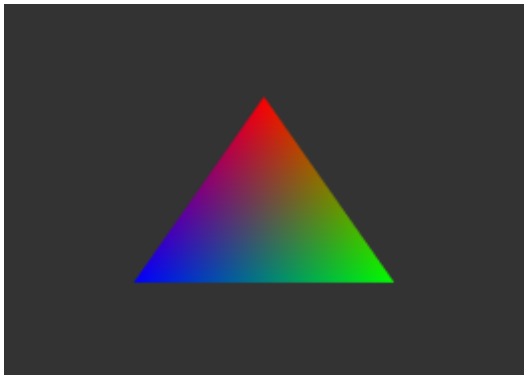
Tutorial 1: Your First Triangle!



Summary

For your first dabble in OpenGL, you are going to create the graphics programming equivalent of 'Hello World' - outputting a single coloured triangle. It doesn't get any simpler than that!.

New Concepts

Vertex Buffers, Vertex Array Objects, Vertex Attributes, Shader Programs, Vertex Shader Objects, Fragment Shader Objects, Symbolic Constants, Object Names

Introduction

So, a triangle - Sounds boring, yes? Well, it is! But drawing a triangle on screen will teach you the basics of OpenGL, and get you well on your way to doing more advanced graphical rendering. In order to write graphical programs using OpenGL, you'll need to know how to handle 3 different types of data - vertex data, texture data, and shaders. We'll handle texture data in a later tutorial; for now we'll concentrate on how to handle vertex data and shaders. OpenGL provides functions for copying vertex data, and compiling shaders, but provides little in the way of an 'Object Oriented' way to handle such data - there's no class structure to neatly encapsulate functionality. For this reason, this first tutorial is going to show you how to write your own *Mesh* class to handle vertex data, and a *Shader* class, to load in and handle shaders. This tutorial series will then use these as a basis to show you more advanced OpenGL functionality, so although a triangle on screen may not sound very interesting, this is an important tutorial!

Vertex Data

Vertices are the points in space that make up the geometry that represent everything in a modern video game - everything from the enemies on screen, to the in-game HUD - they're all made up of vertices! Each of the vertices in a geometry mesh will have a number of *attributes*, such as its position, colour, and texture coordinate. In order to draw a mesh on screen, this vertex data must first be *buffered* by copying it to graphics memory. Each mesh in an OpenGL application will have one of these *Vertex Buffer Objects* for each type of vertex attribute, grouped together into a single *Vertex Array Object*. This array object also stores information on what each buffer contains, and how each vertex buffer is to be accessed, such as what data format it is (i.e whether it is an **integer**, **float**, etc), and how many bytes each vertex element takes up. Once all of the vertex data has been buffered into

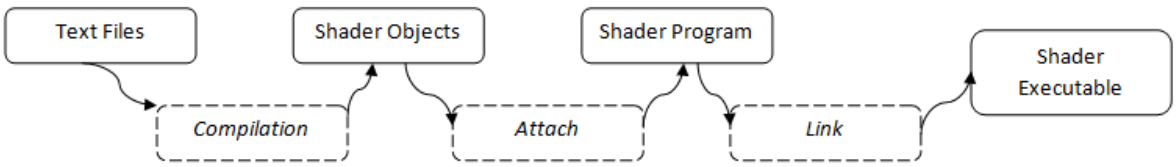
the RAM of the graphics card, and the Vertex Array Object created, it can be used by OpenGL to draw a mesh. It's worth noting that neither VBOs or VAOs define what the vertex data *represents* - it could be a group of triangles, a set of lines, convex polygons, or even just a cloud of points.

Shaders

Shaders are short programs that run on the graphics card, and turn your vertex data into a final image on screen. A single shader executable is made up of separate components - vertex programs, fragment programs, and maybe even geometry programs. Vertex programs take in the vertices of your mesh VBOs, and perform operations on them, before passing the resulting data on to fragment shaders. These operations usually consist of *transforming* the vertex data by several transformation matrices - you'll see how to do this in the next tutorial. Fragment shaders take in the results of this process, and turn them into *fragments* - colour data that will be written to the current OpenGL colour buffer. Operations like sampling from a texture, or performing lighting calculations are the domain of the fragment shader. Some shader executables also have a *geometry* shader. These shaders sit between vertex and fragment shaders, taking in *primitive* data (arrays of vertices that represent lines, or triangles) from the vertex shader, and generating *new* geometry from them - so each inputted triangle could be transformed into a number of new triangles, all sent to the fragment shader for colouring.

