

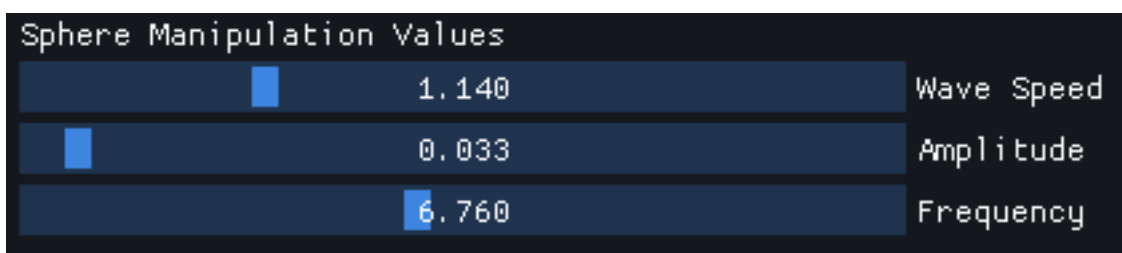
Introduction

The aim of this project was to create a graphic application demonstrating the ability to make use of the DirectX11 Graphical Pipeline in addition to five key graphical techniques. Developed using C++ in Visual Studio 2017, this application further makes use of the provided DirectX11 framework and ImGui API.

The scene that has been rendered by the application makes use of the five following techniques:

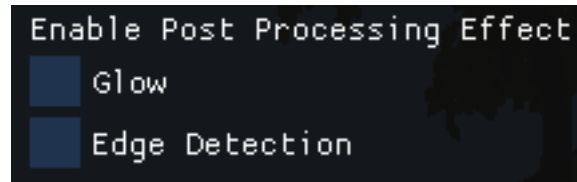
i) Vertex Manipulation

Vertex Manipulation has been demonstrated through the sphere object that has been positioned in the centre of the scene. The user has been provided with the ability to change values in the ImGui window which will alter the manipulation that is applied to the sphere mesh.



ii) Post Processing

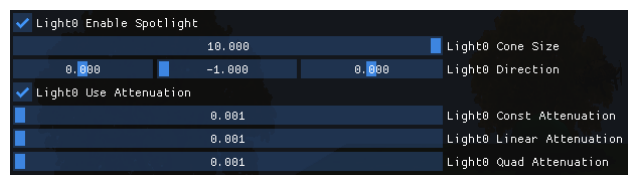
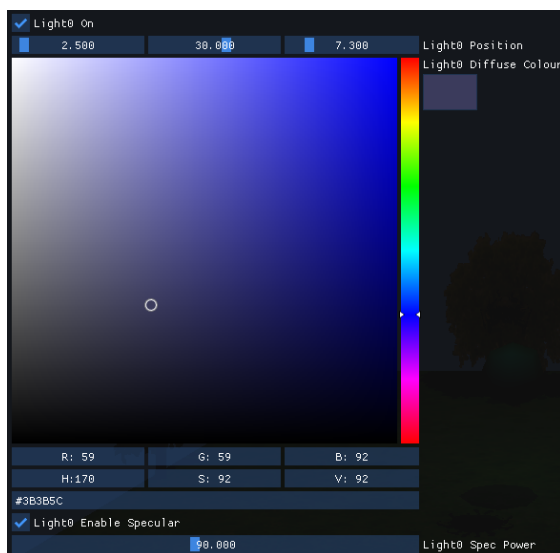
Two post processing techniques have been implemented in this application which are Glow and Edge Detection. Both of these techniques can be toggled on/off by the user in the ImGui window.



Post Processing Checkboxes in the ImGui Window

iii) Lighting

The scene consists of three lights. These lights have the possibility of producing multiple different types of lighting depending on the user's choice. Through controls in the ImGui window the user is given the ability to change attributes of all three lights. This includes the ability to: toggle them on/off, change the position, alter the colour, change the type of light, change direction, alter specular and attenuation component. If a light is not active then its controls in the ImGui window are not displayed.

