OpenGL|ES Tutorials (2): Primitives & OpenGL Fundamentals

OpenGL Fundamentals

Now that you have created your environment, and tested it all works, you can now start coding some OpenGL|ES. The first thing a painter needs to paint his pictures is a blank canvas: Which is what we will be creating below.

Setting up the Project

Open up EVC++ 4.0, and create a project for the Gizmondo, as shown in the previous tutorial, but this time instead of creating a "typical 'Hello World!' Application" create "An empty project". Add a new file called main.cpp (File > New > C++ Source File)

Pre-processor Directives

To set up an OpenGL|ES Window is easy. The code is already written for you in the SDK. The "Framework" greatly simplifies general tasks like this.

To link the Framework to your project you would write:

```
#pragma comment(lib, "Framework.lib")
```

It is also possible to link your files via the Project Settings, although this method I find easier. You will also need to link libGLES_CM.lib to your project, so you will also need to add:

```
#pragma comment(lib, "libGLES_CM.lib")
```

Next you will need to "include" your header files. The following files will need to be linked:

windows.h	: Master include file for Windows applications
gl.h	: Header file for OpenGL. Includes all necessary OpenGL information
egl.h	: Header file which Extends OpenGL
framework.h	: Header file for framework, which is specific to the Gizmondo (Window Creation, Buttons defined, Status, etc)

frameworkGL.h	: Header file for the framework, which is specific to the Gizmondo and OpenGL stuff (Initialization etc)

You use the "#include" directive to include these file in your project. Thus, you would write the following in your project:

```
#include <windows.h>
#include <GLES\gl.h>
#include <GLES\egl.h>
#include <KGSDK\framework.h>
#include <KGSDK\frameworkGL.h>
```

Also add the following two variables, which will be talked about later:

```
static float rot[3];
static float loc[3];
```

Main Functions

In general, all the examples used in this tutorial will use the following logic:

Init	: This is where you set everything up. From the Matrix to Light Properties, and Texture Initialization.
Render	: Any drawing that needs to be done goes here. This function will be called every time a frame is drawn.
Update	: Most commonly used to look out for any key presses.
Main	: Basically where it all starts. This is the first function called as soon as you run your program. It is this function which sets up a main loop and calls other functions when necessary.

Along side these you might have other functions as well. For example, say if you wanted to draw several primitives (discussed later in this chapter) you could add a Draw_Triangle() or Draw_Square() function, then in Render() just add a call to these functions.

Init

Here's the initialization code for this chapter (Note: Line numbers have been added):

```
1: void Init()
2: {
3:    glLoadIdentity();
4:    GLfixed mat[4][4];
5:    float nearz, farz;
6:
```

```
7:    farz  = 300.0f;
8:    nearz = 0.01f;
9:
10:   memset(mat, 0, sizeof(mat));
11:   mat[0][0] = (int) (65536.0f * 2.4f);
12:   mat[1][1] = (int) (65536.0f * 3.2f);
13:   mat[2][2] = (int) (65536.0f * (farz / (farz - nearz)));
14:   mat[2][3] = (int) (65536.0f * 1.0f);
15:   mat[3][2] = (int) (65536.0f * ((-farz * nearz) / (farz - nearz)));
16:
17:   glMatrixMode(GL_PROJECTION);
18:   glLoadMatrixx(&mat[0][0]);
19:
20:   rot[0] = 0;
21:   rot[1] = 0;
22:   rot[2] = 0;
23:   loc[0] = 0;
24:   loc[1] = 0;
25:   loc[2] = 3;
26:
27:}
```

Analysis: We start off by creating storage for the content of out matrix and two variables which determine the camera's cut-off point. These two variables will not be used in this example, but in future tutorials. The Next section of code (lines 10 -15) sets up the matrix and it is loaded on Line 18. Also note that on Line 17 the OpenGL command glMatrixMode is used. This will be discussed in greater detail later on. GL_PROJECTION is used; other types of matrix modes that could have been used are GL_MODELVIEW and GL_TEXTURE. Lines 20 – 25 "Set" the camera up. The function is of type void: thus no return type is necessary. It is recommended at the end of this tutorial, you experiment by changing the values of some of these variables, to make sure you fully understand what is actually happening.

