Lab 2 – A Step into 3D

In this lab we are going to stick with the Fixed Function pipeline but move to a fully 3D Scene. We are also going to start using some of memory resources, namely Vertex Buffer Objects and Element Array Objects.

1. Dealing with Extensions

Working with a VBO requires using extensions above and beyond standard OpenGL that ships with the Platform SDK on Windows. We could load the extensions manually or we can use GLEW to automatically load and register extensions.

1.1 Link with GLEW SDK

We need to link against the GLEW SDK, the directory is installed on **C:\Sepecialised Applications** in the lab.

- You should add **C:\Specialised Applications\glew-1.10.0\include** to **Additional Include Directory** for the project
- You should add for **C:\Specialised Applications\glew-1.10.0\lib\Release\Win3** to **Additional Library Directory** for the project
- You should add **glew32.lib** to the **Additional Dependencies**
- Copy **glew32.dll** to the Debug Directory of the project

1.2 Initialise GLEW

In order to properly register all extensions available to us we need to initialize GLEW. Navigate to the **initOpenGL** function and add the following to the bottom of the function

```
GLenum err = glewInit();
if (GLEW_OK != err)
{
	/* Problem: glewInit failed, something is seriously wrong. */
	std::cout << "Error: " << glewGetErrorString(err) << std::endl;
}
```

2. Rendering a Triangle with a Vertex Buffer Object

The first thing we are going to do is create a Vertex Buffer Object(VBO), fill it with vertices and finally draw the contents of the buffer

2.1 Create a InitGeometry Function

You should add the following function to **main.cpp**, add this empty function near the rest of the global functions

```
void initGeometry()
{

}
```

This function will basically create and fill a VBO with some vertex data, and will be called once OpenGL has been initialized

2.2 Global Vertex Data

The VBO is used to hold vertex data, this vertex data usually takes the form of arrays of vertices(this can be a series of flat arrays with each component of the vertex a separate array or can be Array of structures which represent the data in a vertex). In this part of the lab we are going to have a simple array of floats which are the positional values of model.

Add the following code to the global values of the **main.cpp** file

```
float triangleData[] = { 0.0f, 1.0f, 0.0f, // Top
                        -1.0f, -1.0f, 0.0f, // Bottom Left
                         1.0f, -1.0f, 0.0f }; //Bottom Right
```

2.3 Global VBO Integer ID

Many of the memory resources used by OpenGL are referred to be an integer ID, this means in order to create and use VBOs we have to first create an integer. This will be filled out when generate the buffer, passed into functions which bind the buffer to the pipeline and also when copying data into the buffer.

Add the following global variable to **main.cpp** to the global variables section of the file.

```
GLuint triangleVBO;
```

2.4 Creating and filling the VBO

We are going to generate the VB and fill it with the above vertex data, navigate to the **initGeometry** function and add the following code to create and copy the vertex data into the buffer

```
//Create buffer
glGenBuffers(1, &triangleVBO);
// Make the new VBO active
glBindBuffer(GL_ARRAY_BUFFER, triangleVBO);
//Copy Vertex Data to VBO
glBufferData(GL_ARRAY_BUFFER, sizeof(triangleData),
             triangleData, GL_STATIC_DRAW);
```

glGenBuffers call takes in an integer which specifies the number of buffers you are going to generate and the 2nd parameter is a pointer to one or an array of integers.

glBindBuffer call binds the specified buffer(2nd parameter) to the pipeline(this is a state!), the 1st parameter specifies what type of buffer we are binding(Array Buffer) will hold vertices.

glBufferData copies data to the bound buffer, 1st parameter is what type of buffer we are copying too, 2nd parameter is the size of the data we are copying into the buffer, 3rd parameter is the actual data we are copying, and the last parameter is a hint to OpenGL on what do with the buffer data, in this case the data in the buffer will not be updated.

2.5 Cleanup

The process of generating buffer allocates some memory, we need to reclaim this memory. Navigate to the **CleanUp** function and imediatley after the first curly brace add the following code to delete the buffer.

```
glDeleteBuffers(1, &triangleVBO);
```

This will delete the number of buffers specified(1st parameter), with the actual buffers being passed in as the 2nd parametr

2.6 Drawing the Contents of a VBO

We are now ready to draw the contents of the buffer, this requires two stages. First we are going to bind the buffer and describe contents of the buffer to the pipeline, then we are going to actually draw the contents of the buffer.