In OpenGL, shaders are written in GLSL. This is a C-like language, with datatypes and functions specific to graphical rendering. They can have functions, **if** statements, **for** loops, and so on, just like the C++ programs you have been writing, but they what they *can't* have are *classes*. Shaders specialise in performing operations on floating point *vectors* - structures with multiple data elements. For example, shaders can very quickly perform dot and cross products on 4-component vectors, as well as multiply them together - handy for blending vertex colours and transforming vertex positions! For this reason, GLSL has additional in-built datatypes to represent vectors, beyond the **floats** and **integers** you are used to. The **vec2**, **vec3** and **vec4** GLSL datatypes represent vectors of 2, 3 and 4 floating point components, respectively. Their floats can be accessed just like member variables of a C struct or class, using the identifiers x y z and w - note that w is the *fourth* component of a **vec4**, not the first! You'll see how to perform operations on these vector datatypes as this tutorial series progresses, including the multiplication of vectors by matrices, via the **mat4** datatype.

To use shaders in OpenGL applications, their sourcecode must be loaded in and then *compiled*, just like your C programs must be compiled before they can run. This compilation process takes vertex, fragment, or geometry shader sourcecode, and creates a *Shader Object*. These shader objects are then *attached* to a *Shader Program* - a sort of container for managing Shader Objects. Then, the Shader Object is *linked* together into a final *Shader Executable* - this linking process ties the output of one shader object to the input of the next, creating a final cohesive program for graphical rendering.



Example program

In this tutorial, you'll create 3 classes, and two text files. The first two classes, *Mesh* and *Shader*, are going to be added to the *nclgl* Visual Studio project, so that every tutorial you write will be able to use them. The third class you'll be making is a *Renderer* class, which will go in the Tutorial 1 project, and is a subclass of the NCLGL project *OGLRenderer* class. The two text files will contain the vertex and shader code for this tutorial, and will go in the *Shaders* folder of the nclgl download - we'll be reusing our shader code in later tutorials, too!

The *Mesh* class

Now, to put what you've just learned on vertices into practice! Every tutorial in this series is based around extending the existing nclgl classes, which provides a basic OpenGL creation class, as well as some vector math and input classes. We're going to need a new class to this Framework, to encapsulate the geometry rendering functionality of OpenGL, so add a class called **Mesh** to the *Shared* project in the Visual Studio solution provided for these tutorials, and input the following code into its header file.

Header file

When OpenGL generates a VAO or VBO, you get an **object name** - a numerical identifier for that object. So our *Mesh* class has **protected** member variables to store these object names, of type **GLuint** - an OpenGL **typedef**, which defines an **unsigned int** data type. There's also member variables to hold how many vertices the mesh has, and what its draw *type* is - this could be triangles, points, lines, or any other draw primitive that OpenGL supports.

The *bufferObject* array uses an **enum** to define its size, which might seem unusual. By default, **enums** are named **integers**, incrementing from a starting value of 0 - so the *MAX_BUFFER* enum compiles to a value of 2, enough to use as a size in an array to store the vertex position and vertex colour data we're using in this tutorial, with the enums prior to it equating to valid indices to the *bufferObject* array. In later tutorials, we'll extend the *Mesh* class to hold texture coordinates, normals, and other things - if we add an enum for each of these extra attributes *before MAX_BUFFER*, it will always equate to an array size large enough to store all of the Vertex Buffer names we need - handy! Also, to store our vertices and colours, we have two pointers, which we will initialise with data later.