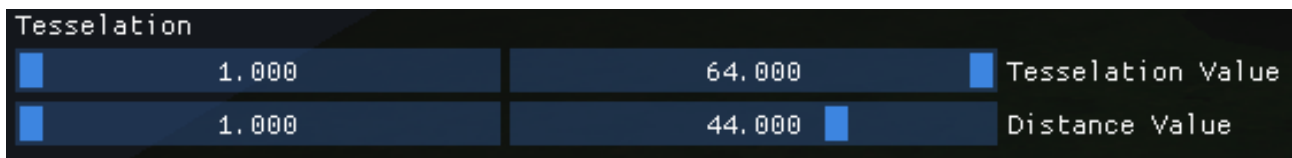


Light controls for light 0 in the ImGui Window

Each object in the scene is also shadowed correctly, based on the position of all active lights.

iv) Tessellation

Tessellation is another skill that can be observed on the sphere object placed within the scene. The tessellation is based on the distance of the vertex from the cameras position. Through controls in the ImGui window, the user is provided with a slider that can be used to change the distance at which tessellation starts and the strength of the tessellation.



Tessellation Value sliders in the ImGui Window

v) Geometry Shader

The three trees within the scene are created through the use of the geometry shader. First, the mesh that creates the tree begins with 3 points. Second, through utilising the geometry shader, these points are then turned into quads and textured to look like a tree. Finally, these quads are then billboarded towards the user to prevent the user going around the tree and viewing the back face which is not textured.

Techniques Used

Lighting

Three lights, that each have the ability to change between a point light or a spotlight, have been created and are demonstrated within the application. In order to achieve this goal, the *Light* class that was provided in the DXFramework had to be expanded, as at the start of development it didn't include all necessary attributes that the lights would need to use. The ability to toggle the light on/off, use attenuation, change cone size of spotlight and the ability to change the light from a point to a spot were all implemented.

All objects in the scene that are rendered use the same pixel shader, *uber_ps*. There are many advantages to using the one pixel shader for each object. First, it reduced the amount of repeated code in the project. If there had been multiple pixel shaders each one would have used the same lighting calculation code. Utilising only one pixel shader further allowed the lighting to be consistent for all objects. If multiple pixel shaders were used then there would be a greater risk of error within lighting calculations due to missed doe.

The array that holds all the light information in the *App* class is passed into the shaders that are being used for each object in the scene. In each Shader class there is a struct 'LightBufferType'. In each shader the light data is packed into an instance of this struct, which will be passed into the *uber_ps*. These structs are created to be 16 byte aligned which allows the data in them to be correctly and efficiently interpreted by the pixel shader.

```
struct LightBufferType
{
    XMINT4 active_Spot_Spec_Atten[3];
    XMFLOAT4 ambient[3];
    XMFLOAT4 diffuse[3];
    XMFLOAT4 position[3];
    XMFLOAT4 attenVal_specPow[3];
    XMFLOAT4 dirValues_Cone[3];
    XMINT4 material;
};
```

Struct the light variables are packed into

```
for (int i = 0; i < 3; i++)
{
    lightPtr->active_Spot_Spec_Atten[i] = XMINT4(light[i]->getIsActive(), light[i]->getUseSpotlight(), light[i]->getUseSpecular(), light[i]->getUseAttenuation());
    lightPtr->ambient[i] = light[i]->getAmbientColour();
    lightPtr->diffuse[i] = light[i]->getDiffuseColour();
    lightPtr->position[i] = XMFLOAT4(light[i]->getPosition().x, light[i]->getPosition().y, light[i]->getPosition().z, 0.f);
    lightPtr->attenVal_specPow[i] = XMFLOAT4(light[i]->getConstantFactor(), light[i]->getLinearFactor(), light[i]->getQuadraticFactor(), light[i]->getSpecularPower());
    lightPtr->dirValues_Cone[i] = XMFLOAT4(light[i]->getDirection().x, light[i]->getDirection().y, light[i]->getDirection().z, light[i]->getConeSize());
}
```

Packing of the variables in the pixel shader

When the *uber_ps* receives the light information from the shader, it will then calculate the outputted colour for that pixel. This is based on the values that are passed into the constant light buffer and the sample of the texture at that pixels UV position. The first calculation that is done in the pixel shader is the making of the light vector's. This is achieved by normalising the world position of the pixel subtracted from the light's position. Three light vectors are created in total, one for each light.

```
for (int j = 0; j < number_lights; j++)  
{  
    lightVector[j] = normalize(position[j].xyz - input.worldPosition.xyz);  
}
```

The light value of the pixel is the combination of all 3 lights on that pixel. To achieve this the pixel shader loops through every light and it then checks if the light is toggled to be active. If it is on, then the copy of the current diffuse and ambient colour is saved.

