

CMP301 – Graphics Programming with Shaders Coursework Report

Sam Gallacher, 1700284

Scene Overview – Brief Response

The core location of the scene is based around a lake, with the surrounding hills/mountains and the lake water itself present. It takes heavy inspiration from Bower Lake – a location present within the game Fable 2 (2008). In a small overview, my scene demonstrates the following features as required by the brief:

- **Vertex Processing**
 - Manipulation of main terrain vertices via a displacement map
 - Normals calculated via a Sobel filter
 - Texturing achieved via a texture map, with blending
 - Water effect created with DUDV texture with reflection, refraction and Fresnel Effect implemented
- **Post Processing**
 - Bloom with editable glow values
 - Relative camera movement-based Motion Blur with editable blur factors
- **Lighting**
 - Directional lighting to represent the Sun
 - Spot light with attenuation
 - Point light behind Hero Hill
 - Core lighting values (position, diffuse) editable via ImGui
 - Shadowing present on Island tomb, using PCF as an anti-aliasing/smoothing technique
 - Self-shadowing present on main terrain using PCF as an anti-aliasing/smoothing technique
- **Additional Stages**
 - Dynamic tessellation implemented based upon camera distance with correct texturing and lighting
 - Tessellation factor bounds are editable via ImGui
 - Geometry shader has been used to generate billboarded grass and trees
 - Generated trees and grass are textured and lit correctly

Scene Overview – UI Elements/Controls

As mentioned above, several elements of my scene are editable via ImGui. These are explained below:

- **Light Options**
 - **Sun Light Options**
 - Sun Light Ambient – Ambient colour emitted by the directional Sun light
 - Sun Light Diffuse – Diffuse colour of the directional Sun light
 - Sun Light Direction – Direction of the directional Sun light
 - **Hill Point Options**
 - Hill Light Diffuse – Diffuse of the hill point light
 - Hill Light Position – Position of the hill point light
 - **Spot Options**
 - Spot Light Diffuse – Diffuse of the spot light
 - Spot Light Position – Position of the spot light
 - Spot Light Direction – Direction of the spot light
 - **Attenuation Options**
 - Constant Factor – The constant factor value in the attenuation calculation
 - Linear Factor – The linear factor value in the attenuation calculation
 - Quadratic Factor – The quadratic factor value in the attenuation calculation
- **Tessellation Options**
 - Minimum Factor – Minimum factor of tessellation
 - Maximum Factor – Maximum factor of tessellation

- **Post-Processing Options**
 - **Bloom Options**
 - Bloom Enabled – Used to enable/disable bloom
 - R Glow Intensity – The intensity applied to the R channel
 - G Glow Intensity – The intensity applied to the G channel
 - B Glow Intensity – The intensity applied to the B channel
 - **Motion Blur Options**
 - Motion Blur Enabled – Used to enable/disable motion blur
 - X Factor – The factor used for blurring in the X direction
 - Y Factor – The factor used for blurring in the Y direction
- Sobel Scale Factor – Used to control smoothness/roughness of lighting
- Terrain Normals Mode – Returns the normals as a colour from the pixel shader so they can be observed

Techniques – Scene Render Flow Overview

Before analysing in detail each technique demonstrated, it is important to establish the flow of render operations within the scene. The code follows the below flow:

- 1) Depth Pass – Calculates depth of all geometry for shadowing purposes
- 2) Reflection Pass – Used to determine what geometry should reflect above the water
- 3) Refraction Pass – Used to determine what geometry should refract under the water
- 4) Main Scene Pass – Used to render the scene to a render to texture
- 5) Bloom Pass – Used to apply bloom to the scene render to texture
- 6) Motion Blur Pass – Used to apply motion blur to the scene render to texture
- 7) Final Pass – Used to output the scene render to texture to the final output ortho mesh

Techniques – Vertex Processing,

Displacement Map

As previously mentioned, the core terrain is generated via a displacement map application onto an originally flat plane. The core idea behind a displacement map is to use the information stored within its pixels to algorithmically displace the current vertex vertically via an arbitrary scale.

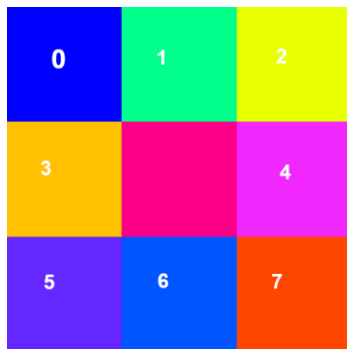
Specifically within the created scene, the displacement occurs at the domain shader stage, as the terrain has been tessellated which will be discussed later (if the terrain was not tessellated, the displacement should occur at the vertex shader stage). The red channel of the displacement map is sampled at the specified texture coordinate and its value halved, then multiplied by a scale factor of 150. This result is then used as the incrementing value of the vertex's y position.

```
float SampleDisplacementMap(float2 texCoords)
{
    float4 textureColour = texture0.SampleLevel(sampler0, texCoords, 0);
    return (textureColour - 0.5f) * 150.0f;
}
```

This creates a unique challenge however – how can the normals be calculated for every vertex? Unless the result of the displacement also resulted in the original shape (a plane, in which case the normals would all be (0, 1, 0)), a form of calculation must be applied to approximate a correct value and acceptable value of the normal for each vertex, where acceptability in this case is defined by visually appealing lighting.

Techniques – Vertex Processing, Sobel Filter

To solve the issue and calculate the normals for the displaced vertices, a Sobel filter was used. A sobel filter allows us to calculate an approximation of the differences between the horizontal and vertical. Hence, by sampling the neighbouring pixels in the displacement map to create a 3x3 matrix, the corresponding X and Y kernels can be applied to calculate the filters. Then, the Z filter can be calculated to define the smoothness/roughness of bumps. Finally, these can be combined into a float3 and normalized to produce an approximate normal for the vertex based upon the differences in displacement from the neighbouring pixels. A visual image depicting the ordering of the pixel image (where the number defines the array index), the sobel kernels used and well as the calculation code itself is visible below:



$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \quad \mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}$$

```
float3 SobelCalculateNormals(float2 texCoords)
{
    //Get the width and height of the displacement texture, get normalized UV coords, sample the neighbours and perform Sobel filtering
    float textureWidth;
    float textureHeight;
    displacementTexture.GetDimensions(textureWidth, textureHeight);
    float2 texelSize = float2(1.0f / textureWidth, 1.0f / textureHeight);

    //Neighbouring pixel offsets
    float2 offsets[8];
    offsets[0] = texCoords + float2(-texelSize.x, -texelSize.y);
    offsets[1] = texCoords + float2(0.0f, -texelSize.y);
    offsets[2] = texCoords + float2(texelSize.x, -texelSize.y);
    offsets[3] = texCoords + float2(-texelSize.x, 0.0f);
    offsets[4] = texCoords + float2(texelSize.x, 0.0f);
    offsets[5] = texCoords + float2(-texelSize.x, texelSize.y);
    offsets[6] = texCoords + float2(0.0f, texelSize.y);
    offsets[7] = texCoords + float2(texelSize.x, texelSize.y);

    //Neighbouring pixel height samples
    float2 samples[8];
    for (int i = 0; i < 8; i++)
    {
        samples[i] = SampleDisplacementMap(offsets[i]);
    }

    //Calculate the Sobel filters using the kernels, apply a constant factor of 2 to X and Y, and a user set factor to Z to control smoothness/roughness
    float filterX = samples[0] - samples[2] + (2.0f * samples[3]) - (2.0f * samples[4]) + samples[5] - samples[7];
    float filterY = samples[0] + (2.0f * samples[1]) + samples[2] - samples[5] - (2.0f * samples[6]) - samples[7];
    float filterZ = sobelScaleFactor * sqrt(max(0.0f, 1.0f - filterX * filterX - filterY * filterY));
    return normalize(float3(2.0f * filterX, filterZ, -2.0f * filterY));
}
```

It is critical to note that a constant factor should be applied to these calculations, particularly in the case of the Z filter – failure to do so can result in the corresponding filter value having little effect once normalized and hence an incorrect normal can be produced. In the scene, the factor is definable by the user so they can witness the effect of the smoothness/roughness of lighting which is defined by the Z filter.

Techniques – Vertex Processing, Texture Mapping

Now that the terrain has been displaced to create the hills and mountains desired, the pixel shader has to apply texturing to the terrain to depict grass, sand, rock and dirt.

To achieve this, a texture map has been used. 4 square textures of each terrain type are included. The texture map is responsible for defining where each of these textures should be positioned in the scene and how they should blend with one another. This is particularly important around the lake edge and edges of the dirt roads, so that a visually pleasing blend between sand, grass, dirt and rock can be observed, versus a sharp transition between the types.

Each channel of each pixel represents a different terrain type as below:

- Red – Grass
- Green – Dirt
- Blue – Sand
- Alpha - Rock

The value of each channel represents the transparency value for the corresponding terrain type. The texture map is sampled at each texture coordinate and each transparency value for each terrain type is obtained. Then, each terrain texture is sampled at the same texture coordinate, the transparency value is applied to the sampled colour and all are combined to obtain the final pixel colour. The below image of code may better demonstrate this concept:

```
float4 mapColour = textureMap.Sample(sampler0, input.tex);
float grassTransparency = mapColour.r;
float dirtTransparency = mapColour.g;
float sandTransparency = mapColour.b;
float rockTransparency = mapColour.a;

float4 textureColour = float4((grassTexture.Sample(sampler0, input.tex) * grassTransparency) +
    (dirtTexture.Sample(sampler0, input.tex) * dirtTransparency) +
    (sandTexture.Sample(sampler0, input.tex) * sandTransparency) +
    (rockTexture.Sample(sampler0, input.tex) * rockTransparency));
```

This method provides a very easy and simple method to not only map textures efficiently, but to easily blend them as well and produce a very nice end result.

Below is an evidence image of the displacement with correct normals, lighting and with the texture mapping.



Techniques – Vertex Processing, Water Effect

As it is in the name of the location and is one of the main focal points within the scene, it felt imperative that a pleasing water effect be produced for the lake. Originally, Gerstner waves were planned as per the original proposal submission. However, as described within the milestone document, these have been omitted due to complexity of implementation as well as being potentially unsuitable/an overly extreme water effect for a lake from an artistic point of view.

Instead, reflection, refraction and the Fresnel effect have been implemented, with a DUDV texture also being used.

As described in the scene render flow overview, 2 passes related to this concept are made – the reflection and refraction pass.

The reflection pass is responsible for rendering any geometry above the water and the refraction pass is responsible for rendering any geometry below the water. In order to accommodate this, a clip plane shader is used to cull the rendered geometry on a set side of the clip plane. By setting the clip plane to be above the y axis for reflection and to be below the y axis for refraction, the clip plane vertex shader can compute the dot product of the vertex position in world space and the clip plane itself using the HLSL clip function to cull the correct vertices, as below:

```
output.clip = dot(mul(input.position, worldMatrix), clipPlane);
```

It is important to note that the correct view matrix has to be determined for the reflection pass. To reflect correctly, the view matrix has to be calculated as if the camera had been translated and rotated under the water. This is done simply by decreasing the camera's y by twice the vertical distance between it and water plane and then inverting the pitch.

