# Transforms and Cameras

Now we know some background for all of this, let’s get to the code!

## Making a mesh class

1. First let’s make a class that will contain our mesh.

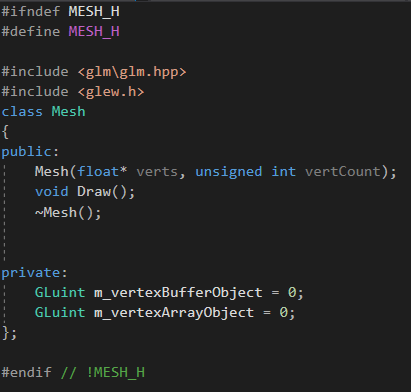
Create a new .cpp and .h for the mesh class.

We’re going to move somethings from the main function to this class. Specifically:

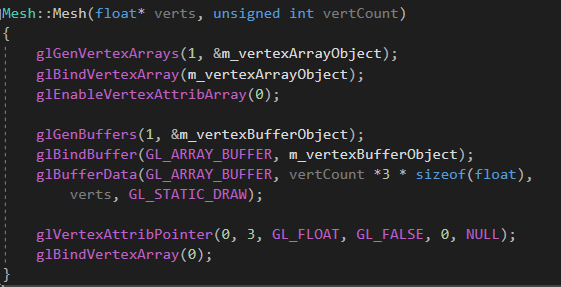
The gluint for vertexBfferObject and vertexArrayObject. Im going to rename these to m\_vertexBufferObject and m\_vertexArrayObject.

I’m going to alter the constructor to take a pointer to a float, called verts (this is going to be a pointer to our array of floats for our vertices, and a unsigned nit vertCount.

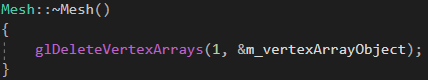
And I’m going to add a Draw method.



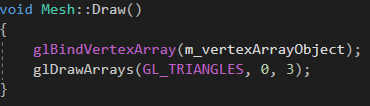
1. In the .cpp, in the mesh constructor, we’re going to put all the code for generating our vertex arrays and buffers along with their initialization code.



1. In the destructor, destroy the vertex array.



1. In the draw method, call the gl functions to draw the vertex array.



1. In the main function, make a new mesh object, pass it the vertex array and tell it how many vertices the object will have.



1. In the main loop, call the new mesh objects Draw() method.



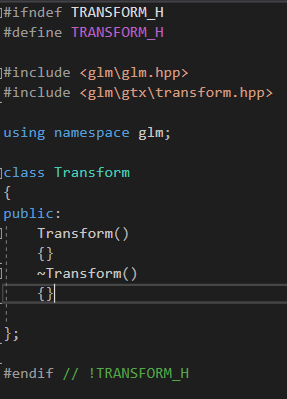
1. Run your program and make sure that everything still works.

## Transforms

1. Lets start with Transforms

Make a new class called Transform. (you can make a .hpp file if you want, instead of a .h and .cpp)

Enter the following



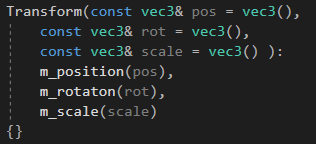
Obviously, we need glm but we also need transform from glm. This will include a bunch of really useful functions, like translate and rotate, so we don’t need to worry about writing them our self’s.

1. Define the following private variables



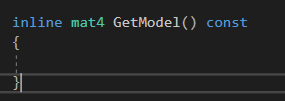
1. We need to set these variables to something sensible when we make a new transform (otherwise we might start off with junk and that wouldn’t be good).

So alter the constructor to take a vec3 for position, rotation and scale

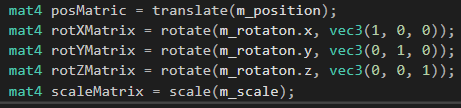


1. Now write the get and set methods for position, rotation and scale, so we can access them outside of this class.
2. Now we need to write a method to calculate our model matrix.

We’ll call it “GetModel”, it will return a mat4, be inline and const (so we can call it on const transform objects later).



1. Now we need to convert our transform values to matrices so we can work with them easily.



Translation and scale a simple, they just take the position and scale values and convert them from vec3’s to matrices.

Rotations are a little more interesting.

We must pass the values for our x rotation to the rotate function and a vec3 which defines which axis to apply the rotation to. Since we now have 3 different rotation matrices, one for each value, we need to combine them.

ORDER MATTERS!

Matrices are none commutative. This means that MatrixA \* MatrixB != MatrixB \* MatrixA.

As a result, we must combine them in the following order: X\*Y\*Z.