Finally, our header file has a **static** public function, *GenerateTriangle*, a **virtual** function *Draw*, and a **protected** function *BufferData*. Their names should make them pretty self explanatory - *Draw* draws the mesh, *BufferData* copies the mesh vertex data into graphics memory, and *GenerateTriangle* returns a **pointer** to a *Mesh*, containing VAO and VBO data for a coloured triangle. This tutorial series will be using such a triangle for the first few tutorials, so making the code that will generate it available in the shared Framework is a useful time saver.

```
1  #pragma once
2  #include "OGLRenderer.h"
3
4  enum MeshBuffer {
5      VERTEX_BUFFER, COLOUR_BUFFER, MAX_BUFFER
6  };
7  class Mesh  {
8  public:
9      Mesh(void);
10     ~Mesh(void);
11
12     virtual void   Draw();
13     static Mesh*   GenerateTriangle();
14
15 protected:
16     void     BufferData();
17
18     GLuint   arrayObject;
19     GLuint   bufferObject[MAX_BUFFER];
20     GLuint   numVertices;
21     GLuint   type;
22
23     Vector3*    vertices;
24     Vector4*    colours;
25 };
```

Mesh.h

Class file

Now for the cpp file. In our **constructor**, we use our handy **enum** to initialise all of our Vertex Buffer Objects to 0 - in OpenGL a name of 0 is a little bit a null pointer, so it will be ignored. We then create a name for our Vertex Array Object using **glGenVertexArrays**, which takes in a **reference** to the **unsigned int** we want the generated name to go into. This is the general mechanism for how OpenGL handles name generation - Vertex Buffers, textures, and other OpenGL data structures you'll come across later, all use a similar function signature. We also set our draw type to **GL_TRIANGLES** - this is an example of an OpenGL **Symbolic Constant**, which is a **define** with a specific value that equates to some OpenGL functionality - in this case, that the vertex data should be rendered as triangles.

```
1  #include "Mesh.h"
2
3  Mesh::Mesh(void)  {
4     for(int i = 0; i < MAX_BUFFER; ++i) {
5        bufferObject[i] = 0;
6     }
7     glGenVertexArrays(1, &arrayObject);
8
9     numVertices   = 0;
10    vertices      = NULL;
11    colours       = NULL;
12    type          = GL_TRIANGLES;
13 }
```

Mesh.cpp

The **destructor** for the *Mesh* class **deletes** our VAO and VBOs, using the appropriate OpenGL delete functions, which just like the generation functions, take a reference to the names you wish to **delete**.

```
14 Mesh::~Mesh(void) {
15    glDeleteVertexArrays(1, &arrayObject);
16    glDeleteBuffers(MAX_BUFFER, bufferObject);
17    delete[]vertices;
18    delete[]colours;
19 }
```

Mesh.cpp

The *GenerateTriangle* static function will return a **pointer** to a *Mesh*, with data in its VAO and VBOs that will draw a coloured triangle. It does so by initialising a new *Mesh* on the heap (on line 21), setting its number of vertices to 3, and then initialising its *vertices* and *colours* pointers with data. You should be able to see how the colours equate to *red*, *green*, and *blue*, respectively, and how the vertices of the triangle are in *top*, *bottom right*, *bottom left* order. The geometry is now in system memory, but to use it in OpenGL, it must be copied into VBOs. This is achieved by a call to *BufferData*. With the data copied, a **pointer** to the new triangle can be returned.

```
20 Mesh* Mesh::GenerateTriangle()   {
21    Mesh*m = new Mesh();
22    m->numVertices = 3;
23
24    m->vertices = new Vector3[m->numVertices];
```

```
25    m->vertices[0] = Vector3(0.0f,   0.5f, 0.0f);
26    m->vertices[1] = Vector3(0.5f,  -0.5f, 0.0f);
27    m->vertices[2] = Vector3(-0.5f, -0.5f, 0.0f);
28
29    m->colours = new Vector4[m->numVertices];
30    m->colours[0] = Vector4(1.0f, 0.0f, 0.0f,1.0f);
31    m->colours[1] = Vector4(0.0f, 1.0f, 0.0f,1.0f);
32    m->colours[2] = Vector4(0.0f, 0.0f, 1.0f,1.0f);
33
34    m->BufferData();
35    return m;
36 }
```

Mesh.cpp

Here's how the *BufferData* function works. It begins by *binding* the Vertex Array of the *Mesh*, as created in the **constructor**. What does binding an object name do? Binding is another general OpenGL concept - most OpenGL functions don't take in object names, they just perform their functionality using whatever object name is bound to their relevant element - so texture functions will perform on the currently bound texture, or as in this case, Vertex Array functionality will be be performed on the newly bound Vertex Array Object.