The Fresnel effect is an effect which describes the reflection of light based on angles of incidence and is an explanation of polarization. In practice, the effect can make a form of optical media appear more/less reflective based upon the angle at which it is viewed. Within the scene, this is an important effect for the water to produce – at shallow angles, the water should appear transparent and reflective with geometry underneath its surface barely visible. But, as the angle becomes sharper to an almost top-down view, the water should become opaque and underwater geometry become easily visible.

The water vertex shader hence needs to relate this effect to that of the camera's position and angle. To do this, it calculates the forward vector from the camera by subtracting the camera's position from the world space position of the vertex. The water pixel shader can then use this with a combination of a constant reflective power and refractive factor to define the strength and "fade" of the Fresnel Effect:

```
float4 reflectivePower = 10.0f;
float3 cameraVec = normalize(input.cameraForwardVector);
float refractiveFactor = dot(cameraVec, float3(0.0f, 1.0f, 0.0f));
refractiveFactor = pow(refractiveFactor, reflectivePower);

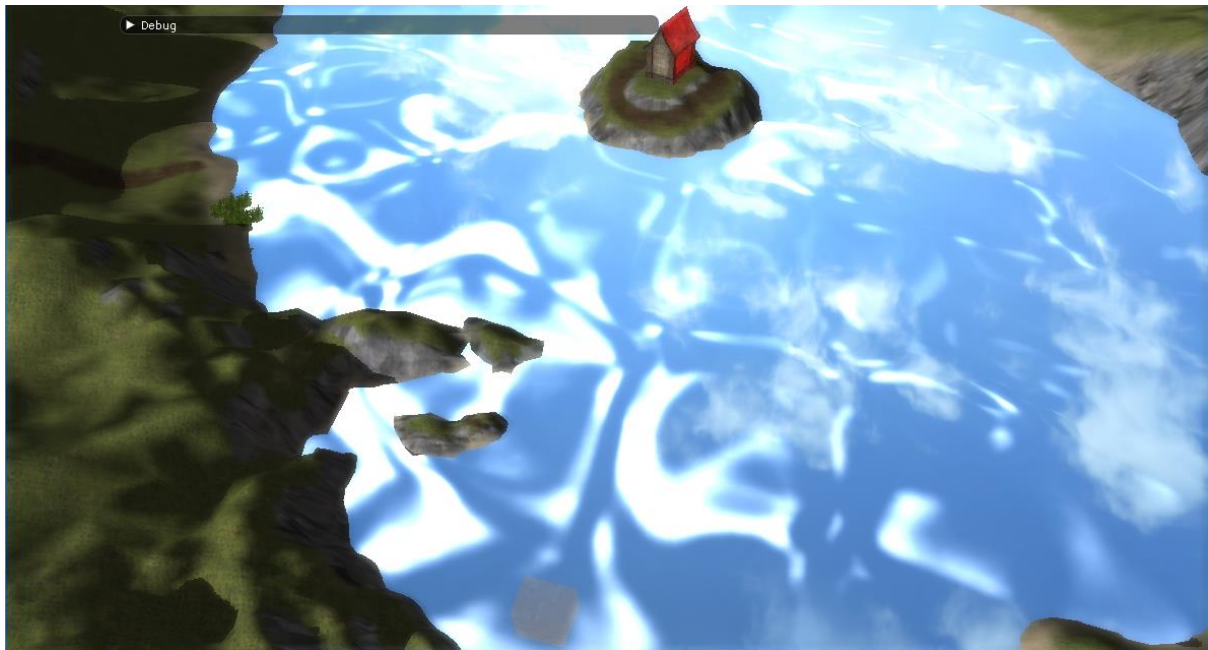
float4 retColour = lerp(reflectionColour, refractionColour, refractiveFactor);
retColour = lerp(retColour, float4(0.0f, 0.3f, 0.5f, 1.0f), 0.2f);
```

Then, the pixel shader can then sample the DUDV texture to obtain a distortion value. By appending a move factor which increases over time, movement of the water can be simulated. Finally, the reflect function can be used in combination with a shine damper, specular and the previously calculated Fresnel effect to produce the final pixel values with reflection and the Fresnel effect:

```
float shineDamper = 20.0f;
float3 reflection = -reflect(normalize(input.lightVector), normal);
float specular = dot(normalize(reflection), normalize(input.lightVector));
if (specular > 0.0f)
{
    specular = pow(specular, shineDamper);
    retColour = saturate(retColour + specular);
}

return retColour;
```

Below is an evidence screenshot of the water effect, with the skybox reflection, the water ripple, the normal map specular effect and the Fresnel effect with a wooden cube visible (toward the bottom middle of the image).



Techniques – Post Processing, Introduction

As previously mentioned, I've implemented both bloom and motion blur for the Post Processing section.

These two will be analysed in detail just below. It is important however to note that both use a simple bypass vertex shader – i.e. the application of each part of each effect is done purely in the pixel shader as the final geometry of the scene will already have been rendered to a texture.

The vertex shader used during the application of all post processing effects is as below:

```
OutputType main(InputType input)
{
    OutputType output;

    // Calculate the position of the vertex against the world, view, and projection matrices.
    output.position = mul(input.position, worldMatrix);
    output.position = mul(output.position, viewMatrix);
    output.position = mul(output.position, projectionMatrix);

    // Store the texture coordinates for the pixel shader.
    output.tex = input.tex;
    output.normal = input.normal;

    return output;
}
```

It is also notable as mentioned in the UI/Controls section that each effect can be independently disabled if desired.

Techniques – Post Processing, Bloom

For my first post processing effect, I've implemented bloom, with editable glow values to increase the effect.

In order to display the effect, the scene is first rendered to a texture with all the geometry in the scene pass.

Once this is done, the bloom pass handles the application of the effect to the scene render texture. The bloom pass performs the following operations:

- Adds glow independently to each colour channel using a glow shader, based on the values set in the UI and outputs the result to a glow texture
- Adds a Gaussian blur to the glow texture using a Gaussian blur shader, by averaging both a horizontal and vertical Gaussian blur and outputs the result to a glow blur texture
- Blends both the glow and glow blur textures together using a blend shader and then outputs the result back to the scene texture

First, the scene texture is passed to a glow shader along with the glow values set in the UI.

Conceptually and in terms of its application, the glow shader is very simple but effective. It samples the scene texture and independently multiplies each channel by the intensities set for each channel via the UI, ensuring that the min function is used to cap the value to the maximum of 255. This new colour is then outputted from the pixel shader to the glow texture.

```
float4 main(InputType input) : SV_TARGET
{
    float4 textureColour = sceneTexture.Sample(Sampler0, input.tex);

    textureColour.r = min(textureColour.r * intensity.r, 255.0f);
    textureColour.g = min(textureColour.g * intensity.g, 255.0f);
    textureColour.b = min(textureColour.b * intensity.b, 255.0f);

    return textureColour;
}
```

Secondly, the Gaussian blur is applied. The glow texture used previously is passed in, as are the dimensions of the screen to allow the texel size to be determined – this is used to sample the neighbouring pixels for the blur. Then, the axis at which the blur should be applied is passed in as a float2, where the value set to 1 in the float2 defines the axis on which to apply the blur. Using a kernel size of 11, the weights are defined as an array of floats. These weights have been determined using the referenced gaussian kernel calculator, with a sigma of 2. Then, looping through by twice the kernel size (starting negatively to go from the left pixels to the right pixels), the returned blur colour is incremented by sampling the glow texture at the specified texture coordinates. This results in a light blur on the left and right edge pixels with an increasing strength of blur toward the centre pixel we're currently acting upon.

```
float4 Gaussian(float texelSize, float2 axis, InputType input)
{
    //Kernel Size 11, Sigma 2
    int kernelSize = 11;

    float weights[] =
    {
        0.0093f,
        0.028002f,
        0.065984f,
        0.121703f,
        0.175713f,
        0.198596f,
        0.175713f,
        0.121703f,
        0.065984f,
        0.028002f,
        0.0093f
    };

    float4 colour = float4(0.0f, 0.0f, 0.0f, 0.0f); //Initialize the colour to be totally black with no alpha

    for (int i = -(kernelSize - 1); i < (kernelSize - 1); i++) //Loop through from left to right
    {
        //Apply the gaussian blur in the direction set, based on the weighting for this pixel
        colour += texture0.Sample(sampler0, input.tex + float2(texelSize * float(i) * axis.x, texelSize * float(i) * axis.y)) * weights[abs(i)];
    }

    return colour;
}
```


The above action is performed in the horizontal and vertical directions, summated and then divided by 2 to determine an average and then returned. This returned blur value is outputted to the blur texture.

```
float4 main(InputType input) : SV_TARGET
{
    float4 horizontalGaussian = Gaussian(1.0f / screenDimensions.x, float2(1.0f, 0.0f), input); //Calculate the texel size and perform a horizontal gaussian blur
    float4 verticalGaussian = Gaussian(1.0f / screenDimensions.y, float2(0.0f, 1.0f), input); //Calculate the texel size and perform a vertical gaussian blur

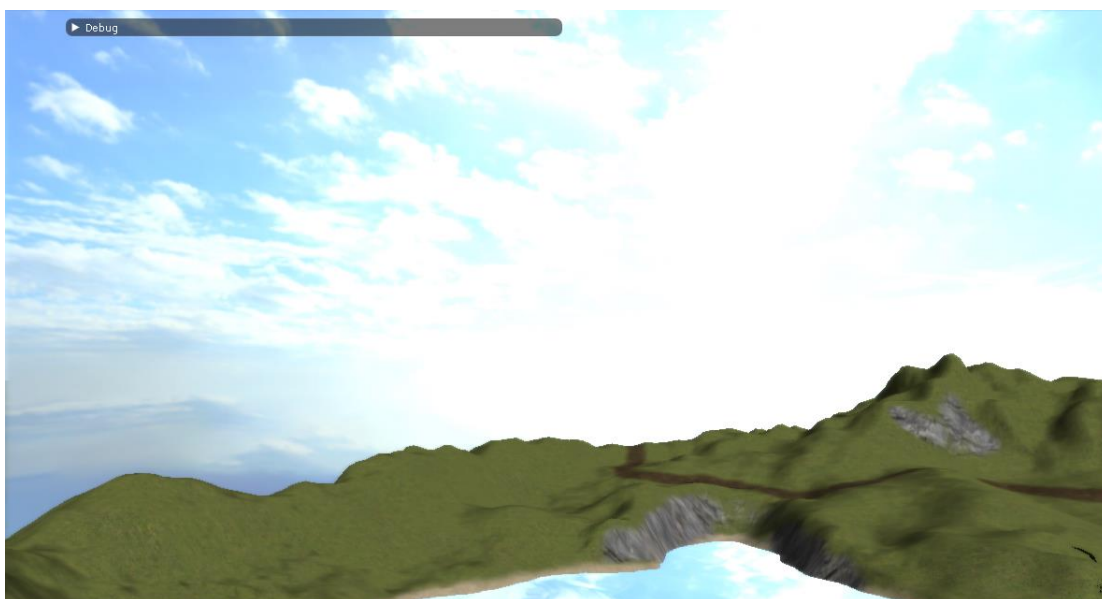
    float4 retColour = (horizontalGaussian + verticalGaussian) / 2.0f; //Average out the returned blurs
    retColour.a = 1.0f; //The colour should be fully opaque
    return retColour;
}
```

The final stage is to combine both the scene and glow blur textures into one to produce the final bloom effect that should be visible to the user.

