CMP301 – Coursework Report

1800997 – Stylianos Zachariou

Overview

For my coursework project, I created my own landscape inspired by the Norwegian fjords. The scene consists of dynamically tessellated mountains and water, billboarded trees, three customizable lights and models (for casting shadows).
Vertex manipulation is demonstrated when creating the mountains, using a height map and the water waves, using sin-cos waves. For post processing, the scene can be displayed using the bloom effect and a mini map is created showing the position of the user. Moreover, the scene contains three customizable lights. All three lights can be turned off or changed to a different type of light source. Each light has its own shadow map and mesh, showing its position. The mountains and water use 12 control point dynamic tessellation and interpolation to make the transition of quality smooth between the tessellated and not tessellated parts. Finally, the billboarded trees are made using the geometry shader to create two intersecting quads that always face the user.

Controls

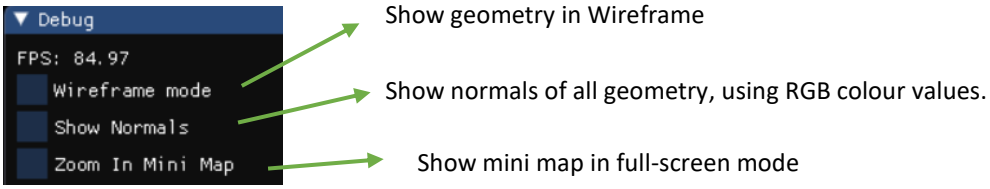

Figure 2-General Settings

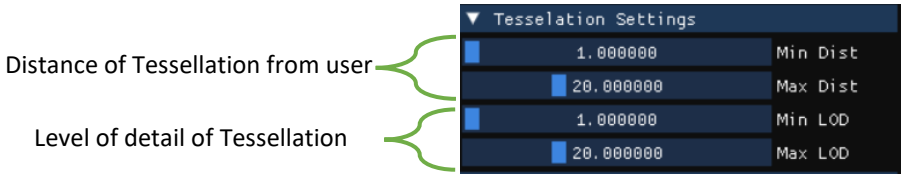

Figure 1-Tessellation Settings