Add the following the code to **Render** function, just after the call to glClear

```
 //Make the new VBO active. Repeat here as a sanity check( may have changed
//since initialisation)
glBindBuffer(GL_ARRAY_BUFFER, triangleVBO);

//Establish its 3 coordinates per vertex with zero stride(space between elements) //
in array and contain floating point numbers
glVertexPointer(3, GL_FLOAT, 0, NULL);

//Establish array contains vertices (not normals, colours, texture coords etc)
glEnableClientState(GL_VERTEX_ARRAY);
```

Now that we bound and describe the contents of the buffer, we are ready to render. Add the following code to the **Render** function just after the above code but before the **SDL_GL_SwapWindow** call

```
//Swith to ModelView
glMatrixMode( GL_MODELVIEW );
//Reset using the Indentity Matrix
glLoadIdentity( );
//translate
glTranslatef( 0.0f, 0.0f, -6.0f );
//Actually draw the triangle, giving the number of vertices provided
glDrawArrays(GL_TRIANGLES, 0, sizeof(triangleData) / (3*sizeof(float))) ;
```

The first 3 lines are basically the same as when we originally had when we where drawing the triangle with the **glBegin** and **glEnd** functions. The last line is where we draw the contents of the buffer - the 1st parameter is the type of primitives we are drawing, 2nd parameter is the start index of the first vertex in the buffer(this can be used to offset into the buffer) and finally the last parameter is amount of vertices we are drawing(we calculate this from the total size of the vertices divided by the size of one vertex(3 * sizeof(float)).

3. Mid-Exercise Exercise

1. Draw multiple triangles(hint you don't need to add more vertices to the VBO)

4. Interleaved Vertices

Currently our vertices are made up just positional information but what happens to our vertices if we are going to send additional elements such as colours or normal. An approach to solve is to have one VBO for positions, one VBO for colours, one VBO for normal and so on. This means the layout of the vertices for the triangle would be **XYZ XYZ XYZ RGBA RGBA RGBA**. This can be fairly inefficient as you are fetching an element for a vertex from two places(and two different memory locations). It would be better to interleave the vertices so that the layout will be **XYZ RGBA XYZ RGBA XYZ RGBA XYZ RGBA**. This is more efficient as we are access the data serially from memory and also we are cutting down on the amount of storage(we only one VBO not two).

4.1 Vertex Structure

The first we have to do is define a structure which represents a vertex. Right click on the project and add a new header file(*.h) and call it **Vertex.h**

Now open up the header file and the following code to form a header guard

```
#ifndef Vertex_h
#define Vertex_h


#endif
```

A header guard ensures that the code in the file between **#define** and **#endif** will only get compiled once.

We are now going to declare out structure, this will have the interleaved format that we discussed in point 3. Insert this code between the **#define** and **#endif**

```
struct Vertex
{
   float x,y,z;
   float r,g,b,a;
};
```

Now switch back to **main.cpp** and add an include statement to the top file for **Vertex.h**.

4.2 Modify Vertex Array

We have changed how our vertices are represented on the Application side of the pipeline. This means we need to update our vertex array to match this layout and provide colours for each vertex.
To achieve this we are going to change the type of **float** array to our new **Vertex** type(see above) and we will need to modify the array layout somewhat.
We are going to use what is know as **Aggregate Initialisation**(http://en.cppreference.com/w/cpp/language/aggregate_initialization) This means that each instance of the structure can be initialized using a curly brace syntax, for example

{ {1.0f,1.0f,1.0f} , {0.0f,1.0f,1.0f,1.0f} }

Let us step through this a bit at a time.