The blend shader performs this task. Again, it performs simply but effectively. A constant blend factor of 0.8 is set. The shader samples both the glow blur and scene textures at the same texture coordinates and stores the colour of the pixels. Then, it simply multiplies the first colour by the blend factor and sums this with the second colour multiplied by 1 minus the blend factor. The 1 minus allows for a bias to be applied i.e. to bias more glow blur. It then returns the blended colour, and this is then outputted to the scene texture, ready for rendering to the screen or indeed for motion blur to then be applied.

```
float4 main(InputType input) : SV_TARGET
{
    float blendFactor = 0.8f;
    float4 firstColour = firstTexture.Sample(sampler0, input.tex);
    float4 secondColour = secondTexture.Sample(sampler0, input.tex);
    return (firstColour * blendFactor) + (secondColour * (1.0f - blendFactor));
}
```

Overall, the multiple shader stages and render textures used with bloom do make it a fairly complex effect to compute and implement. That said, the effect produced is very pleasing indeed. Below is an evidence image of bloom in the scene.



Techniques – Post Processing, Motion Blur

I've also implemented a motion blur effect based upon camera movement, with editable scale values to increase the effect.

In order to display the effect, the scene is first rendered to a texture with all the geometry in the scene pass as described previously. Then, the motion blur pass is responsible for outputting the motion blurred version of the scene to another render texture and rendering that to the scene's orthographic mesh so that it is visible to the user.

To achieve the effect based on camera movement, details about the camera from the previous frame need to be known in order to define movement – in this case, the camera's previous position. At the end of final pass, the camera's current position is stored as the previous. Hence, when the motion blur pass is called next frame, the camera's new position can be compared to the old one to define the motion vector.

To determine blurring in each direction, the forward, up and right vectors need to be known. This is to ensure the movement and blurring is relative to the camera's axis and NOT the world axis.

Once these are calculated, the amount to blur in the X axis is determined by the motion down the right vector. The amount to blur in the Y axis is determined by the motion down the forward vector.

The code for calculating each vector and setting the blur values is as below:

```
//Calculate the right, up and forward vectors
XMATRIX camRotationMatrix = XMMatrixRotationRollPitchYaw(camera->getRotation().x * 0.0174532f, camera->getRotation().y * 0.0174532f, 0);
XMVECTOR camTarget = XMVector3TransformCoord(XMVectorSet(0.0f, 0.0f, 1.0f, 0.0f), camRotationMatrix);
camTarget = XMVector3Normalize(camTarget);

XMATRIX RotateYTempMatrix;
RotateYTempMatrix = XMMatrixRotationY(camera->getRotation().y * 0.0174532f);

XMVECTOR camRight = XMVector3TransformCoord(XMVectorSet(1.0f, 0.0f, 0.0f, 0.0f), RotateYTempMatrix);
XMVECTOR camUp = XMVector3TransformCoord(XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f), RotateYTempMatrix);
XMVECTOR camForward = XMVector3TransformCoord(XMVectorSet(0.0f, 0.0f, 1.0f, 0.0f), RotateYTempMatrix);

//The motion vector is the difference in positions
XMVECTOR motionVector = XMVectorSet(camera->getPosition().x - previousCameraPosition.x, camera->getPosition().y - previousCameraPosition.y, camera->getPosition().z - previousCameraPosition.z, 0.0f);

XMVECTOR xMotion = XMVector3Dot(motionVector, camRight); //X motion is the dot product of our motion vs the right vector
XMVECTOR yMotion = XMVector3Dot(motionVector, camForward); //Y motion is the dot product of our motion vs the forward vector
float xDirection = (*xMotion.m128_f32) * motionBlurXFactor; //Calculate X direction (every component of the vector will have the dot product value)
float yDirection = (*yMotion.m128_f32) * motionBlurYFactor; //Calculate Y direction (every component of the vector will have the dot product value)
```

```

float4 main(InputType input) : SV_TARGET
{
    //Preset weights of the blur
    float weight0 = 0.4f;
    float weight1 = 0.2f;
    float weight2 = 0.15f;
    float weight3 = 0.1f;
    float weight4 = 0.05f;
    float weight5 = 0.045f;
    float weight6 = 0.0425f;
    float weight7 = 0.0420f;
    float weight8 = 0.0410f;
    float weight9 = 0.0405f;

    float4 colour = float4(0.0f, 0.0f, 0.0f, 0.0f); //Initialize the colour to be totally black with no alpha

    float2 texelSize = 1.0f / screenDimensions; //Calculate texel size

    //Apply a blur based on the direction and weights, increasing the position
    colour += texture0.Sample(sampler0, input.tex) * weight0;
    colour += texture0.Sample(sampler0, input.tex + texelSize * direction * 1.0f) * weight1;
    colour += texture0.Sample(sampler0, input.tex + texelSize * direction * 2.0f) * weight2;
    colour += texture0.Sample(sampler0, input.tex + texelSize * direction * 3.0f) * weight3;
    colour += texture0.Sample(sampler0, input.tex + texelSize * direction * 4.0f) * weight4;
    colour += texture0.Sample(sampler0, input.tex + texelSize * direction * 5.0f) * weight5;
    colour += texture0.Sample(sampler0, input.tex + texelSize * direction * 6.0f) * weight6;
    colour += texture0.Sample(sampler0, input.tex + texelSize * direction * 7.0f) * weight7;
    colour += texture0.Sample(sampler0, input.tex + texelSize * direction * 8.0f) * weight8;
    colour += texture0.Sample(sampler0, input.tex + texelSize * direction * 9.0f) * weight9;

    colour.a = 1.0f; //The colour should be fully opaque

    return colour;
}

```

Overall, a simple, yet effective effect at depicting speed/motion of the camera. Below is an evidence image, taken with a high X factor while moving toward the right.



Techniques – Lighting, Introduction

Lighting and shadowing were one of the most complex components to create. The combination of multiple types of light, the different properties of each and indeed how they interact together was tricky to implement correctly but was achieved successfully.

I've implemented 3 types of light in my scene, each with their own properties. These are:

- Directional Light – Used to represent the Sun
 - Ambient is editable
 - Diffuse is editable
 - Direction is editable
- Point Light – Present behind the hill, position can be increased in Y to affect more of the scene
 - Diffuse is editable
 - Position is editable
- Spot Light – Present just outside the scene area, can be brought in to light the corresponding area
 - Diffuse is editable
 - Position is editable
 - Direction is editable

It is key to note the effect of each of these lights is calculated in isolation, and appended to the final output colour at the end of each pixel shader.

It is also key to note that depth values are used to determine whether or not a pixel should be lit – the method for determining these depth values will be explained later in the Shadows section of the report.

Techniques – Lighting, C++ Buffers

On the C++ side, the lighting buffer is quite large. This is to accommodate the effects of all the lights on each other.

```
//The buffer for all the light info
struct LightBufferType
{
    XMFLOAT4 sunAmbient;
    XMFLOAT4 sunDiffuse;
    XMFLOAT3 sunDirection;
    float padding;
    XMFLOAT4 spotDiffuse;
    XMFLOAT3 spotPosition;
    float constantFactor;
    float linearFactor;
    float quadraticFactor;
    XMFLOAT2 paddingTwo;
    XMFLOAT3 spotDirection;
    float paddingThree;
    XMFLOAT4 hillDiffuse;
    XMFLOAT3 hillPosition;
    float paddingFour;
};
```

This buffer is used in 3 places – the terrain tessellation, billboard and shadow pixel shaders. The lighting code itself is identical in each, but separate pixel shaders are used as extra specific operations have to be performed. This includes blending/texture mapping in the terrain tessellation shader and sampling the billboarded sprite in the billboard pixel shader.

Techniques – Lighting, Directional Light

```
if (!(sunTexCoord.x < 0.f || sunTexCoord.x > 1.f || sunTexCoord.y < 0.f || sunTexCoord.y > 1.f))
{
    // Sample the shadow map (get depth of geometry)
    depthValue = sunShadowMapTexture.Sample(shadowSampler, sunTexCoord).r;
    // Calculate the depth from the light.
    lightDepthValue = input.sunViewPos.z / input.sunViewPos.w;
    lightDepthValue -= shadowMapBias;

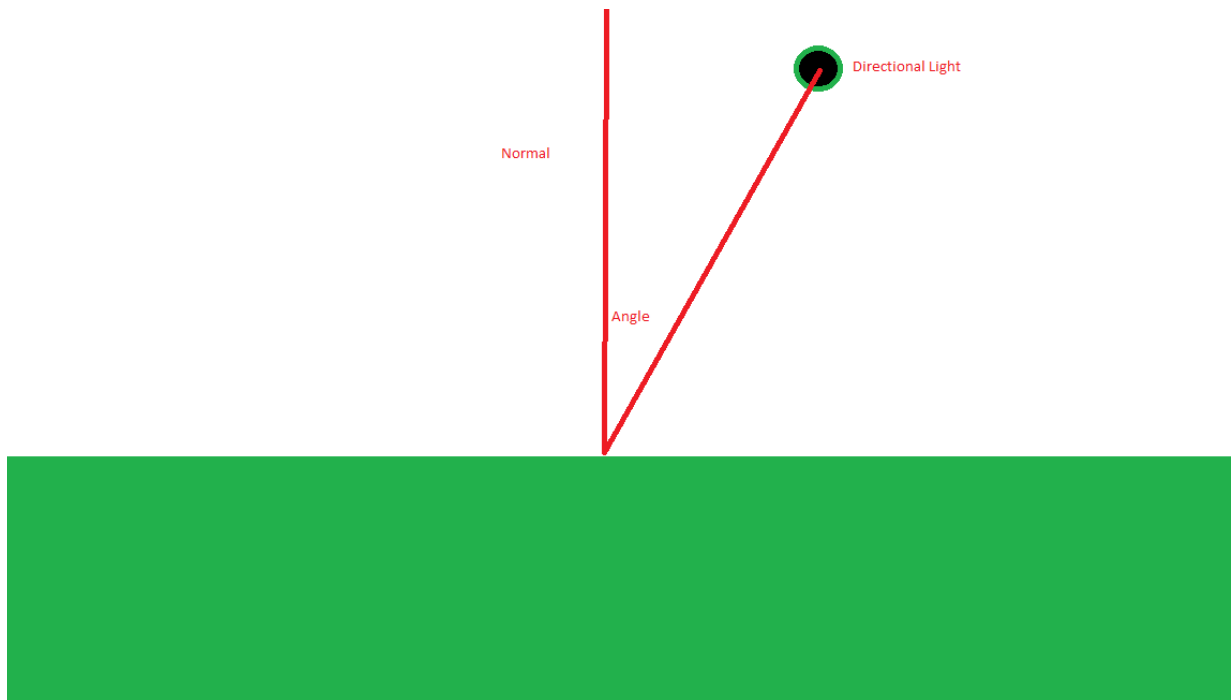
    // Compare the depth of the shadow map value and the depth of the light to determine whether to shadow or to light this pixel.
    if (lightDepthValue < depthValue)
    {
        colour += sunAmbient + calculateLighting(-sunDirection, input.normal, sunDiffuse);
        IsLit = true;
    }
}

float4 calculateLighting(float3 lightDirection, float3 normal, float4 diffuse)
{
    float intensity = saturate(dot(normal, lightDirection));
    float4 colour = saturate(diffuse * intensity);
    return colour;
}
```

For the directional light, the sun shadow map is sampled, and the depth value converted into homogeneous coordinates with the bias subtracted. Then, if the depth value of the light is less than the depth value in the shadow map, the area should have lighting applied.