Then, starting on lines 39 and 46 we go through the process of buffering the data we created in *GenerateTriangle* into graphics memory, for the vertex positions and colours, respectively. On line 39, we generate a new VBO, storing its name in the first index of our *bufferObject* array. Then we *bind* it, which serves a dual purpose - it means all Vertex Buffer functions are performed on our Vertex Buffer, and also assigns that VBO to the currently bound Vertex Array Object, neatly tying our VBOs into a single VAO.

The **glBufferData** function is what actually copies data into graphics memory - of its 4 parameters, the first isn't too interesting, but the other 3 take a little bit of explaining. Just like the **memcpy** function you may have used in the C++ tutorials, **glBufferData** needs to know how large the data we're copying is (in this case 9 **floats**), a **pointer** to the start of the data to copy, and a *hint* to inform OpenGL how you expect the data to be used - either *dynamically* updated, or loaded once as *static* data. In the case of our triangle, we're going to buffer it once and then never modify it, so we're going to use the **GL_STATIC_DRAW** symbolic constant as our hint. These hints allows our graphics card driver to handle our data more effectively, but don't place any actual restrictions on how we can use our data - we can define data as dynamic and then never update it if we really want to.

```
37 void  Mesh::BufferData()  {
38    glBindVertexArray(arrayObject);
39    glGenBuffers(1, &bufferObject[VERTEX_BUFFER]);
40    glBindBuffer(GL_ARRAY_BUFFER, bufferObject[VERTEX_BUFFER]);
41    glBufferData(GL_ARRAY_BUFFER, numVertices*sizeof(Vector3),
42                 vertices, GL_STATIC_DRAW);
43    glVertexAttribPointer(VERTEX_BUFFER, 3, GL_FLOAT, GL_FALSE, 0, 0);
44    glEnableVertexAttribArray(VERTEX_BUFFER);
45    if (colours)  { //Just in case the data has no colour attribute...
46       glGenBuffers(1, &bufferObject[COLOUR_BUFFER]);
47       glBindBuffer(GL_ARRAY_BUFFER, bufferObject[COLOUR_BUFFER]);
48       glBufferData(GL_ARRAY_BUFFER, numVertices*sizeof(Vector4),
49                    colours, GL_STATIC_DRAW);
50       glVertexAttribPointer(COLOUR_BUFFER, 4, GL_FLOAT, GL_FALSE,0,0);
51       glEnableVertexAttribArray(COLOUR_BUFFER);
52    }
53    glBindVertexArray(0);
54 }
```

Mesh.cpp

With our VBO data copied into graphics memory, we can set how to access it, by modifying the VAO. **glVertexAttribPointer** tells OpenGL that the vertex attribute has 3 **float** components per vertex, while **glEnableVertexAttribArray** enables it. Note how the **VERTEX_BUFFER** enum is used throughout - it means we never have to actually look up what index to use for which type of data when setting attributes or buffering data, the **enum** handles it, and provides a neat visual hint as to what data is being modified.
We do the same process again on line 46 for the vertex colour data - note the differing size set in **glBufferData** and **glVertexAttribPointer**. Finally, we *unbind* our VAO, and **return** the newly created *Mesh*. Unbinding is optional, but recommended - it ensures no other function accidentally modifies the wrong VAO state!

Finally, we want a function to draw our *Mesh*. Once the Vertex Array and Vertex Buffers are set up, drawing them is easy - we just *Bind* the VAO, and use the **glDrawArrays** OpenGL function, which has parameters that set what type of primitive to draw (remember how we set it to **GL_TRIANGLES**?), the first vertex to draw (generally the first one - index *0*), and how many vertices to draw (usually all of them, so the value we store in *numVertices*).

```
55 void Mesh::Draw() {
56    glBindVertexArray(arrayObject);
57    glDrawArrays(type, 0, numVertices);
58    glBindVertexArray(0);
59 }
```

Mesh.cpp

The *Shader* class

As with the *Mesh* class, we're going to add a *Shader* class to the *Shared* project. This class is going to encapsulate all of the tricky bits of loading in and using shaders - it's a lot of code, but it'll save us time in the long run!