The first check in this loop is to determine if the light is using attenuation. If it is then the attenuation is calculated using the constant, linear and quadratic values that are passed into the constant Light Buffer. This attenuation value is multiplied by the copy of the current lights diffuse colour.

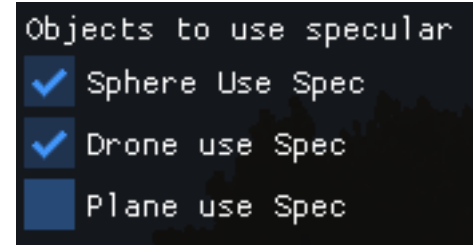
A check then takes place to work out the type of light being used in this iteration which in turn determines which lighting calculation should be made. Both point and spot light calculations make use of the *calculateLighting* function. This function works by first calculating the light intensity. This is done by getting dot product of the normal and light vector and the saturating the result to clamp the value between 0 and 1. This value is then multiplied by the copy of the diffuse colour and saturated returning the final diffuse light colour.

If the light is a point light then this value is added to the copied ambient colour of the light. The only difference is that if the light is a spotlight then the outputted value from the *calculateLighting* function is multiplied by a spot value. This value is used to create a cone that the light will shine through. First, the angle between the light's direction and the increase of the light vector is calculated through use of the dot product. The max function is then used to ensure this value is not less than 0 as this would result in the light shining in-front and behind the spotlight. Finally, the pow function is used with the cone value in

the light buffer being used as the exponent, to determine the cone of light. This formula was adapted from Braynzarsoft's tutorial on spotlights [1].

```
spot = pow(max(dot(-lightVector[i], normalize(dirValues_cone[i].xyz)), 0.0f), dirValues_cone[i].w);
```

The final check that is done for this light is whether or not the light uses specular lighting. If it does, then the object that is being rendered is checked if it exhibits a specular component. This was implemented as in the real world not all materials have a specular component. Therefore, in this scene the user has the ability to toggle specular lighting for each object. By default, the drone and sphere have specular lighting enabled.



If both the light and object allow the use of specular then the specular value is calculated, via the Blinn-Phong method. To calculate this a view vector is created in the vertex shader and passed in to the pixel shader. This is accomplished by subtracting the camera position from the pixel's world position and normalising it. The following calculation is then done to calculate the specular component. This value will be added to the final light colour.

```
// blinn-phong specular calculation
float3 halfway = normalize(lightDirection + viewVector);
float specularIntensity = pow(max(dot(normal, halfway), 0.0), specularPower);
return saturate(specularColour* specularIntensity);
```

Blinn-Phong Specular calculation

At the end of each iteration the final colour is increased by the value that has been calculated. After the three iteration are complete the final colour is saturated and then multiplied by the texture colour that was sampled, this value then has the final specular colour added to complete the calculation.

Shadows

Both point and spot lights are able to cast shadows of all objects in the scene. Within the *App class*, the first pass that occurs in the render function is used to calculate depth maps. These are further used in shadow calculations. To create the depth maps the view

and projection matrixes have to be generated. The *generateViewMatrix* function in the *Light* class of the DXFramework further had to be altered to allow point light calculations. If the light is a spotlight then the previous code would have worked, as it only needs to generate one view matrix for the direction the light is facing.

```
// default up vector
XMVECTOR up = XMVectorSet(0.0f, 1.0f, 0.0f, 1.0f);

direction = normalize(direction);

if (direction.y == 1)
{
    up = XMVectorSet(0.0f, 0.0f, 1.0f, 1.0);
}
else if (direction.y == -1)
{
    up = XMVectorSet(0.0f, 0.0f, -1.0f, 1.0);
}

XMVECTOR dir = XMVectorSet(direction.x, direction.y, direction.z, 1.0f);
XMVECTOR right = XMVector3Cross(dir, up);
up = XMVector3Cross(right, dir);
// Create the view matrix from the three vectors.
viewMatrix[index] = XMMatrixLookAtLH(position, position + dir, up);
```