To apply the lighting, ambient from the Sun is always added on. The `calculateLighting` function is called, passing in the negative direction of the Sun, the normal and the Sun's diffuse colour.

By calculating the dot product, we can determine the angle between the normal and the direction of the light. Based on the value of this, we can determine how intensely the directional light should affect the pixel, where a greater angle should reduce the effect. Saturating this ensures our value remains between 0 and 1. Below is an image visualizing this concept:



Finally, the diffuse is applied to colour the light by multiplying it by the intensity, saturating it to clamp the value and then returning our directional light colour.

This returned colour is appended to the final colour for this pixel.

Techniques – Lighting, Spot Light

```
if (!(spotTexCoord.x < 0.f || spotTexCoord.x > 1.f || spotTexCoord.y < 0.f || spotTexCoord.y > 1.f))
{
    // Sample the shadow map (get depth of geometry)
    depthValue = spotShadowMapTexture.Sample(shadowSampler, spotTexCoord).r;
    // // Calculate the depth from the light.
    lightDepthValue = input.spotViewPos.z / input.spotViewPos.w;
    lightDepthValue -= shadowMapBias;

    // // Compare the depth of the shadow map value and the depth of the light to determine whether to shadow or to light this pixel.
    if (lightDepthValue < depthValue)
    {
        float3 spotVector = normalize(spotPosition - input.worldPosition);
        colour += calculateSpotLighting(spotVector, spotDirection, spotPosition, input.worldPosition, input.normal, spotDiffuse);
        IsLit = true;
    }
}

float4 calculateSpotLighting(float3 spotVect, float3 spotDir, float3 spotPos, float worldPos, float3 normal, float4 diffuse)
{
    float4 colour = float4(0.0f, 0.0f, 0.0f, 1.0f);
    float cosA = dot(spotDir, -spotVect);

    float intensity = saturate(dot(normal, spotVect));
    colour = saturate(diffuse * intensity);
    colour *= getAttenuation(spotPos, worldPos);
    colour *= pow(max(cosA, 0), 8);

    return colour;
}
```

Calculating lighting for the spot light was more difficult than the directional light. This is since a spot light only acts within a certain range and has attenuation implemented which complicates the calculation vs a directional light.

As with the directional light, the same depth logic is applied to determine whether the area should be lit – the only difference being the spot shadow map is used.

In order to determine the range in which the spot light should apply, a vector from the spot light's position to its world position needs to be known.

In the vertex or hull shaders (as appropriate), the vertex's position is determined by lerping between the top edge by the UV's V coordinate, then lerping between the bottom edge by the UV's V coordinate and finally lerping both of those by the UV's U coordinate. Then, to determine the world position, the vertex position is simply multiplied by the world matrix and then outputted to the pixel shaders.

```
float3 v1 = lerp(patch[0].position, patch[1].position, uvCoord.y);  
float3 v2 = lerp(patch[3].position, patch[2].position, uvCoord.y);  
vertexPosition = lerp(v1, v2, uvCoord.x);  
  
output.worldPosition = mul(float4(vertexPosition, 1.0f), worldMatrix).xyz;
```

Now that the world position is known, the vector is calculated and passed into `calculateSpotLighting` along with the spot light's direction, position, world position, normal and diffuse.

First, the return colour is initialized to be completely black. The dot product is calculated between the direction of the spot light and the negative vector to determine $\cos A$ – the angle at which the spot light is to the area being lit. This should be negative to define the angle at which the spot light no longer applies. The intensity is then calculated similarly to that of the directional light, with the spot light vector defining the direction in world space. Diffuse is then applied as well.

Attenuation was used to control the fall-off of the light and each factor is controllable via ImGui. These factors are constant, linear and quadratic where each has a more profound effect respectively when applied. To calculate this, the spot position and the world position are passed in.

```
float4 calculateSpotLighting(float3 spotVect, float3 spotDir, float3 spotPos, float worldPos, float3 normal, float4 diffuse)
{
    float4 colour = float4(0.0f, 0.0f, 0.0f, 1.0f);
    float cosA = dot(spotDir, -spotVect);

    float intensity = saturate(dot(normal, spotVect));
    colour = saturate(diffuse * intensity);
    colour *= getAttenuation(spotPos, worldPos);
    colour *= pow(max(cosA, 0), 32);

    return colour;
}
```

The standard attenuation is applied, with HLSL's `length` function used to calculate the distance between the spot light's position and the world position. It is key to note that a check is performed to ensure the denominator of the function does not exceed one – otherwise, the colour would reduce when multiplied by the attenuation producing incorrect results.

The value returned is then used to multiply the spot colour value.

Finally, the max function is used to select either the negative $\cos A$ value or 0 if $\cos A$ is calculated to be positive. This is then set to the power of 32 where the power defines the roundness of the spot and then multiplied to the return spot colour. This returned colour is appended to the final colour for this pixel.

Techniques – Lighting, Point Light

```
for (int i = 0; i < 6; i++)
{
    float2 hillTexCoords = input.hillViews[i].xy / input.hillViews[i].w;
    hillTexCoords *= float2(0.5, -0.5);
    hillTexCoords += float2(0.5f, 0.5f);

    // Determine if the projected coordinates are in the 0 to 1 range. If not don't do lighting.
    if (!(hillTexCoords.x < 0.f || hillTexCoords.x > 1.f || hillTexCoords.y < 0.f || hillTexCoords.y > 1.f))
    {
        // Sample the shadow map (get depth of geometry)
        depthValue = hillShadowMapTextures[i].Sample(shadowSampler, hillTexCoords).r;

        // Calculate the depth from the light.
        lightDepthValue = input.hillViews[i].z / input.hillViews[i].w;
        lightDepthValue -= shadowMapBias;

        // Compare the depth of the shadow map value and the depth of the light to determine whether to shadow or to light this pixel.
        if (lightDepthValue < depthValue)
        {
            float3 lightVector = normalize(hillPosition.xyz - input.worldPosition);
            colour += calculateLighting(lightVector, input.normal, hillDiffuse);
            IsLit = true;
            break;
        }
    }
}
```

The last type of light implemented is a point light. This was particularly challenging due to its behaviour – it acts in all/infinite directions hence is difficult to simulate in code. To solve this problem, 6 independent shadow maps were taken for the point light for each direction:

- -X
- +X
- -Y
- +Y
- -Z
- +Z

```

for (int i = 0; i < 6; i++)
{
    float2 hillTexCoords = input.hillViews[i].xy / input.hillViews[i].w;
    hillTexCoords *= float2(0.5, -0.5);
    hillTexCoords += float2(0.5f, 0.5f);

    // Determine if the projected coordinates are in the 0 to 1 range. If not don't do lighting.
    if (!(hillTexCoords.x < 0.f || hillTexCoords.x > 1.f || hillTexCoords.y < 0.f || hillTexCoords.y > 1.f))
    {
        // Sample the shadow map (get depth of geometry)
        depthValue = hillShadowMapTextures[i].Sample(shadowSampler, hillTexCoords).r;

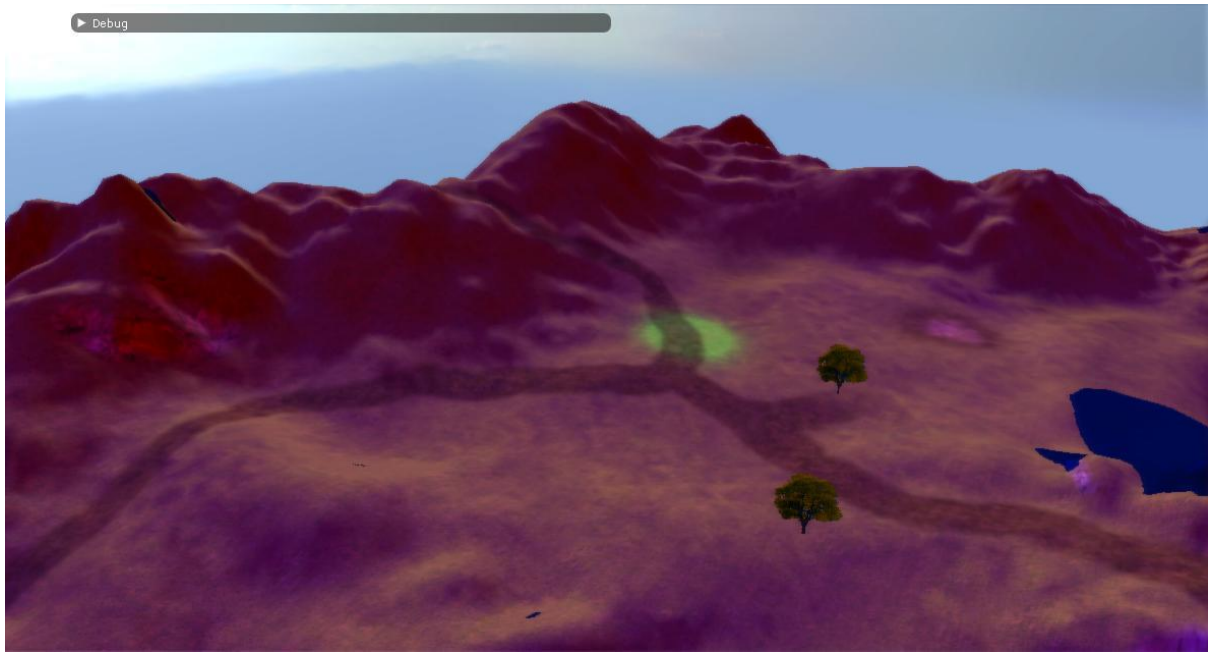
        // Calculate the depth from the light.
        lightDepthValue = input.hillViews[i].z / input.hillViews[i].w;
        lightDepthValue -= shadowMapBias;

        // Compare the depth of the shadow map value and the depth of the light to determine whether to shadow or to light this pixel.
        if (lightDepthValue < depthValue)
        {
            float3 lightVector = normalize(hillPosition.xyz - input.worldPosition);
            colour += calculateLighting(lightVector, input.normal, hillDiffuse);
            IsLit = true;
            break;
        }
    }
}

```

While the same depth logic is applied, it is important to note that a loop is performed through each shadow map for each direction of the point light. It is also important to note that a break is set when lighting is applied as the point light should only apply lighting to the area in the defined direction.

