**IGB381 – Game Engine Technology**

**Workshop 2 (Week 3) – Vertex Rotations, Dynamic Colours and Vertex Scaling**

**Aim:**

The get started programming more complex shader effects and investigate ways to manipulate them using C# in Unity 5.

**Objectives:**

* Extended setup and execution of shaders
* Enable data flow to shaders via uniform and varying shader parameters
* Set up complex geometry for mesh objects in Unity 3D
* Demonstrate the vertex shader’s ability to update vertex locations and colours
* Demonstrate very simple shader management across different objects.

**Preparation:**

Ensure that you are familiar with all current Lecture and Workshop material prior to this workshop. Additionally, you will need to watch and download the [MyFirstProject video](https://www.youtube.com/watch?v=bDx81zT36e8) and accompanying project from Blackboard. This project will form the base for this week’s workshop material.

**Step 1: Vertex Rotation**

Ensure you have watched the previously mentioned video and download and opened the MyFirstProject Unity project. With this project we will modify the program so that a rotation around the object’s Z axis is performed within the vertex shader, effectively rotating the object around its Z axis without actually rotating the object itself. This trick is useful when we want objects and their textures to move around an object, but not necessarily move the object itself (e.g. wheels!).

To do this, we will need to pass into the vertex shader a uniform parameter (e.g. Theta) to signify the amount of rotation and construct a rotation matrix within the vertex shader to implement the rotation. Let’s begin by first opening the NormalMapping.cs file in Materials folder. Add a single private float like so:

private float Theta = 0.0f;

Additionally, in Update() let’s increment this value by deltaTime every frame, as well as communicate to the object’s material/shader to update a value by a similar name.

void Update () {

Theta += Time.deltaTime;

GetComponent<Renderer>().material.SetFloat("\_Theta", Theta);

}

This variable does not yet exist in our shader, so open up the NormalMapping.shader file in your editor of choice. Add the following line inside the Properties{} logic block:

\_Theta("Theta", Float) = 0.0

As well as this line within the SubShader’s Pass{} block, underneath the sampler2D texture declarations:

float \_Theta;

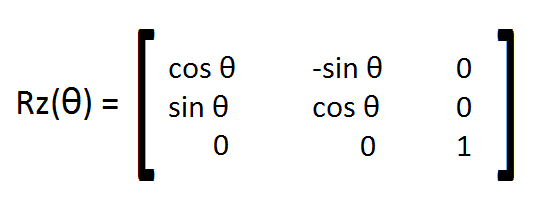
Now we have the information we need to update the texture’s rotation. For this exercise we will perform a Z rotation of the material’s texture within the vertex shader. Our vertex shader is called ‘VS\_NormalMapping’ as declared by our #pragma in our SubShader Pass. Find this vertex shader method so we can set up a rotation matrix that accepts Theta as a value to rotate slowly about the Z axis. This can be done several ways, but the most straightforward is to manually set up a 4x4 matrix of floats. Add this to the top of VS\_NormalMapping:

float4x4 rot;

After this, we should then reset this matrix to ensure all indices are for the current pass of the vertex shader:

rot = 0.0f;

Now we can perform the rotation. Recall that a 3x3 clockwise rotation on the Z axis looks like this:



You can access the individual indices of the matrix ‘rot’ using this format:

rot.\_11 = cos(\_Theta);

This will make the first row, first column (i.e. top left) be cosine with respect to theta. **See if you can fill out the rest of the matrix yourself using this format.** Note that you do not need to input indices that are 0 as they are that already. Additionally, you will need to add this line to make your object visible again:

rot.\_44 = 1.0;

Once you have a correctly completed clockwise Z rotation matrix, you will need to use it to update the actual vertices of the geometry the shader is affecting. To do this we will first need to multiply the original vertices by the new transformation matrix:

rot = mul(UNITY\_MATRIX\_MVP, rot);

Now that we have the difference, and can use this for outputting position of our actual vertices. To do this, tweak the following line in VS\_NormalMapping to use our rotation matrix instead:

output.pos = mul(**rot**, a\_Input.pos);

Assuming everything is set up correctly and you have no compile errors, your shader should now rotate the geometry of the GameObject slowly.