Render

The Render is the function which does all the drawing. It is called when every frame is drawn. Hence FPS: Frame's per seconds, is how many times this function is called in one second.

```
void Render()
{
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  DrawTriangle();

  glFinish();
  eglSwapBuffers(FrameworkGL_GetDisplay(), FrameworkGL_GetSurface());
}


//-------------------------------------------------------------------

 void DrawTriangle()
 {
       GLfloat tri[] =
       {
       -0.5f, -0.5f, -0.5f,
20:            -0.5f,  0.5f, -0.5f,
21:             0.5f, -0.5f, -0.5f
22:            };
23:
24:            GLfloat triColours[] =
25:            {
26:            1.0f, 0.0f, 0.0f, 1.0f,   //Red
27:            0.0f, 1.0f, 0.0f, 1.0f,   //Green
28:            0.0f, 0.0f, 1.0f, 1.0f,   //Blue
29:            };
30:
31:            glVertexPointer(3, GL_FLOAT, 0, tri);
32:            glColorPointer(4, GL_FLOAT, 0, triColours);
33:
34:            glEnableClientState(GL_VERTEX_ARRAY);
35:            glEnableClientState(GL_COLOR_ARRAY);
36:
37:            glDrawArrays(GL_TRIANGLES, 0, 3);
38:
39:            glShadeModel(GL_SMOOTH);
40:      }
```

Analysis: The render he is split into two parts. The general operations which need to be performed are in the root function "Render()" and the operations that need to be performed, specific to creating a triangle are in function "DrawTriangle()".

At the start of the function Render, there is a GL Command called glClearColor(), which 4 values are passed into. This is the function which clears the screen to the colour specified in the parameters. The first parameter is the "RED" value of the colour – where 1.0f is the maximum and 0.0f is the minimum. The next value is the "GREEN" value of the colour, and likewise 1.0f is the maximum and 0.0f is the minimum. The third is the "BLUE" value, and as previous 1.0f is the maximum and 0.0f is the minimum. The last value is the "ALPHA" part. In short, this value is the "transparency" of the colour. We will discuss this in a future tutorial. In this case, the color is set to R-0 G-0 B-0 and A-0: Black. By altering these values you can have a different coloured background.

The next command on Line 4 (glClear) is the command that actually clears the Screen – glClearColor just sets the colour. `GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT` just tells OpenGL|ES to clear the colour buffer and the depth buffer. On Line 6, function DrawTriangle is called and code execution jumps to Line 15.

Firstly, a vertex array is defined. This is one of the major differences between OpenGL and OpenGL|ES, where you would most commonly have used a glBegin and glEnd statement. Each line symbolises the x, y and z co-ordinates of each vertex.

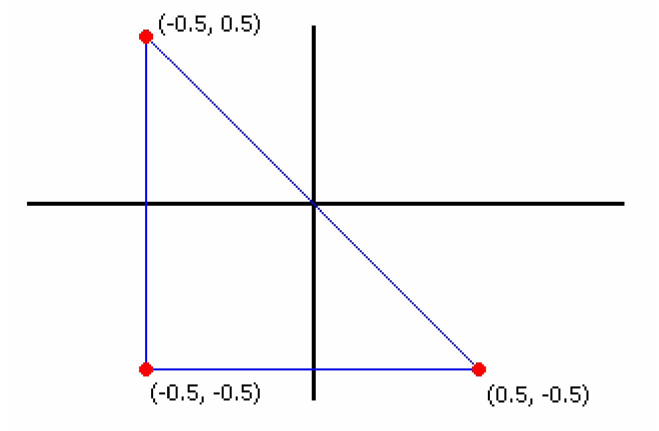

Figure 2.1 How vertexes are drawn

The diagram represents how OpenGL interprets the vertex array and draws it onto the screen. The easiest method is to imagine a 3-Dimensional axis, and then plot the points (vertexes) on. The X co-ordinate is the how far left or right it is. The Y co-ordinate is how far up or down it is. And the Z xo-ordinate is how far "into the screen" it is, or if it is as if it is "coming out of the screen". The Z co-ordinate is ideally used in 3D games, where as only the X and Y are used for 2D.

The next statement is glVertexPointer(), and this tells OpenGL what it is going to draw (gives the data source i.e. the vertex array). This first parameter is the number of co-ordinates for each vertex, in this case 3. The second parameter specifies the data type of each coordinate in the array. Symbolic constants GL_SHORT, GL_INT, GL_FLOAT, and GL_DOUBLE are accepted. The third parameter tells OpenGL the byte offset between each vertex. Here it is 0. If it was 1, the co-ordinates would be read every other one. The last parameter is a pointer to the array you wish to use.

Like the glVertexPointer, the glColorPointer loads the colour array. The first parameter specifies the number of componets for the color (i.e RGBA is 4). The next parameter specifies the data type for each color componet in the array. We defined the tricolours array as GLfloat, hence we have put GL_FLOAT down here.

Next, we use glEnableClientState to enable the client-side capabilities: GL_COLOR_ARRAY and GL_VERTEX_ARRAY.

