|  |
| --- |
| fACULTY OF COMPUTER AND INFORMATION SCIENCES, Benha UNIVERSITY  (Benha FCI Computer Graphics Course , Class 2015/2016) |
| Lab 1 |
| Using OpenGL With C++ |
|  |
|  |
|  |

|  |
| --- |
|  |

# Objectives:

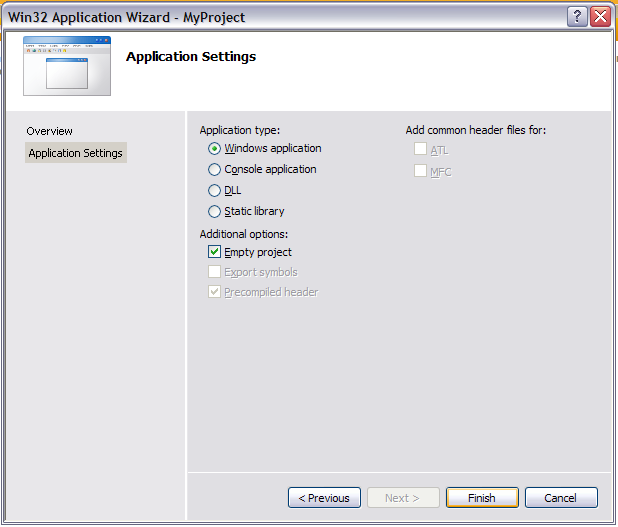
1. **Be able to prepare the development environment.**
2. **Be able to create a Console application that uses OpenGL.**
3. **Be able to draw OpenGL primitives.**

## Preparing the Development Environment

In this section, we describe how to prepare the development environment that we will use to create OpenGL programs. If not installed, you have to first install [Visual C++ 2008 Express Edition](http://www.microsoft.com/express/vc/) (for Free 30 days Trial) OR from <https://www.dreamspark.com/Products/Product.aspx?ProductId=9> Using your benha mail account. Then, OpenGL libraries and header files, If you do not have GLUT installed on your machine you can download it from: <http://www.xmission.com/~nate/glut/glut-3.7.6-bin.zip>

## Using OpenGL from a Windows Forms Application

**Run Visual C++ and create a new project:**

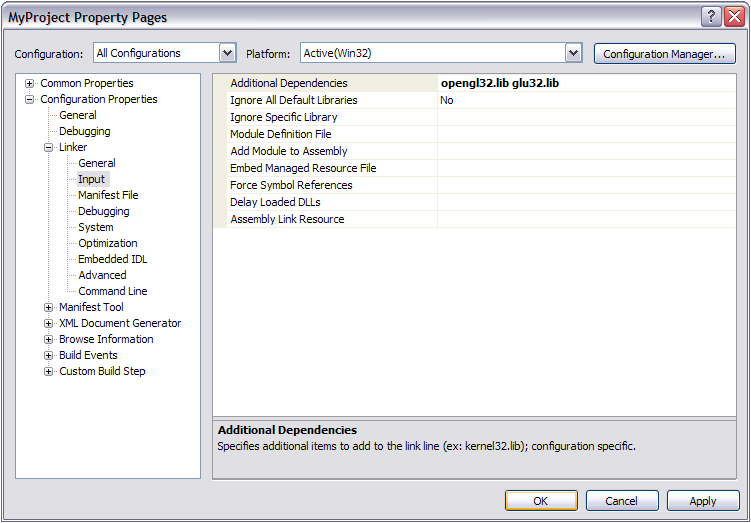
1. Under the File menu select New → Project (Ctrl+Shift+N).
2. Select Win32 Project, enter a Name, and click OK.
3. In the Wizard click Next, then check the box next to Empty Project, and click Finish.

**Add a new source file for the project:**

1. Under the Project menu select Add New Item (Ctrl+Shift+A).
2. Select C++ File (.cpp), enter a Name, and click OK.

**Link to the OpenGL libraries:**

1. Under the Project menu select Project Properties (Alt+F7) at the bottom.
2. Select Configuration Properties → Linker → Input from the navigation panel on the left.
3. Select All Configurations from the Configuration drop-down box at the top of the dialog. This ensures you are changing the settings for both the Debug and Release configurations.
4. Type “opengl32.lib glu32.lib” in Additional Dependencies and click OK.



**Add C++ file to the Project**

* 1. Solution Explorer →Source file →Right Click →Add→ New Item.
  2. Choose C++ file.
  3. Write the name of the C++ file (main.cpp) →Add.



First, we include our header:

#include <GL/glut.h>

#include <GL/gl.h>

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 control needs redrawing. This happens when a part of the control is hidden and shown again (by moving another window or by minimizing/restoring). The program itself can initiate redrawing to update the displayed graphics (e.g. cars moving in a game).

So, we add the following initialization and rendering methods to main.cpp:

/// <summary>

/// Prepares the control to respond to OpenGL commands. In addition,

/// all initialization related to graphics is done here.

/// </summary>

void initRendering ()

{

//set up the coordinate system of the control: [-100, 100] x [-100, 100]

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(-100, 100, -100, 100);

}

/// <summary>

/// All rendering code should be written here.

/// </summary>

void drawScene()

{

//set the background color to white; colors are briefly

//explained in the next section

glClearColor(1, 1, 1, 1);

//fill the whole color buffer with the clear color

glClear(GL\_COLOR\_BUFFER\_BIT);

//drawing code goes here

//force previously issued OpenGL commands to begin

//execution

glFlush();

}

You need to call initRendering() function from the main.cpp file as follow s:

int main(int argc, char\*\* argv) {

**//Initialize GLUT**

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

glutInitWindowSize(400, 400); //Set the window size

**//Create the window**

glutCreateWindow("Primitives – Draw Traingle");

initRendering (); //Initialize rendering

**//Set handler functions for drawing, keypresses, and window resizes**

glutDisplayFunc(drawScene);

**//Start the main loop. glutMainLoop doesn't return.**

glutMainLoop();

**//This line is never reached**

return 0;

}

You can now compile and run the program.

## 3. Drawing OpenGL Primitives

As you have already seen, all OpenGL commands begin with the prefix gl, and all these commands are defined as static methods in the gl.h.

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:

void drawScene()

{

//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);

//green

glColor3f(0, 1, 0);

//primitive type: triangles

glBegin(GL\_TRIANGLES);

//vertices of the triangle

glVertex2f(-25, -25);

glVertex2f(25, -25);

glVertex2f(0, 25);

//this is the end of the vertex list

glEnd();

glFlush();

}

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

(-100, -100)

(100, 100)

(0, 0)



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 figureshould 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. |