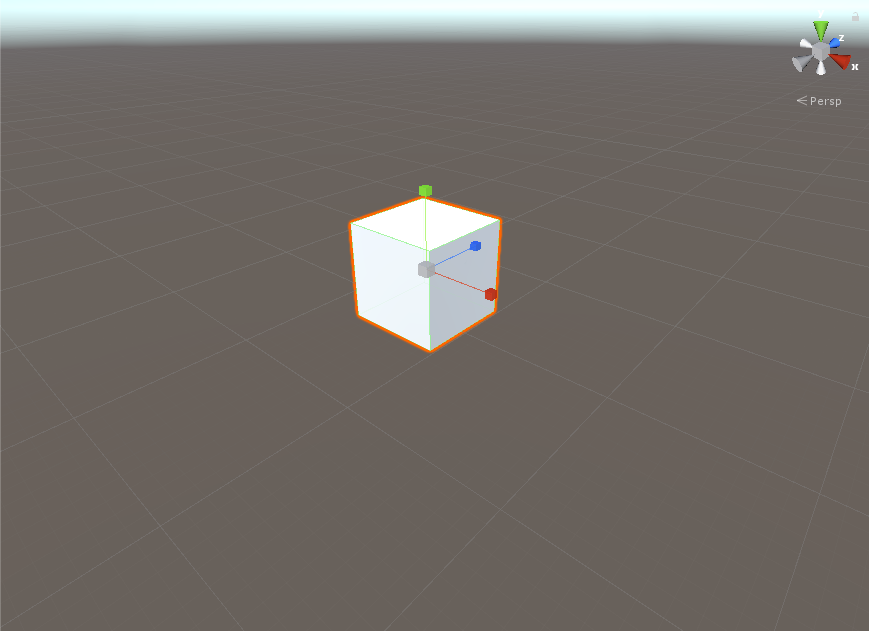
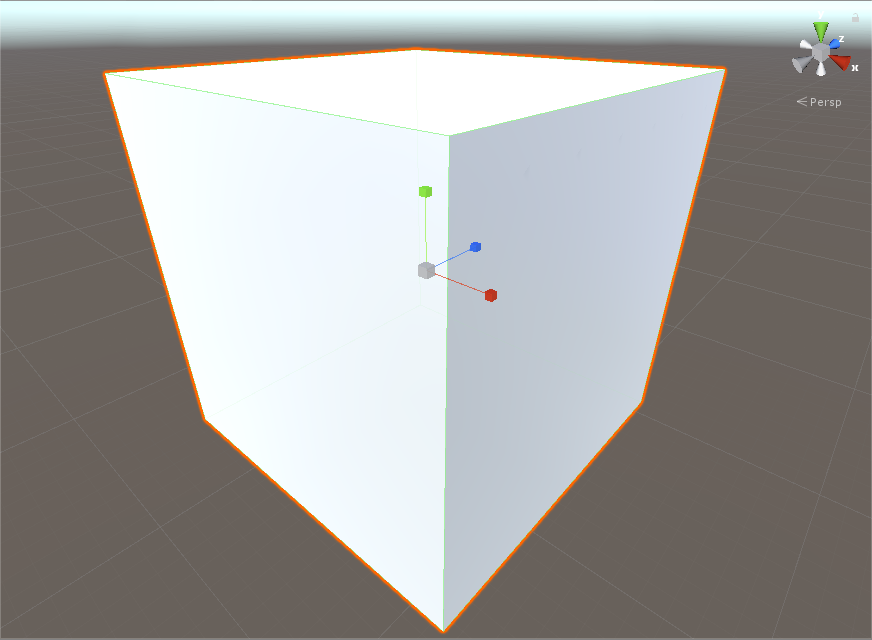
What you see IS NOT AN ERROR!

Welcome to C++, where multiplication of matrices works backwards! In this case you won’t see many problems if you combine in a different order but you will when you combine position, rotations and scale.