Finally, the triangle is drawn onto the screen using the glDrawArrays command. The first parameter states we are drawing a triangle. The second parameter tells OpenGL to start at the beginning of the array, and the third parameters tell OpenGL to draw the next 3 vertexes.

glShadeModel(GL_SMOOTH); means that we want the triangle to shaded, hence creating a gradient effect. If we were to use GL_FLAT instead of GL_SMOOTH OpenGL would just colour the triangle using the last colour specified.

Function DrawTriangle then returns; code execution jumps back to line 8. Where glFinish() is called. glFinish blocks until all GL Execution is complete. (i.e. it does not return).

Finally, `eglSwapBuffers()` is called. This simply swaps our buffers round (back buffer and front buffer).

Update

The Update function is called for “maintenance” purposes. It is often used to process key presses.

```
void Update()
{
  GLfixed mat[4][4];

  memset(mat, 0, sizeof(mat));
  mat[0][0] = (int) (65536.0f *  cos(rot[0]));
  mat[0][2] = (int) (65536.0f *  sin(rot[0]));
  mat[1][1] = (int) (65536.0f *  cos(rot[1]));
  mat[1][2] = (int) (65536.0f *  sin(rot[1]));
  mat[2][0] = (int) (65536.0f * -sin(rot[0]) * cos(rot[1]));
  mat[2][2] = (int) (65536.0f *  cos(rot[0]) * cos(rot[1]));
  mat[3][2] = (int) (65536.0f * loc[2]);
  mat[3][3] = 65536;

  glMatrixMode(GL_MODELVIEW);
  glLoadMatrixx(&mat[0][0]);
}
```

Analysis: You might recognise some of the code here. In this tutorial, the update function simply “Reloads the Matrix” (no pun intended :P), as discussed in the Init Section.

Main

The Update function is called for “maintenance” purposes. It is often used to process key presses.

```
1:      int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPTSTR lpCmdLine, int nCmdShow )
2:      {
3:        if (!Framework_Init(hInstance, hPrevInstance))
4:        {
5:                return 0;
6:        }
7:
8:        if (!FrameworkGL_Init())
9:        {
10:               Framework_Close();
11:               return 0;
12:       }
13:
14:       Init();
15:
16:
17:       do
18:       {
19:               Render();
20:               Update();
21:               Framework_ProcessMessages();
22:
23:               if (!Framework_IsAppActive())
24:               {
25:                             while (!Framework_IsAppActive())
26:                             {
27:                               Framework_ProcessMessages();
28:                               Sleep(50);
29:                             }
30:               }
31:
32:       } while (!Framework_IsButtonPressed(FRAMEWORK_BUTTON_HOME));
33:
34:       FrameworkGL_Close();
35:       Framework_Close();
36:
37:       return 0;
38:     }
```

Analysis: The main function starts off by creating the window (Line 3). This is greatly been simplified to just statement. The actual code is in the Framework. Next, it initializes the OpenGL part (Line 8). That is, getting context, surface, display etc. I won’t go into much detail of these as it is not necessary. Note that if either fails, the program exits. Next we call our initialization on Line 14. The main message starts on line 17. It calls the Render Function, Update Function, and the Framework_ProcessMessages Function, whilst the home button isn’t pressed. (i.e. pressing the home button exits the app). It also checks if the application is active. If it is not, the program goes to “sleep”. You might recognise this effect if you leave the gizmondo for a while and the backlight turns off. Finally, we clean up any resources used.

Output

You should end with something similar to that in Figure 2.2

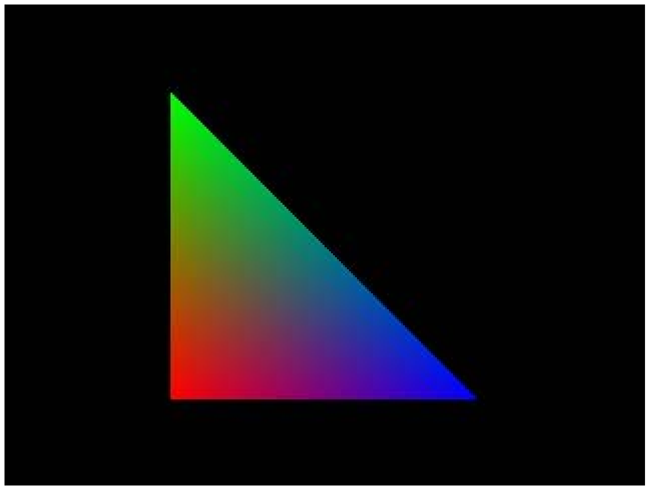

Figure 2.2 Tutorial 1 Output

This is a prime example of a Primitive: "A low-level object from which higher-level, more complex objects can be constructed". Triangles are primitives in OpenGL, and almost all objects are made from these.

- End of OpenGL|ES Tutorials: Primitives & OpenGL Fundamentals-