Excluding the above, the same calculateLighting function is used, with the direction defined as the normalization of the position of the hill point light and the position in world space.



Above is an evidence image of all the lighting combined. The directional ambient has been set to a low dull red, the point light has been raised high in the sky to apply over the whole scene and has had its ambient set to maximum blue – resulting in the purple visible on the terrain. The spot light is also present, set to maximum green with a small quadratic factor.

Techniques – Shadowing, Introduction

As visible in the previous screenshots of the lighting code, a Boolean variable `IsLit` is present. When any lighting is applied, its value is set to true. However, at the end of the lighting pixel shaders, a check is performed against its value. If lighting has been applied, the final light colour is saturated to keep the range between 0 and 1, then multiplied by the texture colour to display to the user.

However, if the `IsLit` is false, shadowing is applied, by calculating a shadow value, applying an ambient and then merging with the texture colour:

```
if (!IsLit)
{
    float4 shadowColour = saturate(PCFShadow(input.tex, input.sunViewPos) + float4(0.2f, 0.2f, 0.2f, 1.0f)); //We're not lit, so let's apply PCF with ambient and saturate
    return shadowColour * textureColour; //Return the saturated shadow multiplied by the texture
}

else
{
    return saturate(colour) * textureColour; //We are lit, so saturate the lit colour to between 0 and 1 and apply it to the texture colour
}
```


Techniques – Shadowing, Basic Shadows

My scene depicts a basic form of shadowing using the Tomb Island hut. To achieve any shadowing, depth must first be calculated. This is the purpose of the depth pass.

In regards to the hut, the depth pass uses the simple depth shader to determine the depth of the hut and output this to a render texture. This process is repeated for every light source.

The sun, spot and point lights each have their own shadow map. Once the view and projection matrices are determined for each, these are passed into the simple depth shader, along with the translations to the hut's position in the world.

The simple depth shader uses a very small and, as expected, simple vertex and pixel shader. The vertex shader does nothing more than outputting the projected vertex space and depth position. The pixel shader calculates and returns a depth value by dividing the z position by the homogeneous W coordinate. Hence, the shadow map will contain black to white pixels which can be sampled to determine depth – where white is close and black is far.

```
OutputType main(InputType input)
{
    OutputType output;

    // Calculate the position of the vertex against the world, view, and projection matrices.
    output.position = mul(input.position, worldMatrix);
    output.position = mul(output.position, viewMatrix);
    output.position = mul(output.position, projectionMatrix);

    // Store the position value in a second input value for depth value calculations.
    output.depthPosition = output.position;

    return output;
}
```

```
float4 main(InputType input) : SV_TARGET
{
    float depthValue;
    depthValue = input.depthPosition.z / input.depthPosition.w;
    return float4(depthValue, depthValue, depthValue, 1.0f);
}
```

Now that the depth pass has calculated depth for each light source in the scene, the scene pass uses the simple shadow shader with the hut to produce the shadow effect. When using shaders that involve light, the shaders are given pointer references to each light to reduce the size of `setShaderParameters`, as below:

```
shadowShader->SetSunLight(sunLight);
waterShader->SetSunLight(sunLight);
terrainTessellationShader->SetSunLight(sunLight);
billboardGeometryShader->SetSunLight(sunLight);

shadowShader->SetHillLight(hillPointLight);
waterShader->SetHillLight(hillPointLight);
terrainTessellationShader->SetHillLight(hillPointLight);
billboardGeometryShader->SetHillLight(hillPointLight);

shadowShader->SetSpotLight(spotLight);
waterShader->SetSpotLight(spotLight);
terrainTessellationShader->SetSpotLight(spotLight);
billboardGeometryShader->SetSpotLight(spotLight);
```

Each shadow map is passed into the simple shadow shader and the lights are already available as described above. The C++ light buffer as described earlier is filled and sent to the shadow pixel shader.

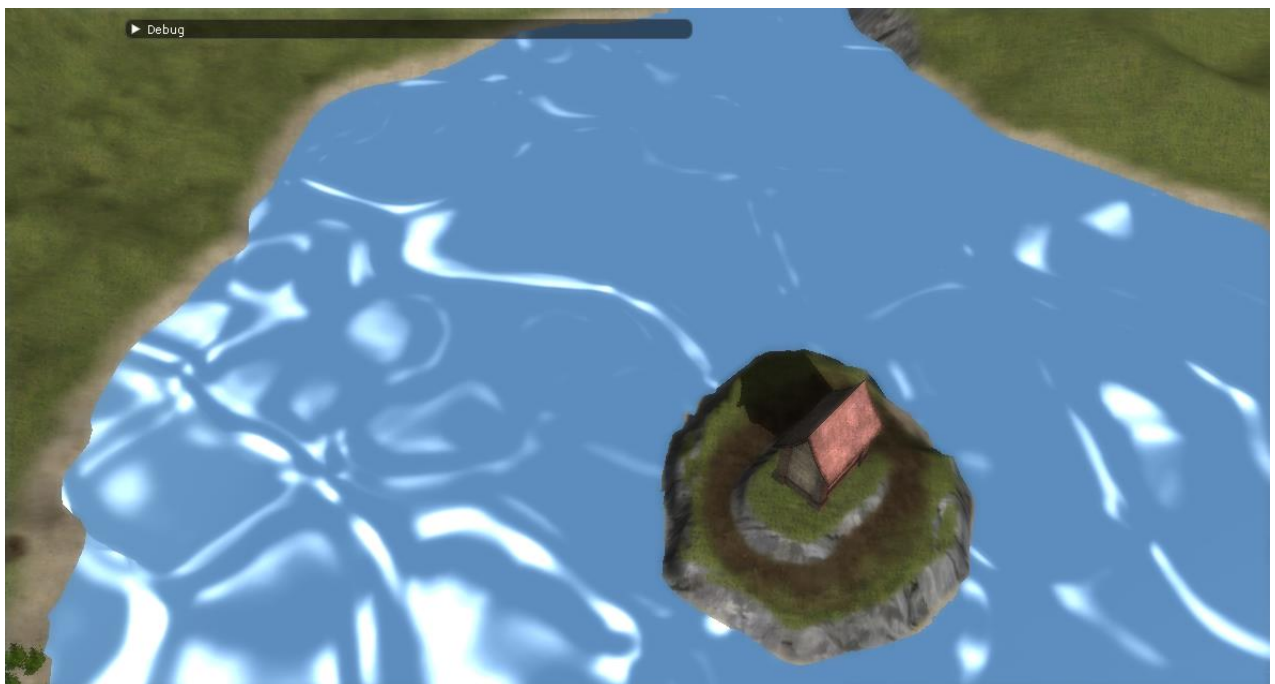
The vertex shader however, needs to be made aware of the view and projection matrices for each light. This is achieved by transposing the return values of calls to `getViewMatrix` and `getProjectionMatrix` respectively. Now that these have been created, the vertex shader can work out the view positions of each light and pass them to the pixel shader. The reason this has to be performed is to enable us to determine the depth value itself in the pixel shader and hence determine whether or not a pixel should be lit – as described and visible in the code screenshots in the Lighting section of the report.

To determine the view position of the spot light, for example, we simply multiply the input position by the world, spot view and spot projection matrices:

```
output.spotViewPos = mul(input.position, worldMatrix);
output.spotViewPos = mul(output.spotViewPos, spotViewMatrix);
output.spotViewPos = mul(output.spotViewPos, spotProjectionMatrix);
```

And hence, in the pixel shader, the view position's Z can be divided by the homogenous W coordinate to determine the depth value from the position of the light. If this is less than the value stored within the shadow map, the pixel should be lit. If not (and assuming other lights do not light the pixel as described by the IsLit Boolean), shadows are applied.

The result of this produces the following evidence of shadowing the hut:



Techniques – Shadowing, Complex Self-Shadows

While the hut shadow is effective, it was decided that shadowing should be extended to be more complex. Particularly evident with the Hero Hill geometry in the scene, it became evident that self-shadowing of the terrain onto itself would be a very good addition to the project.

This did add significant complexity however, as both the simple shadow shader and depth shader used previously would no longer work – this is because the main terrain is displaced and is dynamically tessellated so all these operations need to be applied together, concurrently and in the same shader.

To solve the issue of depth, a terrain depth shader was developed. The pixel shader is identical, but the vertex shader does not calculate the depth position. Instead, the depth shader contains a hull and domain shader.

The reason for this is simple – the terrain depth shader must perform exactly the same actions as the tessellation shader to produce exactly the same geometry to determine the correct depth. Hence, the domain shader outputs the depth position instead.

The exact logic of the domain and hull shader, and indeed how it performs the tessellation will be covered later in the Additional Stages section of the report – the key point is that it must perform exactly the same operations that the tessellation shader performs so that the depth value is correct.

Now that the correct depth has been determined, the terrain tessellation pixel shader simply performs exactly the same lighting calculations that the shadow pixel shader does – again, the only difference is that the light view positions are determined in the domain shader and not the vertex shader.

One key point is that although the lighting calculations are the same, the terrain must use its own independent pixel shader – this is so the texture mapping logic can be applied.

Below is an evidence image, showing Hero Hill which is part of the terrain casting shadows onto the terrain.



Techniques – Shadowing, PCF

Unfortunately, while the simple shadows taught within the module lectures are effective at depicting depth, there are quite heavily aliased. It was determined that this would have to be tackled to depict some visually pleasing shadows.