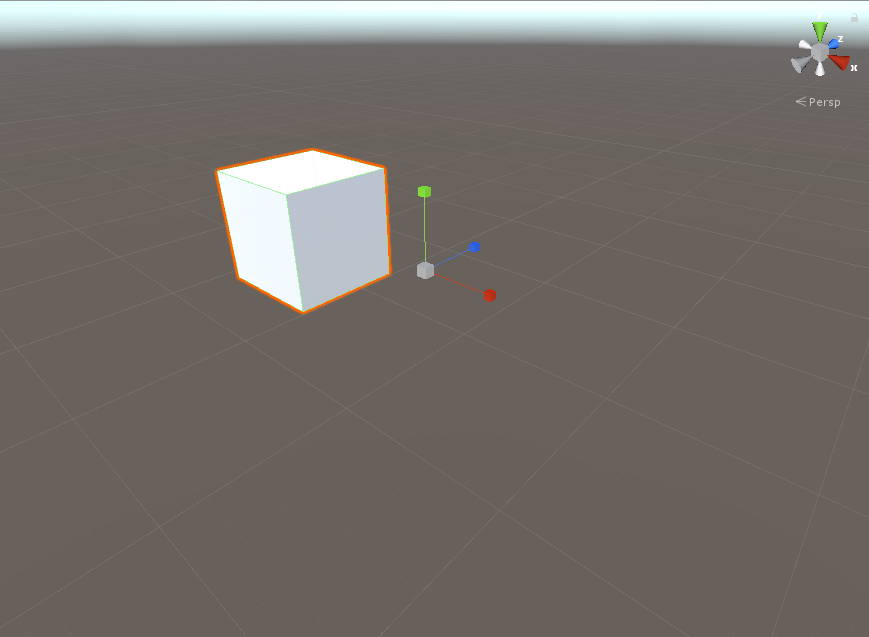
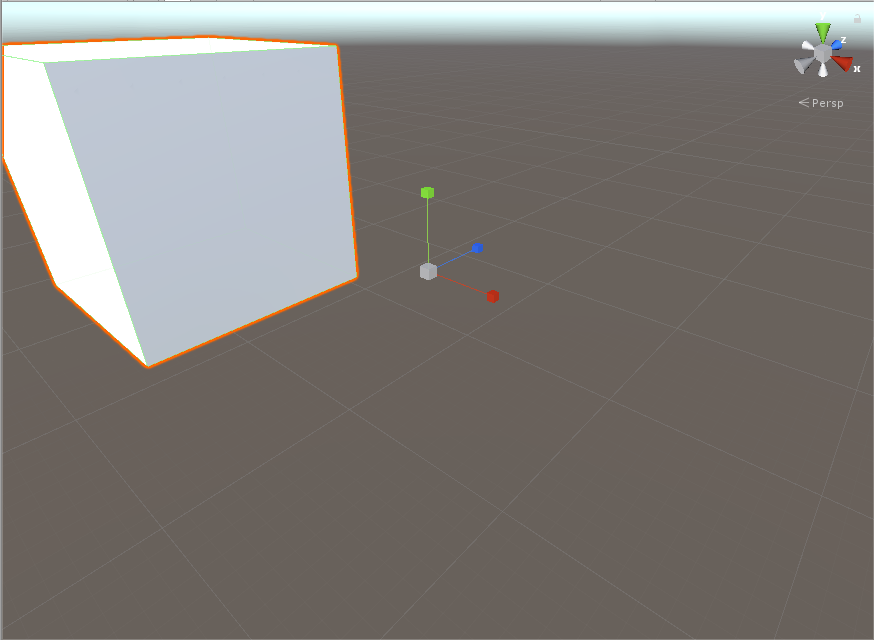


If you do this wrong, you will get strange results.

You need to remember that all transformations are calculated from world 0. So if you moved your mesh 10 units away and then attempted to scale it, the scale wold assume that your mesh is still at world 0 and attempt to scales it from them point.



(correct combination, scale then rotation then position)



(incorrect combination, position, rotation then scale)

1. That’s it. That’s our transform class.

Now we need to make a transform in our mesh class. Include the Transform.hpp and make a new transform object.

1. We need to alter our shader code to accept a transform model and apply it to our mesh.
2. In the vertex shader add the following, above the main function definition.



And alter the gl\_Position line so that our vertex position, vp, is combined with our model. Remember, order is important, and it should be vertex position \* model \* view \* projection, at least it will be when we’re finished. For now, just make it work with what you have.



The uniform mat4 is a special type of variable that will allow us to send the shader data as we update our game, making it perfect for verifying data like a transform.

1. Now all we need to do is send the data to it and the first step in doing that is to find it in the shader. Afterall, if we don’t know where it is, we can’t send data to it.

Before the draw() method call, but under the call to glUseProgram, enter the following.



The first line looks for a uniform variable, in our shader program, called “model”. This has to be spelt exactly the same way as the uniform variable in the shader code, otherwise modelLoc will equal -1.

Once we have found the location of the model uniform, we call glUniformMatrix4fv() and pass it modelLoc, 1, GL\_FALSE and a reference to our model data.

The first perimeter is the location of the uniform we want to send the data to.

The second perimeter is how many we want to send (this is useful for sending array data, but we’re not sending an array of mat4. We are sending just 1 mat4).

The third perimeter is transpose, set to false as we don’t want to transpose it.

And the last perimeter is the data we want to send. Since openGl will read the data as a string of bytes, up to the length of 1 matrix4, we just need to pass it the location of the first element of our model and it will do the rest.

1. Now run your code and make sure it still works again. At this point, nothing should have changed.
2. Now that it works, change the position of the triangle and run your code again.
3. Now make it rotate slowly as the program runs.
4. See how useful transforms are?! :D

## Cameras

Now we need to make a camera, so we can view our objects with perspective, it will also allow us to move through the world

Since this is part of your assessment, I will not tell you how to do this. But there is enough information in the presentation slides to work it out.

1. Create a camera class
2. It should contain the following:

* A transform
* 3 vector3’s, forward, up, right
* A method for updating these vectors.
* A mat4 for perspective projection
* A method for calculating and returning a mat4 for the view matrix

Hint: you will need to pass some data to the camera to make all this work. Use the constructor.