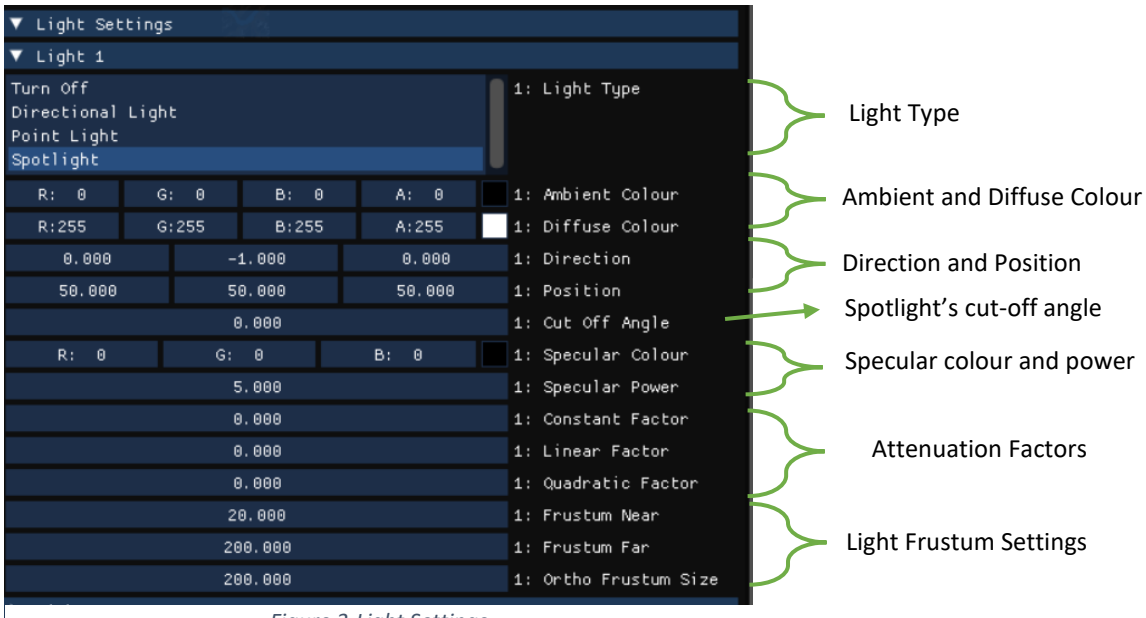

Figure 3-Light Settings

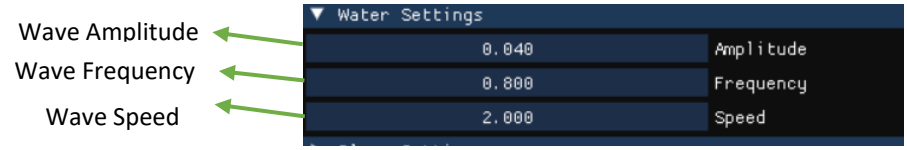

Figure 5-Water Wave Settings

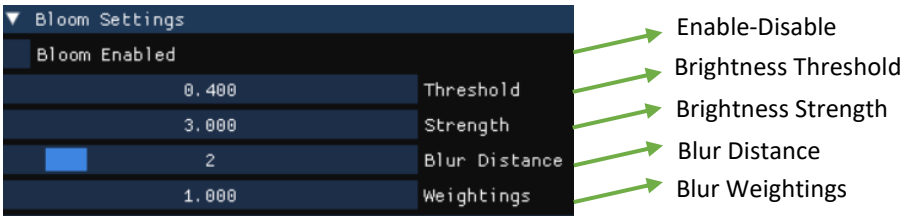

Figure 4-Bloom Settings

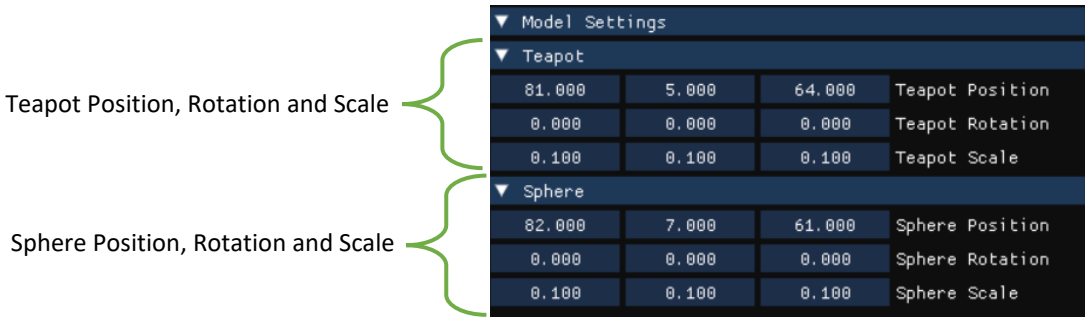

Figure 6-Model Settings

Techniques Used

Vertex Manipulation

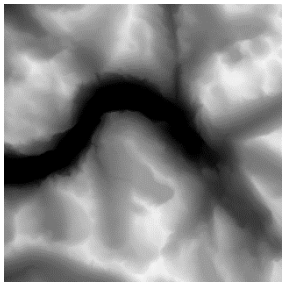

Figure 7- Geirangerfjord Height Map

To create the mountains in my scene, vertex manipulation was used. Firstly, an online height mapping tool was employed to get the height map of a Norwegian fjord. (Mapzen, 2021) The height map (Figure 7) displays the height values of the fjord with 1 (white) being the highest point and 0 (black) the lowest point. Since the mountains are tessellated, instead of the vertex shader, the height map is passed in the domain shader, as a 2D texture along with a Sampler. Then in the "getHeight" function (Figure 8), the domain shader samples the texture and returns the colour value for the current texture coordinate. The colour value RGB are added (which makes the mountains steeper since the range now is 0-3) and then multiplied by the 1/3 of the highest desired point. The value used here was 7, therefore the highest mountain top is 21 units high.