Hence, the shadow logic was changed to include PCF (Percentage Closer Filtering). Conceptually, it is somewhat similar to the Sobel normal calculations in that it is an average approximation. In this case, the average is that of the shadow values obtained from the shadow map. This averaging approach is used as a form of anti-aliasing to smooth out the shadow edges. While the shadows are still visibly aliased very close up, this is barely noticeable from a far to moderate distance – in the real world, it would be very difficult to spot vs the simple shadows.

```
float4 PCFShadow(float2 texCoords, float4 sunViewPos)
{
    float textureWidth;
    float textureHeight;
    sunShadowMapTexture.GetDimensions(textureWidth, textureHeight);
    float2 texelSize = float2(1.0f / textureWidth, 1.0f / textureHeight);

    //Neighbouring pixel offsets
    float2 offsets[8];
    offsets[0] = texCoords + float2(-texelSize.x, -texelSize.y);
    offsets[1] = texCoords + float2(0.0f, -texelSize.y);
    offsets[2] = texCoords + float2(texelSize.x, -texelSize.y);
    offsets[3] = texCoords + float2(-texelSize.x, 0.0f);
    offsets[4] = texCoords + float2(texelSize.x, 0.0f);
    offsets[5] = texCoords + float2(-texelSize.x, texelSize.y);
    offsets[6] = texCoords + float2(0.0f, texelSize.y);
    offsets[7] = texCoords + float2(texelSize.x, texelSize.y);

    //Neighbouring pixel shadow samples
    float shadowValue = 0.0f;
    float2 samples[8];

    //Loop through all the samples
    for (int i = 0; i < 8; i++)
    {
        //Sample the shadow map and determine the depth value
        float shadowSample = sunShadowMapTexture.Sample(shadowSampler, offsets[i]).x;
        float lightDepthValue = sunViewPos.z / sunViewPos.w;
        lightDepthValue -= 0.02f;

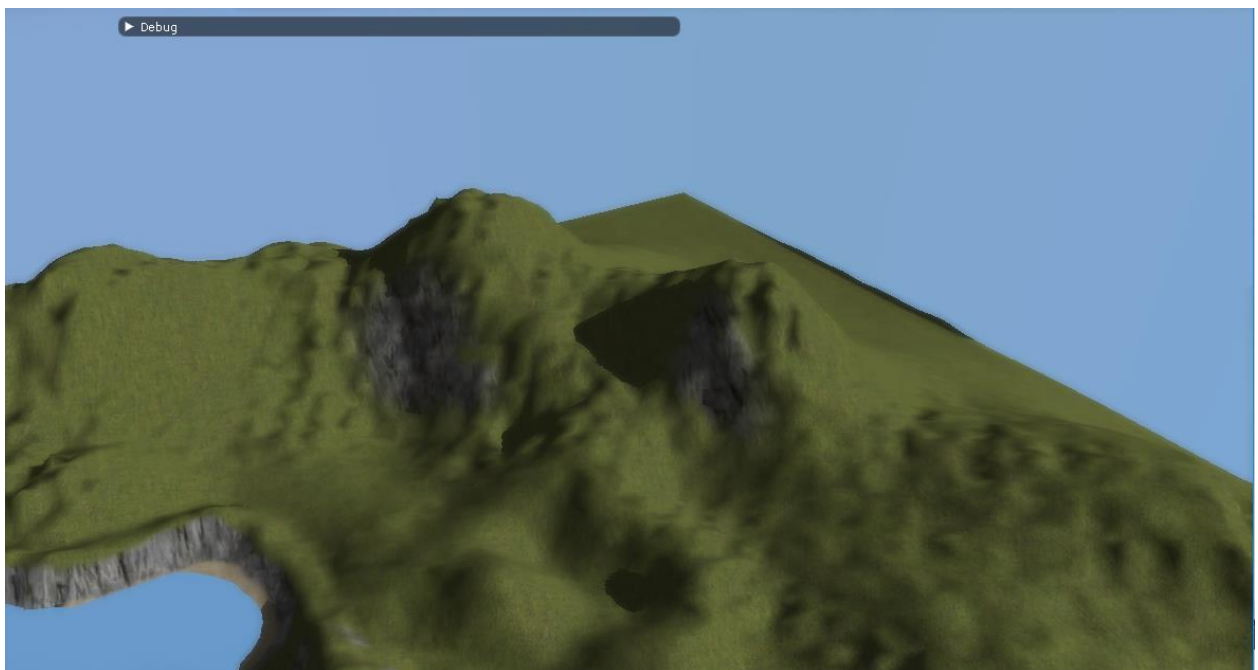
        //If the area is in shadow, increment the value
        if (shadowSample < lightDepthValue)
        {
            shadowValue += shadowSample;
        }
    }

    return shadowValue / 18.0f; //Return the value over a constant factor
}
```

Once the texel size is determined, the neighbouring pixel offsets are determined identically to the Sobel normal calculations. Then, it is determined if the position at the offset is in shadow using the identical method to the lighting. If it is, a shadow value is incremented by the sampled value.

Once all offsets have been sampled, the shadow value is returned after a division by 18 – too small of a division here can result in adversely coloured shadows.

Below is an evidence image to show the smooth shadows the slope produces onto the terrain which are far less aliased than the default simple shadows. The same effect can be viewed with the Hero Hill image used as evidence for self-casting shadows above.



Techniques – Additional Stages, Introduction

As part of the additional stages section, I've implemented dynamically controlled tessellation based on the camera's position. I've also used the geometry shader to produce billboarded sprites in the form of blades of grass and trees.

For tessellation, I've implemented a function in the UI to allow the user to set the minimum and maximum tessellation factors – these are used within the calculations for the edge and inside factors.

Techniques – Additional Stages, Dynamically Controlled Tessellation

The first additional stage, and one of the most complicated parts of the project, was dynamically controlled camera based tessellation.

The logic is simple – vertices closest to the camera should be tessellated more than those further away. This should dynamically alter as the camera moves around the world, hence producing a form of level of detail.

The first complication was that of the shaders themselves – rather than a simple vertex and pixel shader as used in most other shaders, the hull and domain shaders had to be employed instead. With this in mind, the vertex shader has been bypassed, simply outputting the inputted position, texture coordinate and normal. The hull shader has effectively taken its place.

The first step was to setup the control point function to define the patch type, partitioning, etc. My main terrain plane is 200x200, made up of quads with a resolution of 50 i.e. 50 quads long and 50 quads deep –hence the plane's primitive topology was set to 4 control point patch list, the control point function patch type was set to quads and the output control points was set to 4 as below:

```
[domain("quad")]
[partitioning("integer")]
[outputtopology("triangle_ccw")]
[outputcontrolpoints(4)]
[patchconstantfunc("PatchConstantFunction")]
OutputType main(InputPatch<InputType, 4> patch, uint pointId : SV_OutputControlPointID, uint patchId : SV_PrimitiveID)
{
    OutputType output;

    output.position = patch[pointId].position;
    output.tex = patch[pointId].tex;
    output.normal = patch[pointId].normal;

    return output;
}
```

With the control point function setup, the main patch constant function logic had to be created to define the tessellation effect.

The logic is as follows:

- Determine the midpoint positions of each edge of the quad
- Calculate the distance from the camera to each midpoint
- For every midpoint, if the distance is greater than the maximum tessellation factor then the edge factor for this edge should be set to the minimum tessellation factor
- Otherwise, the edge factor should be set to the maximum tessellation factor minus the distance
- Determine the position of the centre of the quad
- Calculate the distance between the camera and the centre of the quad
- If the distance from the camera to the centre is greater than the maximum tessellation factor then both inside factors should be set to the minimum tessellation factor
- Otherwise both inside factors should be set to the maximum tessellation factor minus the distance

So, in order to perform the above, an input buffer was setup containing the camera's position, the user-set minimum and user-set maximum tessellation factors.

The first step was to determine the midpoints. These are calculated by lerp'ing between each two neighbouring patch points by 0.5.

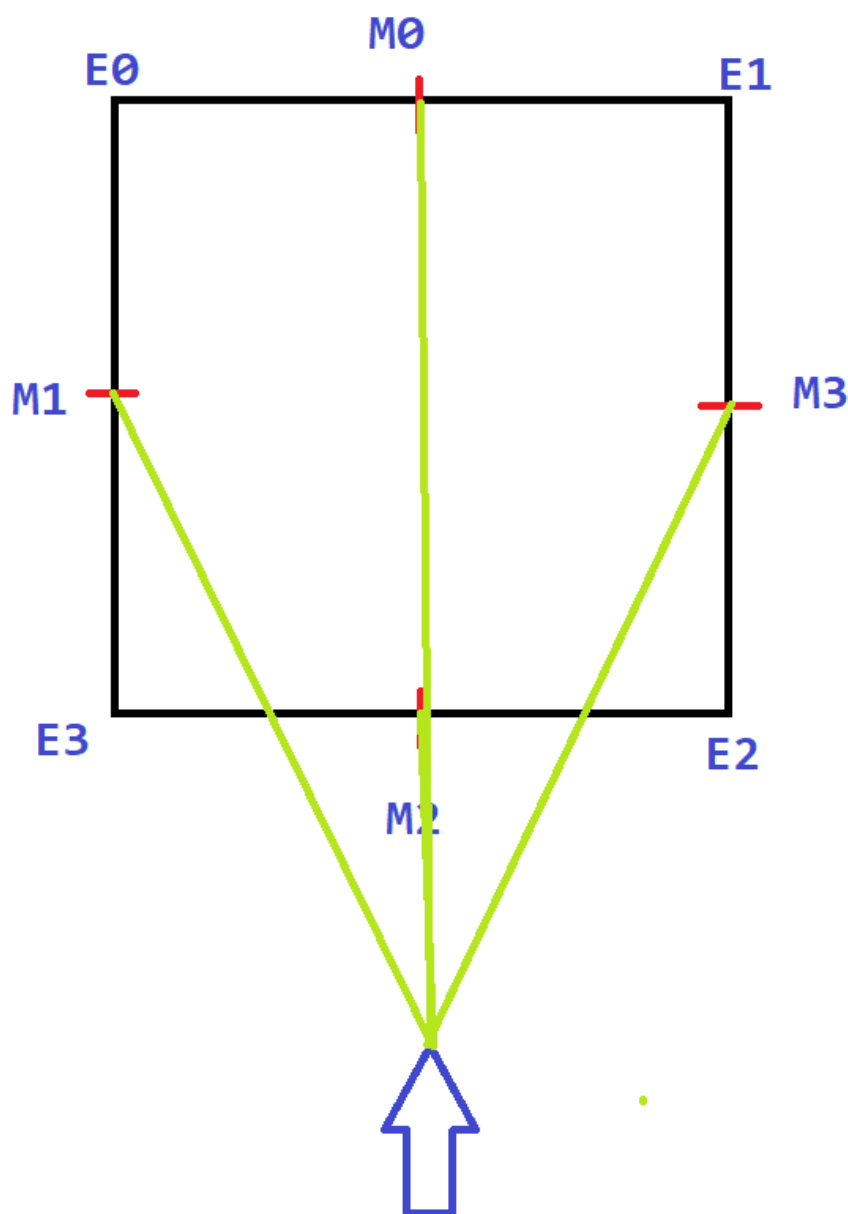
Secondly, a loop iterates through all the calculated midpoints and uses HLSL's distance function for each axis of the camera and the current midpoint, summates then together and then absolutes the value – it is critically important the value is absolutely as direction should be of null effect (i.e. being left of a quad by 50 units should affect the tessellation the same as being right of that same quad by 50 units).