See if you can set up the project to rotate the vertices at a controlled rate, perhaps indicative of something like a wheel or conveyor belt. Additionally, see what happens when you play around with some of the values in the rotation matrix and some of the interesting warping that occurs.

**Step 2: Dynamic Colours**

The texture applied to the geometry in the scene can also be modified. This is typically done within the pixel shader component, rendering the object to camera using the visual settings you have provided. If you look within the fragment/pixel shader called PS\_NormalMapping you will notice that it eventually returns a colour. This colour is initially set up to be a static representation of the texture the geometry is provided, but some calculations are performed to make the diffuse, specular and normal components of the material visible upon the texture as well.

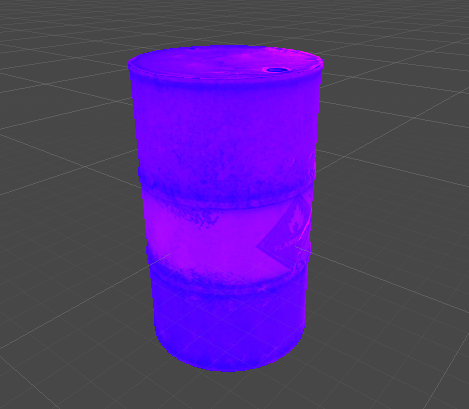
This means that we can adjust this colour, and potentially use more dynamic gameplay elements to do so. For instance, our Theta variable could be something in the real game world, like a disco light. If we wanted to make our barrel have a dynamic, shifting colour range, we could add the following code before it PS\_NormalMapping returns:

//Dynamic Colours

float4 adj = float4(sin(\_Theta), cos(\_Theta), tan(\_Theta), 1.0f);

colour = colour \* adj;

Here we are simply multiplying the colour by a float that is utilising our Theta variable in some way. The end result should be a semi-transparent surface recolouring of the object’s texture that changes quickly based on Theta, like so:



You may wish to experiment with colour, perhaps adjusting the float properties in a more interesting fashion, or seeing what happens when you add (instead of multiply) the adjustment to the colour. Show any interesting results to your classmates and/or workshop tutor!

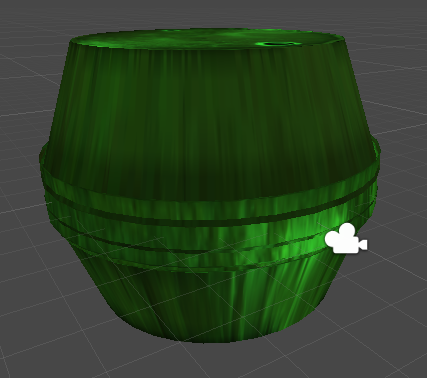
**Step 3: Vertex Scaling**

Heading back into our vertex shader, let’s investigate a way to transform the vertices directly, making them scale outwards from the centre of our object, without actually modifying the object’s in-game size. To do this, we can manipulate the function’s input parameter a\_Input, modifying the coordinate position of each vertex directly. Within VS\_NormalMapping, just before you calculate the output position of the vertices (i.e. output.pos = …) place and complete the following lines with your own formula that utilizes both Theta and a cosine equation:

//Scaling

a\_Input.pos.xyz += a\_Input.nor \* …

Using the vertex normal in the equation will allow us to uniformly scale the object in the desired direction, instead of translating or warping them about chaotically. You may wish to experiment with a formula that not only does this cleanly, but does so without flipping the normals over so we see the inside of the object. The end result should make the object look normal at the minimum cosine size and something like the following at maximum scale cosine size:



So where would this be useful? Scaling objects within a game engine is expensive, but if we only care about how the object looks and do not care about its physical properties, scaling objects via the GPU is very efficient. This could be done for objects such as balloons, making faces look chubbier or even just scaling visual effects in a non-uniform fashion (e.g. explosions).