For the vertex manipulated geometry to be correctly lit, new normals must be calculated. This can be done by creating four tangent vectors, with the current vertex's position and its four adjacent vertices' positions, and finding the cross products of those vectors. The average of the four cross products is the normal of the current vertex. (Figure 9)

```
float getHeight(float2 uv)
{
    //Get height map colour at position
    float4 textureColour = heightMap.Sample(sampler, uv, 0);

    float height;
    height = (textureColour.x + textureColour.y + textureColour.z);
    height *= 7;

    //Lower the lowest parts for water
    if(height == 0)
    {
        height = -5;
    }
    return height;
}
```

Figure 8 - Mountain Vertex Manipulation Psuedocode

```
    //Adjacent vertices Y Values
    float adjacent[4];
    adjacent[0] = getHeight(texCoord + float2(0, diffy));
    adjacent[1] = getHeight(texCoord + float2(diffx, 0));
    adjacent[2] = getHeight(texCoord + float2(0, -diffy));
    adjacent[3] = getHeight(texCoord + float2(-diffx, 0));


    //Get Vectors between positions
    float3 vector1 = float3(vertexPosition.x, adjacent[0], vertexPosition.z + diffy) - vertexPosition;
    float3 vector2 = float3(vertexPosition.x + diffx, adjacent[1], vertexPosition.z) - vertexPosition;
    float3 vector3 = float3(vertexPosition.x, adjacent[2], vertexPosition.z - diffy) - vertexPosition;
    float3 vector4 = float3(vertexPosition.x - diffx, adjacent[3], vertexPosition.z) - vertexPosition;

        //Cross 1 & 2
    float3 cp1 = float3(((vector1.y * vector2.z) - (vector1.z * vector2.y)), ((vector1.z * vector2.x) -
(vector1.x * vector2.z)), ((vector1.x * vector2.y) - (vector1.y * vector2.x)));
        // Cross 2 & 3
    float3 cp2 = float3(((vector2.y * vector3.z) - (vector2.z * vector3.y)), ((vector2.z * vector3.x) -
(vector2.x * vector3.z)), ((vector2.x * vector3.y) - (vector2.y * vector3.x)));
        // Cross 3 & 4
    float3 cp3 = float3(((vector3.y * vector4.z) - (vector3.z * vector4.y)), ((vector3.z * vector4.x) -
(vector3.x * vector4.z)), ((vector3.x * vector4.y) - (vector3.y * vector4.x)));
        // Cross 4 & 1
    float3 cp4 = float3(((vector4.y * vector1.z) - (vector4.z * vector1.y)), ((vector4.z * vector1.x) -
(vector4.x * vector1.z)), ((vector4.x * vector1.y) - (vector4.y * vector1.x)));


    //Find average of Cross products
    vertexNormal = (cp1 + cp2 + cp3 + cp4)/4;
```

Figure 9-Mountain Vertex Normal Calculations

If the normals' coordinates are translated to RGB colour values, the normals positive to the: x-axis can be seen as red, y-axis can be seen as green, z-axis can be seen as blue. (Figure 10)