Thirdly, the calculated midpoint distances are looped through, compared against the set tessellation factors as described earlier and the tessellation factors for each edge are then calculated and set.

After the edge factors have been set, the process for calculating the inside factors begins. Using the distance method in exactly the same fashion as previous, the distance from the centre of the quad to the camera is calculated. The centre of the quad is determined by summing all the patch positions and dividing by 4.

Finally, the same logic for tessellation is applied again as described in the bullet points – both inside factors are set to the same value.

A diagram of the concept has been produced below to provide visual aid to the concept. E represents a patch point, M represents a midpoint, the arrow represents the camera and the green lines represent the distances:



Determining midpoints, distances and edge factors:

```
ConstantOutputType PatchConstantFunction(InputPatch<InputType, 4> inputPatch, uint patchId : SV_PrimitiveID)
{
    ConstantOutputType output;

    float3 midpoints[4];
    float midpointDistances[4];
    float tessFactors[4];

    midpoints[0] = lerp(inputPatch[0].position, inputPatch[1].position, 0.5f);
    midpoints[1] = lerp(inputPatch[0].position, inputPatch[3].position, 0.5f);
    midpoints[2] = lerp(inputPatch[2].position, inputPatch[3].position, 0.5f);
    midpoints[3] = lerp(inputPatch[1].position, inputPatch[2].position, 0.5f);

    for (int j = 0; j < 4; j++)
    {
        midpointDistances[j] = abs(distance(cameraPosition.x, midpoints[j].x) + distance(cameraPosition.y, midpoints[j].y) + distance(cameraPosition.z, midpoints[j].z));
    }

    for (int k = 0; k < 4; k++)
    {
        if (midpointDistances[k] >= maxTessFactor || maxTessFactor - midpointDistances[k] < minTessFactor)
        {
            tessFactors[k] = minTessFactor;
        }
        else
        {
            tessFactors[k] = maxTessFactor - midpointDistances[k];
        }
    }

    output.edges[0] = tessFactors[0];
    output.edges[1] = tessFactors[1];
    output.edges[2] = tessFactors[2];
    output.edges[3] = tessFactors[3];
}
```

Determining quad centre, centre distance and inside factors:

```
float3 centrePoint = (inputPatch[0].position + inputPatch[1].position + inputPatch[2].position + inputPatch[3].position) / 4.0f;
float centrePointDistance = abs(distance(cameraPosition.x, centrePoint.x) + distance(cameraPosition.y, centrePoint.y) + distance(cameraPosition.z, centrePoint.z));
if (centrePointDistance >= maxTessFactor || maxTessFactor - centrePointDistance < minTessFactor)
{
    output.inside[0] = minTessFactor;
    output.inside[1] = minTessFactor;
}
else
{
    output.inside[0] = maxTessFactor - centrePointDistance;
    output.inside[1] = maxTessFactor - centrePointDistance;
}
```

Although tessellation has now been applied, this same plane is being displaced and lit – hence, the vertex position, texture coordinates and normals need to be determined in the domain shader to complete the task.

The normal calculations and displacement procedures themselves have already been explained in this report – the important part is how the vertex position (critically used to calculate lighting positions) and texture coordinates (critically used with the texture mapping) are determined.

The process is unexpectedly straightforward – bilinear interpolation is used to average out the patch positions and texture coordinates respectively.

To determine the vertex position, the positions of patches 0 and 1 are lerped together by the Y component of the UV coordinate passed into the main domain shader function.

The positions Patches 3 and 2 are then lerped together by the Y component of the UV coordinate.

Finally, the vertex position can be determined by lerping both of these averages by the X component of the UV coordinate.

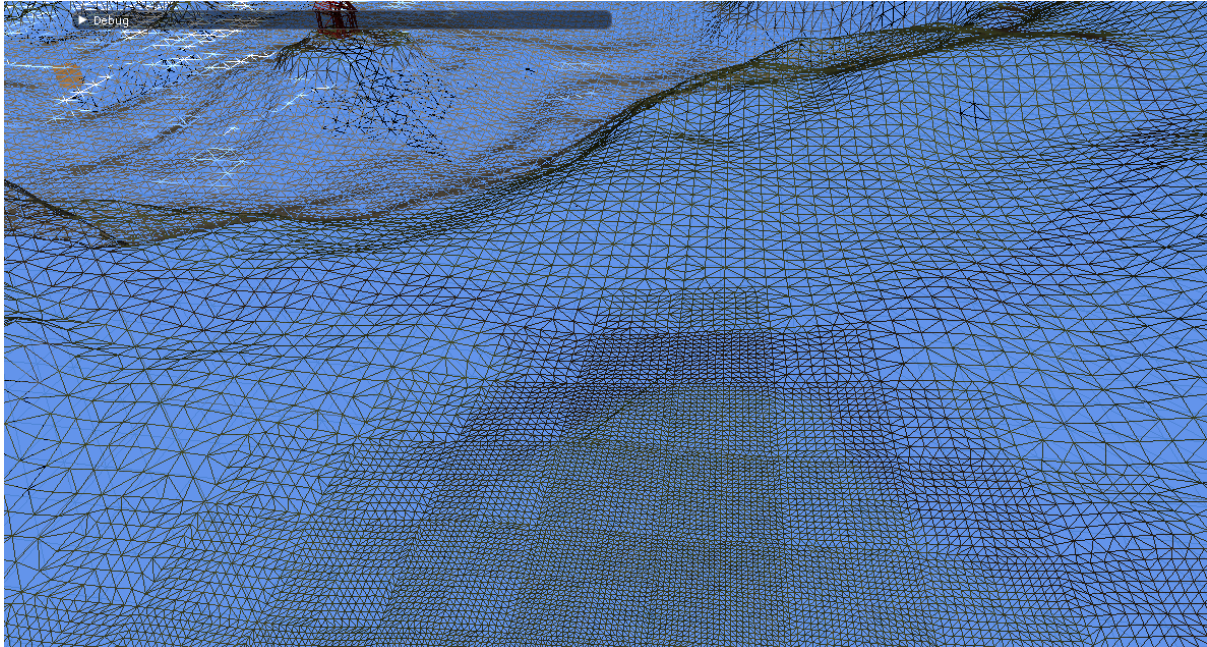
The exact same logic is also applied to calculate the texture coordinates, the difference being that the texture coordinates of the patches are used instead of the positions:

```
float3 v1 = lerp(patch[0].position, patch[1].position, uvCoord.y);
float3 v2 = lerp(patch[3].position, patch[2].position, uvCoord.y);
vertexPosition = lerp(v1, v2, uvCoord.x);

float2 t1 = lerp(patch[0].tex, patch[1].tex, uvCoord.y);
float2 t2 = lerp(patch[3].tex, patch[2].tex, uvCoord.y);
output.tex = lerp(t1, t2, uvCoord.x);
```

Once this process has been completed, the tessellation process is complete – the terrain should be tessellated, displaced and lit correctly.

Here is an evidence photo of the terrain in wireframe mode – the dynamic tessellation is clearly visible:



Techniques – Additional Stages, Geometry Shader

The final key feature of my scene and the additional stages was the use of the geometry shader to create billboarded sprites.

Billboarded sprites were used to represent some tall blades of grass and trees in my scene – the method is computationally cheap but is an incredibly powerful deception at far to moderate range. The billboards constantly follow the camera to provide a pseudo-depth when they are in actuality simple 2D primitives.

In my scene, I've implemented cylindrical billboarding – that is, the billboards are restricted to only rotate around the Y axis. This was decided on by a purely logical basis – trees and grass do not pitch up toward your eye as you look down upon them.

Implementing this specific functionality was more complex than spherical billboarding. The billboard could not simply follow the camera's look at vector, hence some logic would have to be applied to determine how to rotate around the Y axis only while still following the camera.

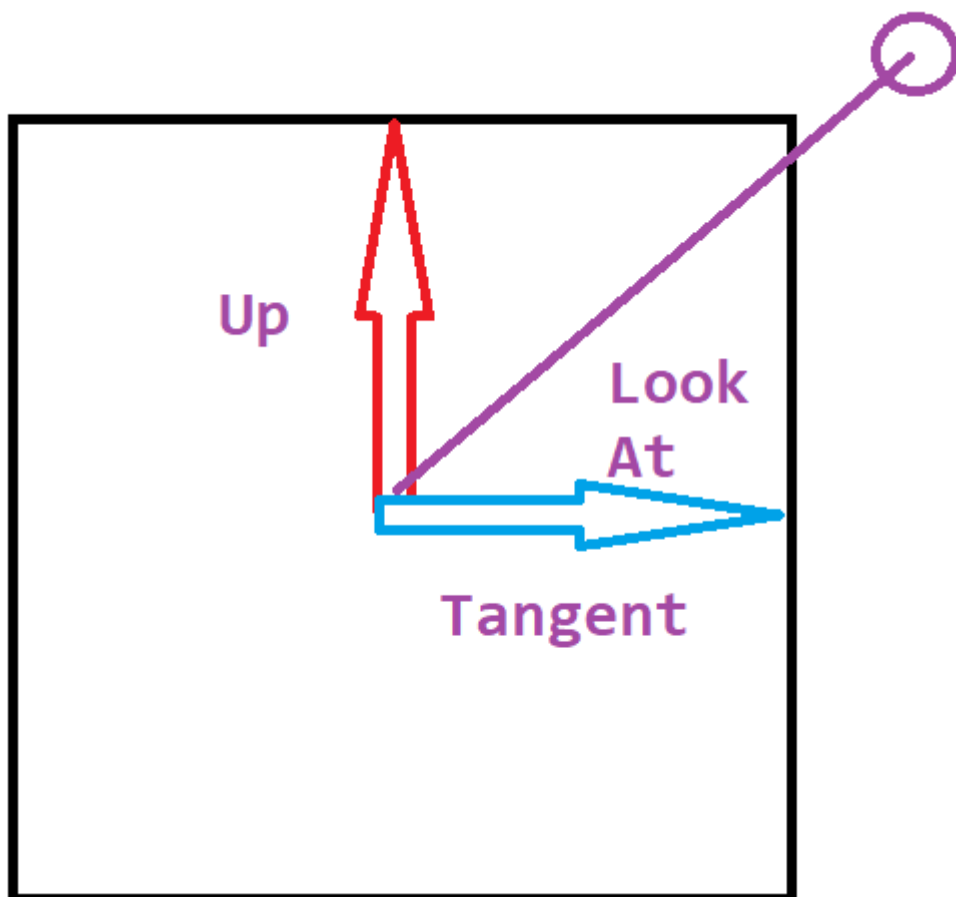
The first step however was to determine how to represent my billboards. I decided to use a 2D quad to use as my billboard, passing in a point representing the top left corner to the geometry shader and setting the maximum vertex count to 4. A custom 1x1 2D quad billboard mesh was created with set vertices, texture coordinates and normals. The quad size would be overridden and sent to the geometry shader depending upon whether a blade of grass or tree was being drawn. This was achieved by passing in a GeometryType enum value when generating the billboard meshes – the C++ side would then send the correct values to the position buffers in the geometry shader.

