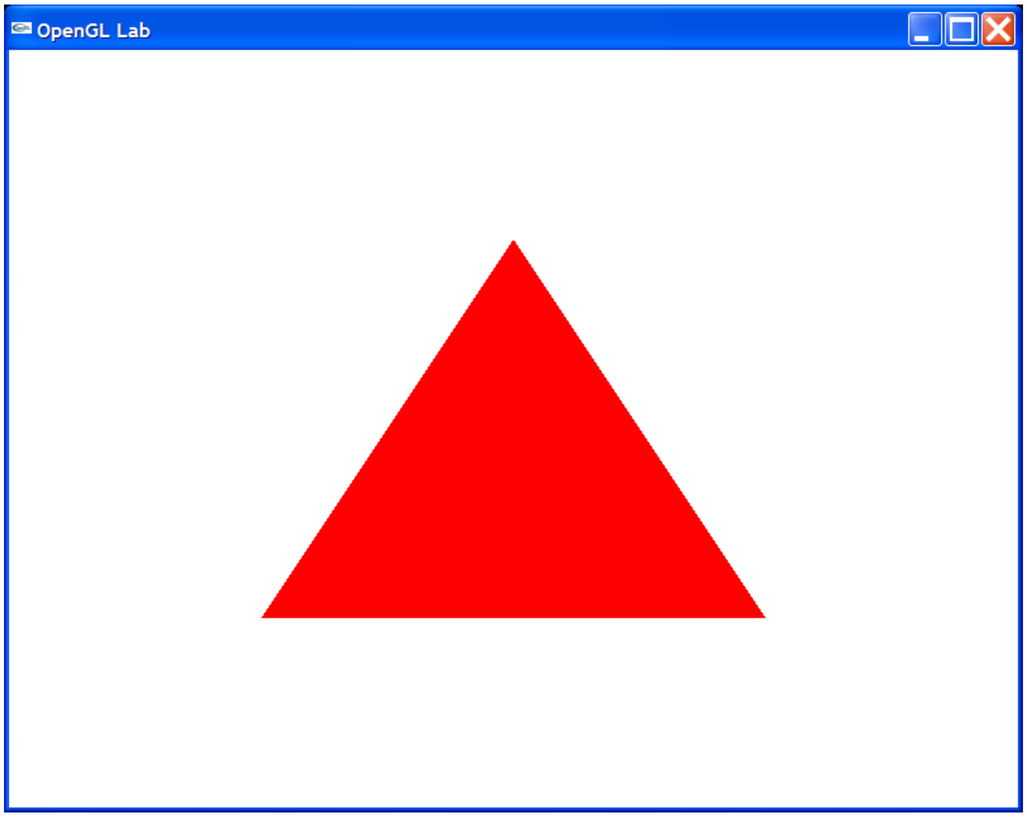
Before going into further details, we give an example of drawing a triangle in OpenGL:

```
/**
Handles the paint event. This event is triggered whenever
our displayed graphics are lost or out-of-date.
ALL rendering code should be written here.
*/
void OnDisplay() {
      //set the background color to white
      glClearColor(1, 1, 1, 1);
      //fill the whole color buffer with the clear color
      glClear(GL_COLOR_BUFFER_BIT);
      SetTransformations();

      glBegin(GL_TRIANGLES);
            glColor3f(1, 0, 0);
            glVertex3f(-50, -50, 0);
            glVertex3f(50, -50, 0);
            glVertex3f(0, 50, 0);
      glEnd();

      //force previously issued OpenGL commands to begin
      //execution
      glFlush();
}
```

If you run the above program, you will get the following output:



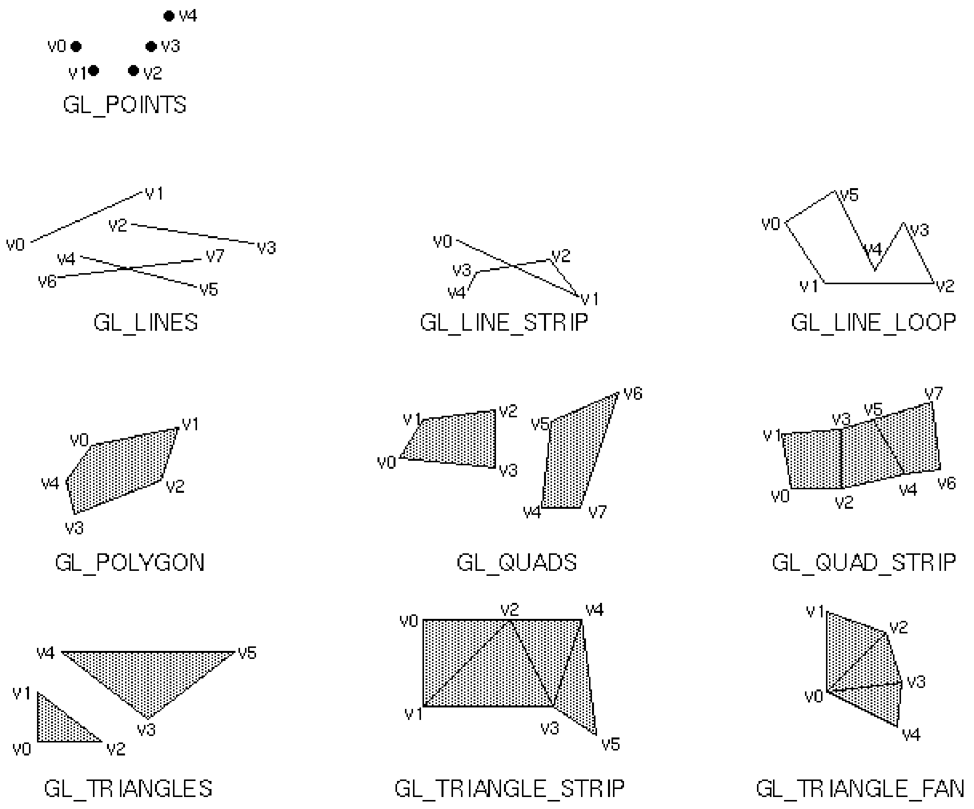
The command glColor3f(r, g, b) sets the current color, where r, g, b can take values between 0.0 and 1.0. r specifies the intensity of the red component, g specifies the green component, and b specifies the blue component. 0.0 means don't use any of that component, and 1.0 means use all you can of that component. Thus, glColor3f(1.0, 0.0, 0.0) makes the brightest red the system can draw, with no green or blue components. All zeros gives a black color (zero intensity in all components); in contrast, all ones makes white.

The command 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*() should be executed between a glBegin() and glEnd() pair.

You bracket each set of vertices between a call to glBegin() and a call to glEnd() (see the example code above). The argument passed to glBegin() determines what sort of geometric primitive is constructed from the vertices. Assuming that *n* vertices (v0, v1, v2, ... , vn−1) are described between a glBegin() and glEnd() pair, the following table shows the possible primitive types that you can pass to glBegin() and their descriptions:

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_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.
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_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_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 the following figure. *n* must be at least 4 before anything is drawn, and if *n* is odd, the final vertex is 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. The following figure should make the reason for the ordering obvious. *n* must be at least 3 for anything to be drawn.
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.


GL_POINTS
GL_LINES
GL_LINE_STRIP
GL_LINE_LOOP
GL_POLYGON
GL_QUADS
GL_QUAD_STRIP
GL_TRIANGLES
GL_TRIANGLE_STRIP
GL_TRIANGLE_FAN
v0
v1
v2
v3
v4
v5
v6
v7