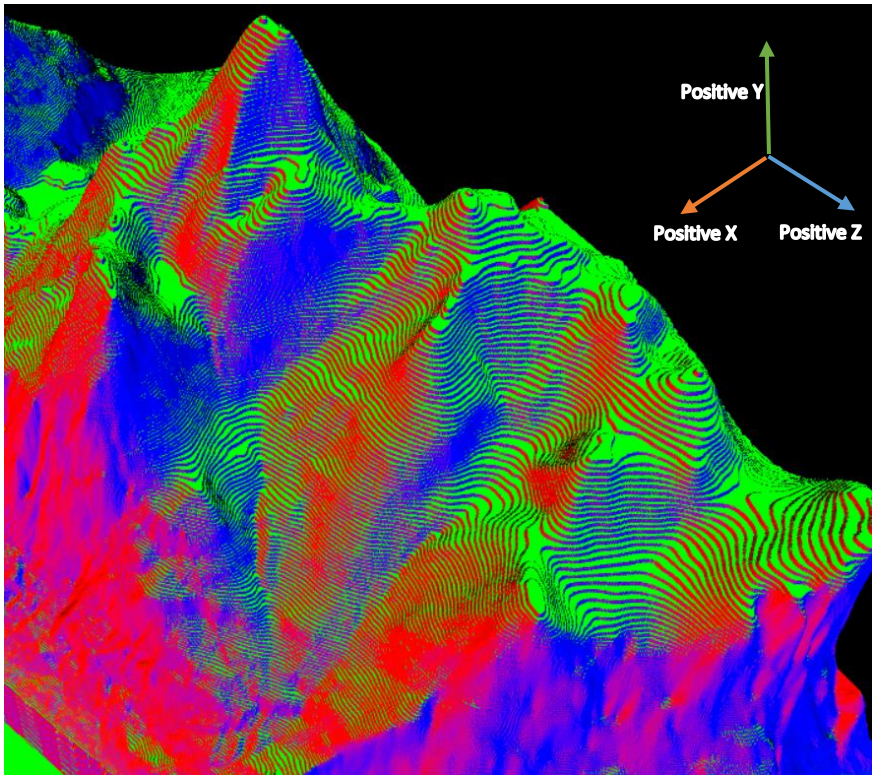

Figure 10-Vertex Manipulated Mountains with calculated normals

Vertex manipulation was also used when rendering the water. The water mesh is dynamically tessellated; hence, the vertex manipulation takes place in the domain shader. To create water waves, sin and cos waves are used with amplitude, speed, and frequency variables. These variables, along with the simulation time are passed to the domain shader using a buffer. Then, the shader performs a y-axis sin transformation (Figure 11) and an x-axis cos transformation (Figure 12) for the four control points of the tessellated quad and interpolates to find the position of the current vertex.

```
position.y = position.y + sin(position.x * frequency + -time * speed) * amplitude;
```

Figure 11- Sin Wave Y Transformation

```
position.x = position.x + cos(position.z * frequency + -time * speed) * amplitude;
```

Figure 12-Cos Wave X Transformation

For the sin wave, the y-positions are affected by their initial x-positions, where in the cos wave, the x-positions are affected by their initial z-positions. The simulation time is needed to make the waves move, because it adds to the position that's inside the cos and sin functions. The speed and frequency variables directly affect the time of the simulation and position of the wave respectively. The amplitude variable then multiplies the whole cos or sin function to "exaggerate" the transformation. Therefore, two waves are created, the first moving in the y-axis direction and the other moving in the x-axis direction. (Figure 13)