Header file

There's nothing too surprising in the *Shader* class header - a **constructor**, a few function declarations, and some member variables to hold the object names of the shader objects and shader program - just like the *Mesh* class VBO object names. There's also 3 **defines** to use as indices into the objects array - we don't need any fancy **enums** this time, as there'll only ever be 3 types of shader object. Note how the **constructor** has an optional third parameter - most shader programs don't have a geometry shader object. We could have made a separate subclass for geometry-enabled shader programs instead, but for the saving of a single **unsigned int**, it's not really worth it!

```
1  #pragma once
2  #include "OGLRenderer.h"
3
4  #define SHADER_VERTEX   0
5  #define SHADER_FRAGMENT 1
6  #define SHADER_GEOMETRY 2
7
8  using namespace std;
9
10 class Shader   {
11 public:
12    Shader(string vertex, string fragment , string geometry = "");
13    ~Shader(void);
14
15    GLuint   GetProgram() { return program;}
16    bool     LinkProgram();
```

```
protected:
    void      SetDefaultAttributes();
    bool      LoadShaderFile(string from, string &into);
    GLuint    GenerateShader(string from, GLenum type);

    GLuint    objects[3];
    GLuint    program;

    bool      loadFailed;
};
```

Shader.h

Class file

The *Shader* class **constructor** takes in 3 strings as parameters, corresponding to the file names for the vertex, fragment, and geometry shaders, respectively. The **constructor** starts off by generating a new shader program object name, and then uses the *GenerateShader Shader* class function to generate shader objects for the vertex and fragment shaders; you'll see how shortly. In OpenGL 3, a shader program *always* has both a vertex and a fragment shader object, but geometry shader objects are optional. You should be able to see how the default third parameter of the constructor works now - it'll be seen as an 'empty' string, and geometry shader object construction will be skipped. Once compiled by the *GenerateShader* function, the shader objects are then *attached* to the shader program using the **glAttachShader** OpenGL function, ready to be *linked* into the final executable. We also call the **protected** function *SetDefaultAttributes*, which will be explained shortly.

```
#include "Shader.h"

Shader::Shader(string vFile, string fFile, string gFile) {
    program      = glCreateProgram();
    objects[SHADER_VERTEX]   = GenerateShader(vFile,GL_VERTEX_SHADER);
    objects[SHADER_FRAGMENT]= GenerateShader(fFile,GL_FRAGMENT_SHADER);
    objects[SHADER_GEOMETRY]= 0;

    if(!gFile.empty()) {
        objects[SHADER_GEOMETRY] = GenerateShader(gFile,
                                                  GL_GEOMETRY_SHADER);
        glAttachShader(program,objects[SHADER_GEOMETRY]);
    }
    glAttachShader(program,objects[SHADER_VERTEX]);
    glAttachShader(program,objects[SHADER_FRAGMENT]);
    SetDefaultAttributes();
}
```

Shader.cpp

Next up, our class **destructor**. It **deletes** our shader objects and program, allowing OpenGL to free up their memory, and reuse their object names. It is essentially the reverse of the **constructor** - we detach the previously attached shader objects, then **delete** the shader objects and shader program.

```
Shader::~Shader(void)   {
    for(int i = 0; i < 3; ++i) {
        glDetachShader(program, objects[i]);
        glDeleteShader(objects[i]);
    }
    glDeleteProgram(program);
}
```

Shader.cpp

The *GenerateShader* function used in the **constructor** creates the shader objects for the vertex and fragment shaders. It loads in shader object source code, compiles it, and returns an object name for the resulting shader object. It takes in two parameters - the name of the file to load, and the type of shader object to compile (either a vertex, fragment, or geometry shader object, denoted by the OpenGL symbolic constants **GL_VERTEX_SHADER**, **GL_FRAGMENT_SHADER** and **GL_GEOMETRY_SHADER**, respectively).

