Introduction

OpenGL 3.x and OpenGl 4.x have since deprecated and started removing all matrix operations as part of the OpenGL library, which I believe is a really good thing for the computer graphics community, but for beginners can make things a little difficult to get up and running at first. What this means if you are coming from the previous OpenGL tutorials, is that you can no longer use calls such as glTranslate or glRotate. Even glLoadIdentity and setting the gluPerspective is now redundant.

When it comes to computer graphics in 3D, you typically have a number of matrices that make up your final scene. They are the projection matrix, model matrix and view matrix. You can also go into the texture matrix and normal matrix, etc, but for simply positioning shapes in the world I will stick with the more basic matrices. OpenGL 2.x and prior combined the model and view matrices into the modelview matrix and used a stack to let us add transformations matrices to the current modelview matrix. We could then pop off the top of the stack to a previous matrix if needed.

Personally I prefer to use the separate matrices so that you don't have to constantly monitor the one matrix, whilst also making it easier to use scene graphs and other scene management techniques. How you do this is up to you, but I am going to use multiple matrices.

Projection Matrix

The projection matrix is a 4x4 matrix that defines the 2D to 3D projection for a scene. This is often calculated by a pre-defined function, such as gluPerspective in previous versions of OpenGL. The library we are going to be looking at, provides a similar method.

View Matrix

The view matrix, which can also be known as the world matrix determines the position of the 'camera' in space. This can be thought of as a global matrix that affects everything and places everything relative to the location of the 'camera'.

Model Matrix

The model matrix is a local matrix that affects models on a more discrete basis. For example, if you want to translate, rotate or scale a specific object you will use that objects local matrix or model matrix.

Luckily in the wake of OpenGL dropping support for its own matrices, a mathematics library has been developed to replace it, called GLM. GLM is short for OpenGL Mathematics and has been designed to replicate the matrix functions found in GLSL along with giving us the scale, rotate and translate methods we are used to from OpenGL 2.x and lower. GLM can be found here: http://glm.g-truc.net/

Installation is simple; once you unzip the file you download, find the folder name "glm" and copy it into your IDE's header directory.

Once you have GLM installed and ready to use, move on to the coding section of this tutorial :D

Coding

The very first thing we need to do to get GLM working in our OpenGL application is to include the relevant header files. So go and open up main.h and add the following lines at the end of the file:

```
// GLM include files
#include <glm/glm.hpp>
#include <glm/gtc/matrix_projection.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

Now we should be able to use glm functions, stored in the glm namespace. To access them will be done similar to glm::function_name unless you add the line:
using namespace glm;
From which you can then just call function_name.

Let's set up the variables that will hold our matrices now. Open up opengl_3.h and in the private section of our class, created three variables, one for each of our matrices.

```
glm::mat4 projectionMatrix; // Store the projection matrix
glm::mat4 viewMatrix; // Store the view matrix
glm::mat4 modelMatrix; // Store the model matrix
```

Out of these three matrices, the projectionMatrix is the only matrix we will keep consistent throughout the entire application (of course, you can change this during your application if you are after a specific effect). So in our setupScene method, inside of opengl_3.cpp, we are going to create our projectionMatrix. To do so, we call a glm::perspective method, which works virtually identically to gluPerspective. It requires us to specify a fov (field of view) in degrees, followed by the aspect ratio for our window and finally the near and far planes for our scene. To do so, add the following line:

```
projectionMatrix = glm::perspective(60.0f, (float)windowWidth / (float)windowHeight,
0.1f, 100.f);  // Create our perspective projection matrix
```

Now that isn't so scary so far. And the following is hopefully just as nice to us. Everything else we will be changing in our C++ code will take place in the renderScene method, followed by modifying our vertex shader. Let's continue with our opengl_3.cpp file for now. The first thing we are going to do is after we clear our color, depth and stencil buffers, we are going to set our view and model matrices. I am going to set the view matrix, to translate back 5 units into the scene. This is the equivalent to our old glTranslate(0, 0, -5). The view matrix is going to be just for kicks at the moment, as we have no geometry, but it will scale geometry by a half.

```
viewMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, -5.f)); // Create
our view matrix which will translate us back 5 units
modelMatrix = glm::scale(glm::mat4(1.0f), glm::vec3(0.5f));  // Create our model
matrix which will halve the size of our model
```

After we have set these matrices, we then have to use them. To do so, we need to send them through to the shader that will be using them. In our code, we first bind our shader, just like we

would in the GLSL tutorials. We then need to link to the location of the variables we will be using inside of the shader. This is the locations for the perspectiveMatrix, viewMatrix and modelMatrix.

```
int projectionMatrixLocation = glGetUniformLocation(shader->id(), "projectionMatrix");
// Get the location of our projection matrix in the shader
int viewMatrixLocation = glGetUniformLocation(shader->id(), "viewMatrix"); // Get the
location of our view matrix in the shader
int modelMatrixLocation = glGetUniformLocation(shader->id(), "modelMatrix"); // Get
the location of our model matrix in the shader
```

Once we have these locations stored, we can go ahead and send through our matrices using glUniformMatrix4fv, just like we would in previous versions of OpenGL if we wanted to play with matrices. GLM allows us to access our matrices to send through to OpenGL easily. So add the following code after our locations.

```
glUniformMatrix4fv(projectionMatrixLocation, 1, GL_FALSE, &projectionMatrix[0][0]); //
Send our projection matrix to the shader
glUniformMatrix4fv(viewMatrixLocation, 1, GL_FALSE, &viewMatrix[0][0]); // Send our
view matrix to the shader
glUniformMatrix4fv(modelMatrixLocation, 1, GL_FALSE, &modelMatrix[0][0]); // Send our
model matrix to the shader
```

Shader.vert

Now we will move onto our vertex shader so that we can put these variables to good use. Inside of our vertex shader, we need the three matrix variables. So to create those, we are creating 3 mat4 variables.

uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;
uniform mat4 modelMatrix;

To use these matrices, you might remember a line similar to:
gl_Position = glModelViewProjectionMatrix * gl_Vertex;

Well in our new code, our line will be very similar:
gl_Position = projectionMatrix * viewMatrix * modelMatrix * vec4(in_Position, 1.0);

Where in_Position is the equivalent to gl_Vertex.

To wrap up, that is all that is required to switch from the OpenGL 2.x and lower style of matrix use, to the new developer-managed OpenGL 3.x and higher style of matrix use. I will point out that this tutorial still won't show anything more than a window on the screen. In the next tutorial we will finally be getting a square rendering on the screen. Well done so far!

If you have any questions you can comment below or email me at swiftless@gmail.com