Spotlight View Generation Matrix

However, with a point light six view matrixes would have to be generated. One for each direction: up, down, forward, back, left and right. To accomplish this, when generating the view matrix you pass in the type of light and the index of the map for each direction e.g spotlight index will always be 0 and point can go between 0 and 5. When the light is a point light this index is used to define a direction. This view matrix is saved in an array of view matrixes, at the position of the index passed in.

```
XMVECTOR direct;

switch (index)
{
    case 0: direct = XMVECTOR(1.f, 0.f, 0.f); break;
    case 1: direct = XMVECTOR(-1.f, 0.f, 0.f); break;
    case 2: direct = XMVECTOR(0.f, 1.f, 0.f); break;
    case 3: direct = XMVECTOR(0.f, -1.f, 0.f); break;
    case 4: direct = XMVECTOR(0.f, 0.f, 1.f); break;
    case 5: direct = XMVECTOR(0.f, 0.f, -1.f); break;
}

//direct = normalize(direct);

// default up vector
XMVECTOR up = XMVectorSet(0.0f, 1.0f, 0.0f, 1.0f);

if (direct.y == 1)
{
    up = XMVectorSet(0.0f, 0.0f, 1.0f, 1.0);
}
else if (direct.y == -1)
{
    up = XMVectorSet(0.0f, 0.0f, -1.0f, 1.0);
}

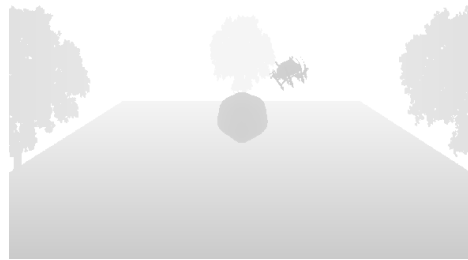
XMVECTOR dir = XMVectorSet(direct.x, direct.y, direct.z, 1.f);

XMVECTOR right = XMVector3Cross(dir, up);
up = XMVector3Cross(right, dir);
// Create the view matrix from the three vectors.
viewMatrix[index] = XMMatrixLookAtLH(position, position + dir, up);
```

Point light View Generation Matrix

Every object in the scene, with the same translations, rotations and scaling are used in the creation of the depth maps. The *Depth Map Shader* takes in the world matrix, light view matrix and projection matrix. Both the sphere and trees use different depth map shaders in the depth map calculations. This occurs due to the facts that for the sphere, tessellation and vertex manipulation all have to be applied before any depth calculation.

The tree mesh also had to be put through the geometry shader before any depth calculations are taken. Both these shaders make use of the same hlsl files that are used in the rendering of the geometry. However, the pixel shader that is used is changed from the *uber_ps* to the *depth_ps*.



An example of a depth map that can be generated using the current scene

With three lights in the application there is the potential to generate thirteen depth maps. As the third light has been permanently set as a spotlight, it may only generate one map so the max is two point lights (six maps each) and one spot light (one map) making a maximum of thirteen. All maps created are saved to an array of *ShadowMaps*. This array is passed into all shaders along with a counter of how many maps have been used. Each shader has a struct that is passed into the *uber_ps*, called 'ShadowBufferType'. This passes through values such as: all the light's view and projection matrixes: a counter of the number of depth maps that have been used and which light was used for each map and finally the shadow bias value. All maps are then passed into *uber_ps* as 2D textures and a second sampler is passed in that is responsible for sampling the depth maps.

In the *uber_ps* we calculate if any light is shining on this pixel before any of the lighting calculations are complete. First, we loop thirteen times, with every pass checking if the shadow map is being used. If it isn't then the process bypasses straight to the next iteration. Otherwise, we then calculate the *lightViewPosition* based on the world position of the pixel and the light view and projection matrixes. The projected texture co-ordinates are then calculated. If these co-ordinates are inside the limits of 0 and 1 on the x and y, then we work out if the pixel is lit.

Calculating this involves the depth map of iteration 'i' being sampled using the shadow-

Sampler and the projected tex coords as the UV co-ordinates. The light depth value is then calculated and has the shadow map bias subtracted from it. If the light depth value is less than the sampled value from the depth map then that pixel is lit by the light that was used for this depth map. To save this value an array of 3 integers are saved, if the pixel is lit then it is saved which light is responsible for lighting it.