It uses another *Shader* class function on line 31, **LoadShaderFile**, to create a single long string containing the shader object source code. Assuming this is successful, We then get an OpenGL object name for the shader on line 37 using the OpenGL function **glCreateShader**, and use the OpenGL functions **glShaderSource** and **glCompileShader** to actually compile the shader source code. If successful, the function returns the shader object's name. If not, it outputs why compilation failed, including the exact line number of failure.

```
25 GLuint   Shader::GenerateShader(string from, GLenum type)   {
26    cout << "Compiling Shader..." << endl;
27
28    string load;
29    if(!LoadShaderFile(from,load)) {
30       cout << "Compiling failed!" << endl;
31       loadFailed = true;
32       return 0;
33    }
34
35    GLuint shader = glCreateShader(type);
36
37    const char *chars = load.c_str();
38    glShaderSource(shader, 1, &chars, NULL);
39    glCompileShader(shader);
40
41    GLint status;
42    glGetShaderiv(shader, GL_COMPILE_STATUS, &status);
43
44    if (status == GL_FALSE) {
45       cout << "Compiling failed!" << endl;
46       char error[512];
47       glGetInfoLogARB(shader, sizeof(error), NULL, error);
48       cout << error;
49       loadFailed = true;
50       return 0;
51    }
52    cout << "Compiling success!" << endl << endl;
53    loadFailed = false;
54    return shader;
55 }
```

Shader.cpp

For the **GenerateShader** function to work, it needs a string containing all of the required shader source code. We load this in from a text file, using the **LoadShaderFile** function. You've probably written very similar functions for text file loading before. It takes in two strings as parameters, a filename, and a destination string **reference**. The function uses the STL **ifstream** class to stream lines of text into our destination string. It'll return **true** if the file exists, or **false** if it doesn't, or otherwise cannot be opened. For good measure, it also outputs the destination string, so we can examine what it's loading.

```
56 bool  Shader::LoadShaderFile(string from, string &into)  {
57    ifstream file;
58    string      temp;
59
60    cout << "Loading shader text from " << from << endl << endl;
61
62    file.open(from.c_str());
63    if(!file.is_open()){
64       cout << "File does not exist!" << endl;
65       return false;
66    }
67    while(!file.eof()){
68       getline(file,temp);
69       into += temp + "\n";
70    }
71
72    file.close();
73    cout << into << endl << endl;
74    cout << "Loaded shader text!" << endl << endl;
75    return true;
76 }
```

Shader.cpp

In the constructor for our *Shader* class, we call the function *SetDefaultAttributes*. In order to correctly pass our vertex data to our shader, we must bind each vertex attribute to one of the input variables of our shader program. We do so using the **glBindAttribLocation** OpenGL function, which takes in a shader object name, attribute index, and shader input variable name as parameters. Note how we use the *Mesh* file **enums VERTEX_BUFFER** and **COLOUR_BUFFER** again - statically assigning attributes to specific indices has saved us quite a lot of attribute handling logic! It's quite good practice to keep attribute names the same in every shader, which also has the benefit of meaning you can bind every attribute in one function for all your shaders! If a shader doesn't contain a given attribute, the function doesn't do anything, so it's safe to call the same function for every shader you write.

```
1 void  Shader::SetDefaultAttributes()   {
2    glBindAttribLocation(program, VERTEX_BUFFER,  "position");
3    glBindAttribLocation(program, COLOUR_BUFFER,  "colour");
4 }
```

renderer.h

The final function links our attached shader objects into a shader *executable*. It returns **true** if the link is successful, or **false** if it fails - or if the loading of the shader objects failed. This function could be extended to include error output, by using a **switch** statement on the *code* variable, if we wanted to be a bit more verbose.

```
77 bool Shader::LinkProgram() {
78    if(loadFailed) {
79       return false;
80    }
81    glLinkProgram(program);
82
83    GLint code;
84    glGetProgramiv(program, GL_LINK_STATUS, &code);
85    return code == GL_TRUE ?  true : false;
86 }
```

Shader.cpp

The Renderer class

The *Renderer* class we're making in this tutorial is going to be pretty simple - Add it as a new class to the Tutorial1 project, with *OGLRenderer* as its parent class - the OGLRenderer constructor creates an OpenGL 3 *context* for you, and even comes ready with your *Mesh* and *Shader* header files included! It also already has a pointer to a *Shader*, called *curentShader*, which will be **deleted** in its **destructor**.