{ {1.0f,1.0f,1.0f} , {0.0f,1.0f,1.0f,1.0f} **}**

The first set of brackets(in bold) denotes an element in the array(instance of a structure)

{ **1.0f,1.0f,1.0f** , 0.0f,1.0f,1.0f,1.0f }

The first three values map onto the x, y, z variables of our structure

{ 1.0f,1.0f,1.0f , **0.0f,1.0f,1.0f,1.0f** }

The last four values map onto the r, g, b, a variables of our structure

We are now ready to change the structure of the variable, carefully change the array to match the following(changes are in bold)

```
Vertex triangleData[]={ {0.0f,1.0f,0.0f, //x,y,z
                  1.0f,0.0f,0.0f,1.0f}, //r,g,b,a

                  {-1.0f,-1.0f,0.0f, //x,y,z
                  0.0f,1.0f,0.0f,1.0f}, //r,g,b,a

                  {1.0f,-1.0f,0.0f, //x,y,z
                  0.0f,0.0f,1.0f,1.0f}}; //r,g,b,a
```

4.3 Layout of a Vertex & Drawing Changes.

We have now significantly changed the structure of the vertices in the VBO and we need to tell the pipeline what to expect from us. Navigate to the **render** function and add/change the following code in bold

```
//the  3 parameter is now filled out, the pipeline needs to know the size of
//each vertex
glVertexPointer(3, GL_FLOAT, sizeof(Vertex), NULL);
//The last parameter basically says that the colours start 3 floats into
// each element of the array
glColorPointer(4, GL_FLOAT, sizeof(Vertex), (void**)(3 * sizeof(float)));

//Establish array contains vertices & colours
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);

glDrawArrays(GL_TRIANGLES, 0, sizeof(triangleData) / sizeof(Vertex)) ;
```

Build and run the application, you should see a multi-coloured triangle.

5. 3D

To move from a 2D space to 3D space we only require the addition of a new matrix which is pushed onto the **MODELVIEW** matrix stack. Navigate to the **render** function and the following before the **glTranslate** but after **LoadIdentity**

```
gluLookAt(0.0, 0.0, 0.0, 0.0, 0.0, -1.0f, 0.0, 1.0, 0.0);
```

This function will calculate what is know as the view matrix and push it onto the top of the matrix stack, we require that is done for every object which be rendered in 3D. You can think of this as your virtual camera which has the following properties

- 1st three parameters is the camera position in 3D space(x, y ,z)
- 2nd set of three parameters is the centre(look at point) in 3D space(x, y, z)
- Last set of three parameter is the Up axis of the camera(x,y,z)

6. Rendering a Cube

We are now going to a render a cube. A cube is made up **6** faces, each face is made up of **2** triangle, each triangles is made up **3** vertices and each vertex is made up of **7** floating point numbers(position and colours). This means that our cube will be made of **36** vertices, which gives a total of **1008** bytes for coloured cube. This is because a floating point number is **4** bytes in size, and one vertex(position & colour) is **28** bytes

6.1 Front Face

Change the coordinates of the vertex array to specify the coordinates of the front face(changes in bold!).

```
Vertex triangleData[]={
//Front
{ -0.5f, 0.5f, 0.5f,
   1.0f, 0.0f, 1.0f, 1.0f },// Top Left

{ -0.5f, -0.5f, 0.5f,
   1.0f, 1.0f, 0.0f, 1.0f },// Bottom Left

{ 0.5f, -0.5f, 0.5f,
   0.0f, 1.0f, 1.0f, 1.0f }, //Bottom Right

{ 0.5f, 0.5f, 0.5f,
   1.0f, 0.0f, 1.0f, 1.0f },// Top Right

{ -0.5f, 0.5f, 0.5f,
   1.0f, 0.0f, 1.0f, 1.0f },// Top Left

{ 0.5f, -0.5f, 0.5f,
   0.0f, 1.0f, 1.0f, 1.0f }, //Bottom Right
};
```

Build and run the application and we should see a square displayed on the screen.

6.2 Back Face

We are now ready to specify a back face.

Add the following code after the front face vertices but before the last closing curly brace

```
//back
{ -0.5f, 0.5f, -0.5f,
   1.0f, 0.0f, 1.0f, 1.0f },// Top Left

{ -0.5f, -0.5f, -0.5f,
   1.0f, 1.0f, 0.0f, 1.0f },// Bottom Left

{ 0.5f, -0.5f, -0.5f,
   0.0f, 1.0f, 1.0f, 1.0f }, //Bottom Right

{ 0.5f, 0.5f, -0.5f,
   1.0f, 0.0f, 1.0f, 1.0f },// Top Right

{ -0.5f, 0.5f, -0.5f,
   1.0f, 0.0f, 1.0f, 1.0f },// Top Left

{ 0.5f, -0.5f, -0.5f,
   0.0f, 1.0f, 1.0f, 1.0f }, //Bottom Right
```

Build and run the application, we may not see this at first but you can adjust the translation to shift the cube left or right to see the face.