The final step for the implementation of shadows includes the loop of the three lights that are used to calculate the combined light of the three light colours. A check is done to ensure that the pixel is definitely being lit by that specific light.

Vertex Manipulation

The scene demonstrates one instance of vertex manipulation, through the morphing of the sphere mesh. A combination of waves are passed through the sphere mesh with an amplitude, frequency and speed which the user can define in the ImGui window. As the sphere is also a tessellated object, the *Tessellation Shader* was created to combine both vertex manipulation with tessellation.

Three values that are responsible for vertex manipulation are passed into the *Tessellation Shader* and are packed into the struct 'VertexBufferType' which is 16 byte aligned. This struct is then passed into the *SphereTessellation_ds*.

As the position of the vertex was being morphed and changed then the normals had to be re-calculated every time there was a change. To accomplish this task the parametric equation of the sphere was used. For this the spherical polar co-ordinates needed to be calculated. Phi and theta were determined by using the normal of the point on the sphere. To change the vertex position itself the inputted normal was then manipulated by multiplying itself by the wave that was being passed over the sphere.

Then position on the sphere was then pushed out along this manipulated normal.

```
temp_normal = temp_normal * ((sin((temp_normal.x + speed)*frequency)*amplitude) + (cos((temp_normal.y + speed)*frequency)*amplitude) + (sin((temp_normal.z + speed)*frequency)*amplitude));
```

Wave Formula

After changing the position of the vertex, the new normal had to be calculated. In order to do this, two vectors were created: the tangent and the bi-tangent. Both were created using the *positionCalc* function, which takes in two angles and use them in the parametric equation of the sphere equation to calculate the new point. This point is then manipulated, using the same wave formula as the initial point was morphed by.

Tangent:

When creating the tangent, the same theta value as the initial point was used. However, the phi value was offset by 0.01. The value that was returned by the function was then subtracted by the world position of the point to create the vector.

Bi-tangent:

When creating the bi-tangent the same phi value as the initial point was used. However, the phi value was offset by 0.01. The value that was returned by the function was then subtracted by the world position of the point to create the vector.

```
tangent = positionCalc((phi - offset), theta) - vertex_position;  
bi_tangent = positionCalc(phi, (theta - offset)) - vertex_position;
```

Tangent and Bi-Tangent creation

By getting the cross product of these two vectors the new normal is calculated.

Post Processing

The application exhibits the ability to demonstrate two post processing effects: Glow and Edge Detection. Depending on whether or not post processing effects are enabled will change how the application outputs the scene to the window. If there is no effect then the meshes' in the scene will be rendered straight to the window. Otherwise, the scene gets render to a texture. This texture will then be used in functions in the post processing class, which was created for dealing with all post processing techniques. The outputted texture from this class will then be rendered to the window.

In the Post Processing class there is multiple functions that are used in the creation of post processing techniques. The only two public functions *ApplyGlowPostProcessing* and

ApplyEdgeDetectionPostProcessing take in a texture of the scene, use it in their own calculations and finally output a new render texture to be outputted to the window. This class holds all Shader classes that are responsible for the effects such as the Horizontal Blur, Vertical Blur, GlowMap, Glow and Edge Detection Shader.

Edge Detection:

The application makes use of Sobel Edge Detection [2] as a post processing effect. The *EdgeDetection* function uses the render texture of the scene and passes it into the *EdgeDetection_ps*. The current pixel and its surrounding eight neighbouring pixels are sampled. The vertical and horizontal Sobel convolution kernels are then created, based off of the values from the sampled points on the texture.