Header file

```
5  #pragma once
6  #include "./nclgl/OGLRenderer.h"
7
8  class Renderer : public OGLRenderer {
9  public:
10    Renderer(Window &parent);
11    virtual ~Renderer(void);
12    virtual void RenderScene();
13
14 protected:
15    Mesh* triangle;
16 };
```

renderer.h

Class file

The *Renderer* class **constructor** begins by using the **GenerateTriangle** function we wrote earlier to initialise our member variable *triangle*. We then initialise the *curentShader* member variable of the OGLRenderer class, passing the file names of the vertex and fragment source files we just created as parameters. We then set the *OGLRenderer* member variable *init* to **true** - our main function will check against this value to see if everything in the constructor worked as it should. The **destructor** is really short - all it has to do is **delete** the triangle member variable!

```
1  #include "Renderer.h"
2
3  Renderer::Renderer(Window &parent) : OGLRenderer(parent) {
4     triangle = Mesh::GenerateTriangle();
5
6     currentShader = new Shader(SHADERDIR"basicVertex.glsl",
7                                SHADERDIR"colourFragment.glsl");
8
9     if(!currentShader->LinkProgram()) {
10       return;
11    }
12
13    init = true;
14 }
15 Renderer::~Renderer(void)  {
16    delete triangle;
17 }
```

renderer.cpp

The last function in our tutorial's *Renderer* class is the most important one! **RenderScene** enables our shader, and draws our triangle. Note how it also *unbinds* the shader by sending a value of 0 to the **glUseProgram** function - for this simple tutorial this is a bit unnecessary, but it's good practice, just as with unbinding VAOs. As well as drawing our triangle, it also sets up the scene by

clearing the previous frame, and tells OpenGL to swap the front and back buffers around. **glClearColor** tells OpenGL what colour to clear the screen to - in this case a dark grey. This is better for debug purposes than the default colour of black, as OpenGL tends to draw objects black if it has a broken state, somewhere. The *OGLRenderer* base class already contains this function, but it was worth explicitly stating its purpose here. **glClear** is the function that actually clears the screen to the chosen colour, using the **GL_COLOR_BUFFER_BIT** symbolic constant. In later tutorials you'll see how other buffers can also be cleared simultaneously by **OR**ing several symbolic constants together.

```
18 void Renderer::RenderScene()  {
19    glClearColor(0.2f,0.2f,0.2f,1.0f);
20    glClear(GL_COLOR_BUFFER_BIT);
21
22    glUseProgram(currentShader->GetProgram());
23    triangle->Draw();
24    glUseProgram(0);
25
26    SwapBuffers();
27 }
```

renderer.cpp

Main file

The following code is the basic template that all of the tutorials will use for their main functions, with only minor variations. We begin by using a compiler **pragma** to load in the nclgl **static library**. This, along with the **include** on line 3, gives you access to all of the nclgl-provided code, including the *Window* class that handles all of the tedious OS window initialisation, allowing these tutorials to concentrate solely on OpenGL programming. We also **#include** the header file for our new *Renderer* class, so we can create an instance of our new *Renderer*.

In the **main** function, we start by declaring an instance of the nclgl *Window* class. It takes in 4 parameters - A **string** that is displayed at the top of the window, a window *width* and *height* (in pixels), and a **boolean** value to determine whether to be fullscreen or not. If for some reason our window fails to initialise, we make the program exit, via the **if** statement on line 6. Then, we create an instance of our *Renderer* class, passing it our *Window* class, so that it knows what to render to. As with the *Window* class, there is a function to check for successful initialisation.

```
1  #pragma comment(lib, "nclgl.lib")
2  #include "./nclgl/window.h"
3  #include "Renderer.h"
4  int main()  {
5     Window w("My First OpenGL 3 Triangle!", 800 , 600, false);
6     if(!w.HasInitialised()) {
7        return -1;
8     }
9     Renderer renderer(w);
10    if(!renderer.HasInitialised()) {
11       return -1;
12    }
13    while(w.UpdateWindow() &&
14    !Window::GetKeyboard()->KeyDown(KEYBOARD_ESCAPE)){
15       renderer.RenderScene();
16    }
17    return 0;
18 }
```

main.cpp

With our initialisation out of the way, we can enter our main loop. You've probably used something similar to this before - it's a **while** loop that continually updates our scene, until either the window is closed (the first half of the **while** loop break condition), or the user presses escape (the second half). In our loop, we're going to repeatedly call the **RenderScene** function of our *Renderer* class. Once the **while** loop is eventually broken out of, the function returns 0, ending the program execution.