6.3 Left, Right, Top & Bottom Faces

Using the information above see if you can complete the other faces on the cube. I would suggest that you try to plot it out in paper first!

If you have any problems with this, please ask your lab demonstrator for help.

7. Element Buffer

An Element Buffer(often known as an Element Array Buffer) holds indices into a vertex buffer. This allows us to make an optimization by sending less vertices per model. Lets illustrate it with an example, if we are going to render a cube using a VBO and EBO we only need to send 8 vertices and 36 indices. This means the total amount of memory used by the cube will be **244** bytes for the vertices and **144** bytes for the indices, which makes a total of **388** bytes. This gives us saving of **620** bytes, while this doesn't sound a lot imagine if our game was more complex.

7.1 Global EBO Integer ID

Like all buffers, you need to store an integer which will be used to refer to the EBO. Add the following code snippet to the global variables section of **main.cpp**

```
GLuint triangleEBO;
```

7.2 Change Vertex Data

Since we are going to store triangle indices, we now only need to store each corner of the cube. Change the vertex array so that it looks the following

```
Vertex triangleData[]={
//Front
{ -0.5f, 0.5f, 0.5f,
   1.0f, 0.0f, 1.0f, 1.0f },// Top Left

{ -0.5f, -0.5f, 0.5f,
   1.0f, 1.0f, 0.0f, 1.0f },// Bottom Left

{ 0.5f, -0.5f, 0.5f,
   0.0f, 1.0f, 1.0f, 1.0f }, //Bottom Right

{ 0.5f, 0.5f, 0.5f,
   1.0f, 0.0f, 1.0f, 1.0f },// Top Right


//back
{ -0.5f, 0.5f, -0.5f,
   1.0f, 0.0f, 1.0f, 1.0f },// Top Left

{ -0.5f, -0.5f, -0.5f,
   1.0f, 1.0f, 0.0f, 1.0f },// Bottom Left

{ 0.5f, -0.5f, -0.5f,
   0.0f, 1.0f, 1.0f, 1.0f }, //Bottom Right

{ 0.5f, 0.5f, -0.5f,
   1.0f, 0.0f, 1.0f, 1.0f },// Top Right

};
```

7.3 Define Indices

We are ready to define the indices, add the following code just after the **vertex array**, please note these indices represent an index of a vertex in the VBO!

```
GLuint indices[]={
   //front
   0,1,2,
   0,3,2,

   //left
   4,5,1,
   4,1,0,

   //right
   3,7,2,
   7,6,2,

   //bottom
   1,5,2,
   6,2,1,

   //top
   5,0,7,
   5,7,3,

   //back
   4,5,6,
   4,7,6
};
```

7.4 Creating and filling the EBO

The indices are defined, we can now create our EBO. Navigate to **initGeometry** and add the following code after creation of the VBO.

```
    //create buffer
    glGenBuffers(1, &triangleEBO);
    //Make the EBO active
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, triangleEBO);
    //Copy Index data to the EBO
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
GL_STATIC_DRAW);
```

7.5 Cleanup

The above calls have allocated memory, we need to reclaim this memory. Add the following to the cleanup function, this should go before the deletion of the VBO.

```
glDeleteBuffers(1, &triangleEBO);
```

7.6 Render with EBO

Using an EBO will change how we render, in addition to binding a VBO we have to bind the EBO. Add the following code to the **render** function just after we bind the VBO.

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, triangleEBO);
```

In order to draw using VBOs and EBOs our draw call has to use **glDrawElements** instead of **glDrawArrays.** Replace glDrawArrays with the following

```
glDrawElements(GL_TRIANGLES, sizeof(indices)/sizeof(GLuint),
        GL_UNSIGNED_INT,0);
```

Exercise

1. Draw an additional cube
2. Move one of the cubes using the keyboard
3. Rotate the other cube using the keyboard
4. Move the camera using the keyboard
5. Attempt to move the viewpoint(look) using the mouse