| | | |
|----|---|----|
| -1 | 0 | +1 |
| -2 | 0 | +2 |
| -1 | 0 | +1 |

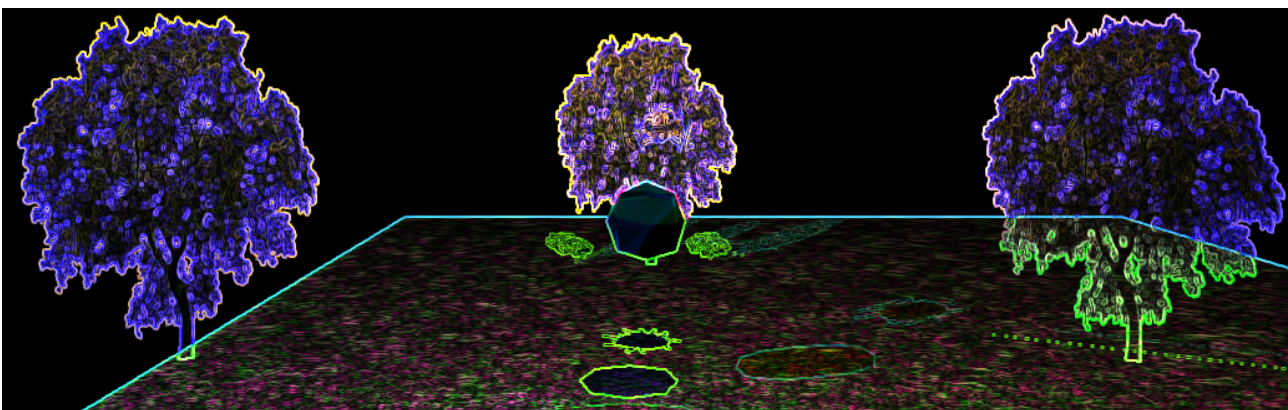
| | | |
|----|----|----|
| +1 | +2 | +1 |
| 0 | 0 | 0 |
| -1 | -2 | -1 |

Sobel Convolution Kernels [3]

```
sobel_verticle = (kernal_pos[0] + (2.f*kernal_pos[1]) + kernal_pos[2]) + (-kernal_pos[6] - (2.f*kernal_pos[7]) - kernal_pos[8]);
sobel_horizontal = (kernal_pos[2] + (2.f*kernal_pos[5]) + kernal_pos[8]) + (-kernal_pos[0] - (2.f*kernal_pos[3]) - kernal_pos[6]);
```

Kernels Creation

The output colour is then calculated by square rooting both the square of these values together. This texture is then outputted to the screen.



Example of Edge Detection

Glow:

The application produces a glowing effect through post processing techniques. In the `ApplyGlow` function multiple passes are made to create the glow. The creation of the glow map was adapted from LearnOpenGL Example [4].

The first pass is the creation of the glow map. This uses the inputted texture of the scene that is being rendered. This texture is passed into the *GlowCalc_ps*, in addition with a struct of the colour of the glow, which can be changed by the user. In this pixel shader the calculation works out the brightness of the value by transforming the pixel to grayscale. This value of this is then checked to see if it exceeds a threshold limit. If it doesn't then the outputted colour is set to black. If it does then the outputted colour is set to the value of the colour that is in the constant buffer.



Example of a GlowMap -
before downscaled and blurred

The texture that is outputted from this function is then downscaled using an ortho mesh that is half the size of the screen's width and height. This downscaled texture is then put through the Horizontal and Vertical Blur Shader which applies a Gaussian Blur to the texture. This texture is then upscaled for its use in the final pass. In the final pass the glow map that has been created is combined with the current scene texture. This uses the Glow Shader where both textures are passed into the *Glow_ps*. Both of these textures are then sampled and the values of the samples are combined and clamped between 0 and 1.

This outputted texture is then returned to the *App* class where it is either outputted to the screen or is used in the edge detection calculation.

Tessellation

The application uses the tessellation stage of the DirectX11 pipeline only once in this application, this can be observed in the sphere mesh. The tessellation that has been implemented is dynamic, as it takes into consideration the distance from the vertex to the camera's position. First, a new sphere mesh, *TessellationSphereMesh*, was created with a Primitive Topology with a 3 control point patchlist. This mesh is then used in the *Tessellation Shader*, which makes use of the *SphereTessellation_hs* and *SphereTessellation_ds*.