Vertex Shader

Your first vertex shader isn't going to do a lot - it'll take in the vertex position and colour of each vertex in turn, and then simply output them. We start off with a GLSL **preprocessor definition**, to set out GLSL shader output to an OpenGL 3 compatible shader executable. Then, we define two input variables - note how they have the same names as the values we bound to the Vertex Array attributes on lines 9 and 10 of the *Renderer* class. Then we define an output **interface block** that consists of a single 4-component vector, to hold our vertices colour. You can think of **interface blocks** as being C++ structs that keep input and output data together.

Vertex and Fragment shaders must have a **main** function, just like C++ programs - and they're even defined in exactly the same way! Our vertex shader's **main** function only has two lines; the first sets the OpenGL output vertex position to the incoming vertex position, expanded out to the 4-component vector format OpenGL uses internally for vertices. The second line sets the vertex shader output to our incoming colour. That's everything!

```
1  #version 150 core
2
3  in  vec3 position;
4  in  vec4 colour;
5
6  out Vertex {
7     vec4 colour;
8  } OUT;
9
10 void main(void)   {
11    gl_Position   = vec4(position, 1.0);
12    OUT.colour    = colour;
13 }
```

basicVertex.glsl

Fragment Shader

Your first fragment shader is going to be even shorter - all it is going to do is output a fragment, with the colour sent to it from the vertex shader. As with our vertex shader, we'll start off by setting the *version* **preprocessor definition**, so that the compiler knows it should compile an OpenGL 3 compatible shader. Next up, we'll define our vertex input interface block. It contains the same values as the output interface block of our vertex shader. Then, we have our output value - OpenGL dissallows interface blocks for fragment output, so it is just a single **vec4** - this makes sense, as all a fragment shader *can* output are fragment colours. Finally, we create the required **main** function of the fragment shader. As you can see, all it does is set the fragment colour to the incoming colour, and return.

```
#version 150 core

in Vertex{
    vec4 colour;
} IN;

out vec4 fragColour;

void main(void)   {
    fragColour = IN.colour;
}
```

colourFragment.glsl

Tutorial Summary

All being well, you'll see something like the picture on page 1 when you run the program. If so, well done! You've just written your first OpenGL 3 program, and your first shaders. A triangle might not be too impressive, but you'll have learnt a lot of the basic concepts of OpenGL rendering. You should now know what a Vertex Array Object is, what a Vertex Buffer Object is, how to create them, how to send geometry to the graphics card, and finally how to actually draw something on screen using vertex and fragment shaders. You've also written two important classes, to encapsulate the functionality of shaders and mesh drawing. In the next tutorial, you'll build on these, and see how to move, scale and rotate the triangle using the *model matrix*, and add perspective to the scene using the *projection matrix*.

Further Work

1) Try making a few more shapes out of triangles - graph paper might come in handy here! For a 10 triangle mesh, what values would you use in the call to **glDrawArrays**? What about **glBufferData**? How would you draw a quad instead of a triangle?

2) The colours defined in the triangle VBOs were automatically interpolated per-pixel, to give the 'rainbow' effect. Investigate the **smooth**, and **flat** interpolation qualifers in GLSL.

3) Once Vertex Buffer data has been copied to the graphics hardware, it can be modified by mapping it to system address space. Investigate the **glMapBuffer** and **glUnmapBuffer** OpenGL commands - Try using them to generate random colours for the triangle every frame.

4) In this tutorial we used the **GL_TRIANGLES** symbolic constant with **glDrawArrays**. Investigate **GL_LINES**, **GL_TRIANGLE_STRIP**, and **GL_POINTS**. The OpenGL function **GLPointSize** might come in useful here. Does **GL_LINES** behave how you would expect it to? How could **GL_TRIANGLE_STRIP** help when drawing a quad?

5) When we buffered data to the VBO, we used the symbolic constant **GL_STATIC_DRAW**. Investigate the **GL_DYNAMIC_DRAW** and **GL_STREAM_DRAW** symbolic constants. What symbolic constant would you use to create deformable objects?