With the billboard representation implemented, the hardest step was to implement the cylindrical billboard functionality itself.

The cylindrical effect was achieved by moving the billboard relative to the tangent between the camera look at vector and the billboard's up vector.

To determine the look at vector, the camera's position was subtracted from the world position of the point. The y was then set to 0 and the vector normalized.

The tangent could then be calculated by performing the cross product of the look at vector and the up vector – as the billboard should only ever rotate in the Y axis, the up vector can be assumed and set to (0, 1, 0). A diagram of this concept is below:



With the tangent calculated, setting the position of each vertex was as simple as multiplying the tangent by each vertex's X position, adding on the Y position and then setting into homogenous form.

With this now determined, calculating output positions as well as lighting positions is the same as before – multiplying by the world, view and projection matrices.

Finally, the output could be appended to the triangle stream and the strip restarted once each vertex has been handled.

However, one final issue was encountered. Although alpha blending was enabled, strange texture repetitions/artifacting issues were observed in areas in which the billboard should have been fully transparent.

Upon research of the issue (the sources of said research have been referenced), it was determined that the rendering blend state would have to be changed to use Alpha to Coverage. Upon setting this, the issue was solved, and the billboards were completed.

The evidence photo below shows the pleasing result of the trees:



Critical Reflection – Lessons Learned

Considering the amount of work required to implement a good quality scene and the time constraints, it cannot be understated the amount of knowledge gained and lessons learned from creating the scene.

The first lesson learned was that of effective debugging. A huge constraint with shaders was that there are very difficult to debug vs CPU code. This issue was compounded by the time constraints of the project. So it became important very early on to develop ways of effectively debugging code.

This included researching into using tools such as RenderDoc and Visual Studio's Graphics Debugger. Learning how to use these tools to review shader code made a difficult task significantly easier and will be great to apply to future projects.

The second lesson learned was effective time management and planning. The introduction of using Gantt Charts and submitting both a proposal and milestone was especially helpful at laying out the individual tasks required to complete the project. While the scale was initially daunting, breaking the problem down into smaller chunks and allocating enough time to each made meeting the deadline with all the requirements significantly easier.

The third and final lesson is a more general one of graphics overall – the vast amount of techniques introduced such as normal mapping, displacement mapping, billboard, etc are significantly useful to be aware of for future applications. These techniques and the experience gained with them should allow more detailed scenes to be developed in the future at a lesser computational and implementation time expense.

Critical Reflection – Shortfalls, Introduction

Although the produced scene has worked out very well in the end, many shortfalls were experienced along the way and had to either be dealt with or avoided.

Critical Reflection – Shortfalls, Hut Reflection

The first shortfall encountered was that of reflecting the hut. Although the skybox reflects properly now, consistent issues were encountered with either the skybox reflection moving with the camera, or the hut reflection moving with the camera.

This is almost certainly due to an issue with how the geometries are drawn during the reflection pass, hence when the pixel shader samples the reflection render texture, the outputted results are incorrect.

Even with multiple attempts, I was unable to get both working consistently well together.

A decision was made to use the correct skybox and omit the hut reflection. This is due to the skybox being a much larger geometry and is considered more important to reflect.

The produced effect is still very pleasing indeed but should be expanded/fixed to include the hut and the terrain as will be mentioned later in the future expansion section.

Critical Reflection – Shortfalls, Generated Geometry Placement

The second shortfall encountered was that of placing the generated geometry from the geometry shader.

Initially, the plan was to implement a grass/tree placement map, where each specifically coloured pixel would represent a position of a piece of geometry.

However, it became clear that this task would not be as straightforward as it seemed. When the geometry shader is called, the positions need to be already known and passed into the vertex shader to be generated. This caused a huge problem – I was unable to find a way to have a shader sample the map texture but output the results to the CPU, to then pass onto the beginning of the geometry shader.

Experiments were attempted with the compute shader. However, this also appeared inherently complex – a decision was made to conserve time and abandon this idea.

One final attempt was to write CPU code to analyse the map file. However, many common formats such as PNG and JPG required external libraries to parse. While an attempt to manually parse a BMP file was successful, the BMP file size was not only orders of magnitudes bigger in file size but also very slow to parse and added a significant amount of load time.

The decision was then made to simply generate the geometry by hand. 8 blades of grass and 2 trees were added.

While still perfectly adequate at demonstrating the requirements, from an artistic point of view, it would have been much nicer to try to add more geometry as the scene looks sparse in some areas.

However, time was considered a much more valuable resource and so was allocated to more important tasks and implementations required to fulfil the coursework brief.

To solve this issue in future, I would incorporate a PNG reading library to read a smaller file size PNG map on the CPU side. PNGs would reduce load times, allow quicker parsing via BMPs and the map would remove the need to hand place the geometry.

Critical Reflection – Shortfalls, Proposal Time Shift

The third shortfall was that of timing and implementation order, specifically regarding what was submitted in the original proposal.

Although almost of all the intended features proposed were implemented, it is important to mention that the order in which they were completed in relation to the Gantt Chart was different.

For example, it was intended to implement tessellation in part 2 of the project and to implement billboarding in part 1. This happened in reverse during implementation.

The reasons for this were relatively straightforward – these topics were covered earlier in the module, so it made sense to implement tessellation immediately after it was covered to conserve time, rather than wait until billboarding was covered to implement it first and then implement tessellation as per the Gantt Chart plan.

Although this didn't have any negative effect toward the project as neither of them were dependent on one another, and nor did it consume more time than planned, this may not be the case in other projects.

Hence in future, it would be wise to confirm the delivery schedule of content to ensure that the Gantt Chart and project planning could logically work with it.

Critical Reflection – Shortfalls, Missing Proposal Features

Although fortunately few and far between, it is important to mention that some features proposed were not implemented in the project. These include:

- Gerstner Waves
- Lens Flare

However, while it is true that both above have been replaced adequately with alternatives, it is important to consider the shortfalls and reasons behind the failure to implement them in order to avoid the situation in future.

Regarding Gerstner waves, the primary reasons were the mathematical complexity involved and the visual of the final effect being potentially too extreme for the setting of the scene.

Both issues could have been avoided had more intrinsic and rigorous initial research into Gerstner waves taken place. This also applies to its replacement with the DUDV texture.

If specific implementations of Gerstner waves been looked at in more detail to observe the code and the final visual effect, then it could have been discounted before the proposal. In the same way, this would have logically led to further research in water simulation techniques and potentially resulted in finding the eventual replacement method with a DUDV texture far earlier.

Although it took less time to implement than Gerstner waves were planned to (which is positive), it could have easily become longer which could potentially derail timescales of projects in the future. Put simply, adequate planning and research saves time and effort in the long run.

Lens flare was also planned but was swapped with motion blur. This is due to a simpler reason – it was decided after being introduced to motion blur that it could be an easier natural extension and complexity addition from the lab task and be complimentary to bloom, while conserving time. Rather than discard the lab work completed and having to spend more time on implementing and researching lens flare from scratch, extending and complicating already completed work would be beneficial for time. This would allow more time to be spent on implementing other areas, fixing bugs and polishing. Much the same reasoning applies to that of the time shift – had the delivery schedule been reconsulted, it could have been determined that researching blurs could have been performed earlier and decided upon during planning.

Critical Reflection – Application Extension

Multiple changes and additions could be made to extend and improve the scene in the future.

Firstly, I would further extend the ImGui controls to include an option to set the glow blur/scene blend factor, as well as spot light roundness power. Both would take some time to add as extra buffers would have to be setup and general tweaks made to both shaders but would allow extra control over the visuals of the post processing effects and lighting respectively.

Secondly, I would add terrain reflection i.e. have the terrain reflect onto the water, as well as fixing the hut reflection. Although the skybox produces a satisfying effect, adding terrain and the hut would cement and finalize the effect and become very pleasing overall.

Thirdly, I would consider having a rotating skybox and directional light, perhaps adding a day and night cycle to further produce interesting lighting effects and add a whole new level of depth and detail to the scene.

Finally, if time permitted, I would have liked to have incorporated some form of complex weather system. This would have allowed a natural incorporation of other visually appealing and interesting effects.

For example, god rays could have been used during a very sunny period of weather. When raining, water droplets could be observed contacting the surface in the lake water, etc.

Generally, while these additions would be artistically appealing, they were too overkill and time consuming for the project. If more time was allocated and/or the scope of the project was increased, these would have been seriously considered as potential additions to the scene.

Conclusion – What would I do differently?

To conclude, I believe the process of implementing the scene and indeed the produced result was very successful.

However, lessons have been learned and will be applied in the future. Hence, if I could approach the project again with the benefit of hindsight, I would make changes.

The biggest change I would make would be to perform more research into my desired effects/features to implement before solidifying them in the proposal. Although the changes/omissions made did not negatively affect the timescale the scene and were few and far between this could be very negative to other projects.

I would also ensure that I had researched debugging techniques and processes in further detail as well. Ineffective debugging can become a huge source of time waste. Hence, had I known prior about RenderDoc and Visual Studio's Graphics Debugger, less time would have had to be spent learning how they work, and more time could have been spent on the important parts – debugging and solving issues. In future, I would ensure that I correctly understand and appreciate differences in writing GPU based code so that time can be saved.

References

The Devil in the Details, Gaussian Kernel Calculator (2014). Available at: <http://dev.theomader.com/gaussian-kernel-calculator/> (Accessed: 21 November 2019)

ThinMatrix, OpenGL Water Tutorial (2015). Available at: https://www.youtube.com/watch?v=HusvGeEDU_U&list=PLRIWtlCgwaX23jiqVByUs0bqhnaINTNZh (Accessed: 21 November 2019)

RasterTek, Small Body Water (2016). Available at: <http://www.rastertek.com/tertut16.html> (Accessed: 21 November 2019)

Jason Zink, Practical Rendering and Computation with Direct3D 11 (2011), 1st Edition

Braynzar Soft, Billboarding (Geometry Shader) (2015). Available at: <https://www.braynzarsoft.net/viewtutorial/q16390-36-billboarding-geometry-shader> (Accessed: 21 November 2019)

GameDev.net, [DX11] Deferred Rendering and alpha-to-coverage (2011). Available at: <https://www.gamedev.net/forums/topic/617756-dx11-deferred-rendering-and-alpha-to-coverage/> (Accessed: 21 November 2019)

MSDN, D3D11_BLEND_DESC structure (2018). Available at: https://docs.microsoft.com/en-us/windows/win32/api/d3d11/ns-d3d11-d3d11_blend_desc (Accessed: 21 November 2019)

All maps and hut model created by Brandon Filer. <http://brandonf.net/>

Grass, Rock, Dirt and Sand assets from textures.com. <https://www.textures.com/>