The Hull Shader uses a patch type of 'tri' and a partitioning type of 'fractional_odd', to avoid detail popping. In the PatchConstantFunction, three edge tessellation factors are calculated and one inside. These calculations use the *LODCalc* function, which takes in a float3, that defines the midpoint of two vertices. In this function the distance between the camera and the vertex is computed. This distance is then used to calculate the lerp value that is used to calculate the amount of tessellation, this is based on the minimum and maximum distance that the user can change in the ImGui window. This function then returns the tessellation factor of the patch by linear interpolating between the minimum and maximum tessellation value set by the user, this has a limit between 1 and 64. The inside value is calculated by getting using the average of all three vertex world positions as the midpoint in the function.

```
output.edges[1] = LODCalc((inputPatch[0].worldPosition + inputPatch[2].worldPosition) / 2.f);
output.edges[2] = LODCalc((inputPatch[0].worldPosition + inputPatch[1].worldPosition) / 2.f);
output.edges[0] = LODCalc((inputPatch[1].worldPosition + inputPatch[2].worldPosition) / 2.f);

//Set the tessellation factor for tessallating inside the triangle.
output.inside = LODCalc((inputPatch[0].worldPosition + inputPatch[1].worldPosition + inputPatch[2].worldPosition) / 3);
```

Edge and Inside function calls

Geometry Shader

The three billboarded trees in the scene are created in the geometry shader. To achieve this a new mesh was created 'TreePointMesh'. 'TreePointMesh' defines three points that would be passed into the geometry shader. The procedure for creating the trees and billboarding them was adapted from the Braynzarsoft tutorial on Billboarded Trees [5].

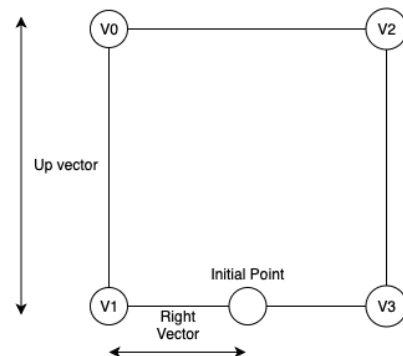
In the TreeShader class, the only buffer that is passed into the shader files contains the matrixes for calculating the world position and the view vector of the quads. However, in the *Tree_gs* a constant buffer was created that holds the UV co-ordinates of the four corners of the quad that will be created.

The first step in creating the billboard is defining the height and width that will be used in the quads creation. This is followed by creating a vector between the cameras position and the inputed position, which is multiplied by the world position. The Y component of this vector is set to 0 before it is normalised, this is done to ensure that any rotation is only done in the XZ axis.

The second step involves using this vector in the vector that will be added or subtracted to the current position of the point. To do this we calculate the cross product between an up vector and with the normal vector that was created. This value is then normalised, so to allow the multiplication of the vector with the width of the quad. A new up vector is created with a y component of the height of the quad.

The vertices are the created by adding or subtracting them from the initial points position.

```
verts[0] = input[0].position - right + up; //top left
verts[1] = input[0].position - right; //bottom left
verts[2] = input[0].position + right + up; // top right
verts[3] = input[0].position + right; //bottom right
```



The four position are then looped through and the quad is created around this point.

Critical Evaluation

When I began this project, I aspired to create an application that demonstrated five key graphical techniques and displays my knowledge and understanding of the DirectX11 Graphics Pipeline. Upon evaluating the final outcomes and analysing my own achievements, I feel have achieved what I set out to accomplish. However, that is not to exclude the different methods and techniques that could have been implemented to improve this application.

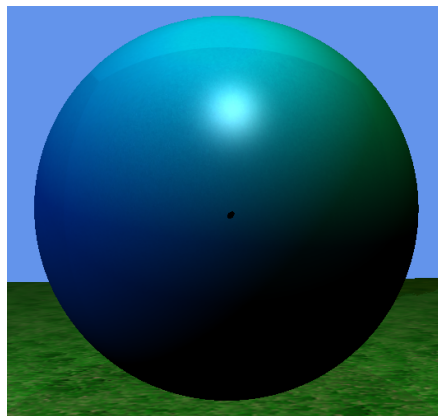
First, upon reflection the structure of the code that has been implemented could have been improved. In the current project there is a large amount of code repetition in the Shader classes, from the defining of structs that are consistent in the majority of shader to the create of the buffers. This could be improved upon by having a class that holds all structs that would be used in multiple shaders and have static functions that could have been used for the creation of the buffers and samplers. Furthermore, late into the development of the application, I came to the realisation the way each shader class was being passed variables may not have been the most sophisticated. Instead of passing every variable that was being used, the values could have been saved in a struct and passed in.

