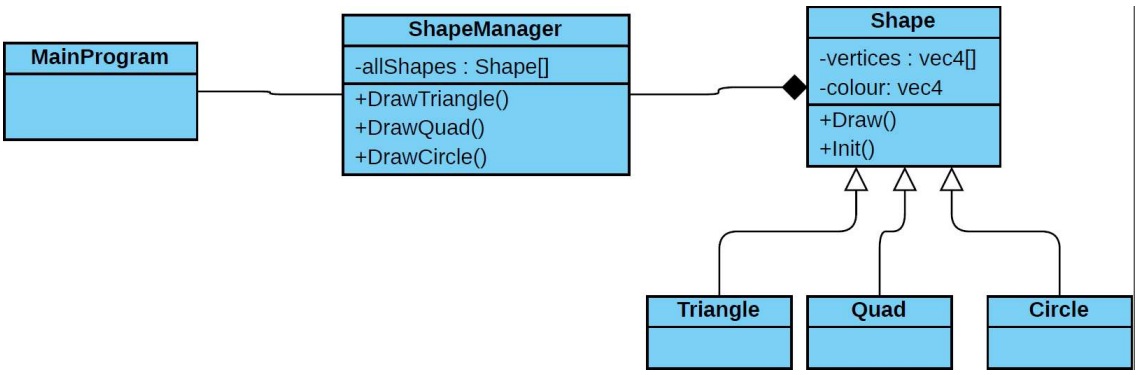
# 2D Shapes with OpenGL

I have been reading OpenGL Super Bible (the blue book) and decided to put my understanding to a test to see what I know. My aim was to create a simple program where I could draw simple 2D shapes on the screen by passing in vertexes using the console. The focus for me was passing the correct data from the C++ application to each shader using uniforms indexing. At index = 0 I would pass the position data of any shape and at index = 1 would be the colour of a shape.

Simple UML:




## Shaders

So, the first stage was to create the vertex and fragment shaders, so I knew where to pass the data of my shapes.

Vertex:

```
#version 450 core

layout (location = 0) in vec4 position;
layout (location = 1) in vec4 colour;

out vec4 col;

void main()
{
    col = colour;
    gl_Position = position;
}
```

Here I am expecting position to come in at index zero. Then using gl_Position apply that position to the vertex.

For colour I just pass it over to the fragment shader.

```
#version 450 core

in vec4 col;

out vec4 color;

void main()
{
    // Simply assign the color we were givenby the vertex shader to our output
    color = col;
}
```

The Application

```
void Triangle::init(GLuint* vao)
{
  glCreateBuffers(1, &m_verticesBuffer);
  glBindVertexArray(*vao);
  glBindBuffer(GL_ARRAY_BUFFER, m_verticesBuffer);

  glNamedBufferStorage(m_verticesBuffer,           // name of buffer
    6*sizeof(vmath::vec4),         // triangle has 3 vector4 points
    NULL,            // no sizeof(position) data
    GL_DYNAMIC_STORAGE_BIT);

  glVertexArrayVertexBuffer(*vao,   // vertex array attrib
    0,                  // first vertex buffer binding
    m_verticesBuffer,              // buffer object
    0,                  // start from the beginning
    sizeof(vmath::vec4)); // each vertex is a vec4

  glEnableVertexAttribArray(0);
  glEnableVertexAttribArray(1);

  glVertexArrayAttribFormat(*vao, // vertex array attrib
    0,                // first attribute
    4,                // four components
    GL_FLOAT,         // floating poiint data
    GL_FALSE,         // ignore normalised for floats
    0);               // first element of vertex

  //glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, sizeof(vmath::vec4), 0);
  glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, m_colours.size() * sizeof(vmath::vec4), (void*)(3*sizeof(vmath::vec4)));
}
```

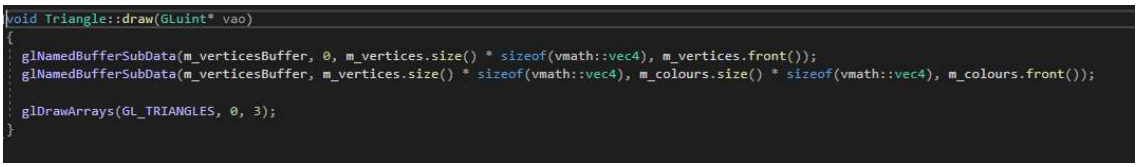
The init function is called when you want to create the memory space that will be used to pass into the shaders.

I create a buffer and bind it the to vertex array object passed in. Then I set the size of the buffer to be 6 * size of vec4 because there are 3 vertices in a triangle and then for each vertex, I set the colour. So, there for 6 vec4.

I then link the vertex (position) data to the vertex buffer, so the program knows its position in the buffer. I also do this for the colour further down

I pass the position data into vertex attrib at index = 0 and colour at index = 1 so I need to enable these.

This then set up an empty buffer ready to be passed into the vertex shader.



```
void Triangle::draw(GLuint* vao)
{
  glNamedBufferSubData(m_verticesBuffer, 0, m_vertices.size() * sizeof(vmath::vec4), m_vertices.front());
  glNamedBufferSubData(m_verticesBuffer, m_vertices.size() * sizeof(vmath::vec4), m_colours.size() * sizeof(vmath::vec4), m_colours.front());

  glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

Just before I draw the triangle, I fill buffer with the most recent vertex and colour data.