Second, the implementation of Soft Shadows could have added to the amount of post processing techniques that were displayed in the scene. In the application's current state, the shadows that are generated have hard edges that are slightly pixilated. Thus, taking away from the scenic quality of the application. In order to remedy this, I would have used the blurring technique - which is also implemented in the glow effect - on all the active depth map textures before they were passed into the *uber_ps* to be analysed and used.

The final improvement of the application would have been an improved implementation of Tessellation. At the moment the tessellation is based on the sphere's vertex's before it has been manipulated. However, I feel like this could be changed so that the edge and inside values that are calculated take into consideration how the vertex has been morphed. I feel a possible way of doing this would be to pass the amplitude, frequency and wave speed into the *SphereTessellation_ds*. Before calculating the midpoint values the patch would be pushed out along the normal which would have been manipulated by the wave formula. This technique could give a more interesting form of tessellation.

Despite some initial apprehension, overall I found this project not only enjoyable but thought-provoking and a true test of my development skills. Unfortunately, some challenges that I encountered provided to be more complex than initially perceived. I managed to overcome multiple of these challenges. However, not all were able to be fixed to be include in the final build of the application.

My first major problem was with normal calculations for the sphere. The technique with the tangent and bi-tangent had been implemented. However, there was a problem with the pixel in the center of the sphere, this problem also occurred at the opposite side of the sphere.



I overcame this challenge by changing how the calculation is done at these points. A check was implemented to see if phi was at 0 or 180 on the sphere, if it was then the following values were passed into the calculation function.

```
tangent = positionCalc(phi - offset, rad0) - vertex_position;  
bi_tangent = positionCalc(phi - offset, rad90) - vertex_position;
```

The second problem was not as easily fixed. In my proposal I had intended for the center sphere to have been floating above a cylinder which had a displacement map wrapped around it. The cylinder mesh was created and a displacement map had been applied. However, the normals that were being calculated were not correct. I used the same method for the calculations as I had in the sphere manipulation, the difference being that the parametric equation of a cylinder was used. Multiple methods were attempted to get a correct looking lighting however, none returned a correct looking result.

The displaced cylinder would have been textured using slope-based texturing, which was

adapted from the RasterTek Tutorial on Procedural Terrain Texturing [6]. The aim of this technique was to blend the texture depending on the steepness of mesh at every point on the displacement map. Three textures would've been blended together that would have given a nice weather look to the column. This technique was implemented and is in the *calcTexture* function found in the *uber_ps*.

References

- [1] Braynzarsoft.net. (2019). 21. *Spotlights - Braynzar Soft*. [online] Available at: <https://www.braynzarsoft.net/viewtutorial/q16390-21-spotlights> [Accessed 17 Oct. 2019].
- [2] - Gist. (2019). *GLSL Fragment Shader: Sobel Edge Detection*. [online] Available at: <https://gist.github.com/Hebali/6ebfc66106459aacee6a9fac029d0115> [Accessed 14 Nov. 2019].
- [3] - Homepages.inf.ed.ac.uk. (2019). *Feature Detectors - Sobel Edge Detector*. [online] Available at: <https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm> [Accessed 14 Nov. 2019].
- [4] - Learnopengl.com. (2019). *LearnOpenGL - Bloom*. [online] Available at: <https://learnopengl.com/Advanced-Lighting/Bloom> [Accessed 10 Nov. 2019].
- [5] - Braynzarsoft.net. (2019). 36. *Billboarding (Geometry Shader) - Braynzar Soft*. [online] Available at: <https://www.braynzarsoft.net/viewtutorial/q16390-36-billboarding-geometry-shader> [Accessed 5 Dec. 2019].
- [6] - Rastertek.com. (2019). *Tutorial 13: Procedural Terrain Texturing*. [online] Available at: <http://www.rastertek.com/dx11ter13.html> [Accessed 26 Oct. 2019].
- [7] - Deviantart.com. (2019). *seamless cartoon grass texture by mbrockwell on DeviantArt*. [online] Available at: <https://www.deviantart.com/mbrockwell/art/seamless-cartoon-grass-texture-201505192> [Accessed 7 Nov. 2019].
- [8] - Freepngs.com. (2019). [online] Available at: <https://www.freepngs.com/tree-pngs> [Accessed 5 Dec. 2019].