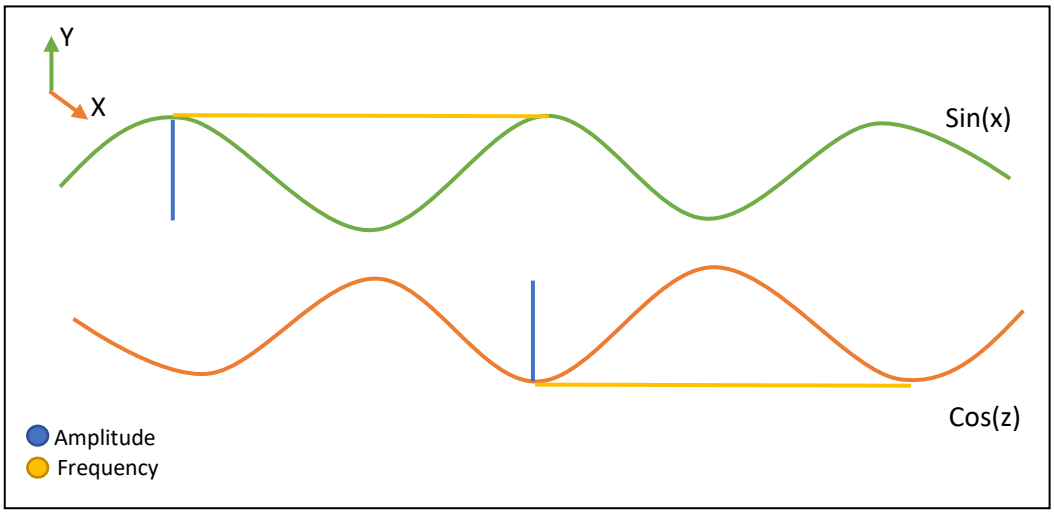

Figure 13-Waves Created by Domain Shader

For each wave a separate normal calculation had to be done. Since the sin wave used the initial x-position for creating the wave, its x-normal is the -cos function because the derivative of sin(x) is cos(x). (Figure 14)

```
normal.x = -(cos(position.x * frequency + (-time * speed))) * amplitude;
```

Figure 14-Sin Wave Normal Calculation

Following the same principle, the cos(z) wave's derivative is -sin(z), therefore, its normal z-normal is sin(z). (Figure 15)

```
normal.z = sin(position.z * frequency + (-time * speed)) * amplitude;
```

Figure 15-Cos Wave Normal Calculation

The y-normal is set to 1 since the normals of the waves should always point upwards. As with the mountains, the water normals be can seen when mapping the normal's coordinates to RGB values. (Figure 16)

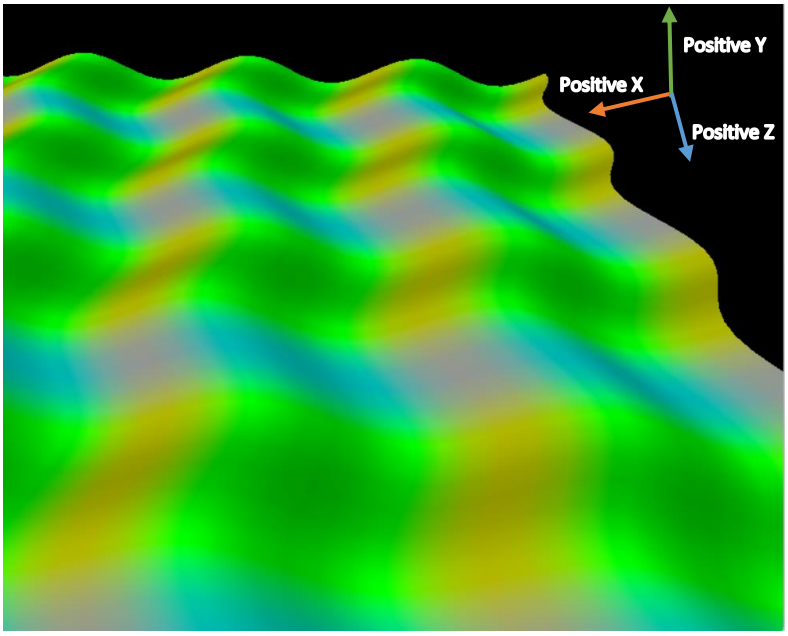

Figure 16-Water Normals Mapped to RGB

Post Processing

In the scene, the user can choose to enable or disable the Bloom post processing effect. Bloom mimics the blurriness bright objects have when captured by a camera. For this to be achieved, the bright parts of the image must be detected, blurred, and added to the original image. The first step is rendering the scene on to a render texture. Then, that render texture is passed into the brightness detection pixel shader as a 2D texture along with a sampler. To make the effect more customizable two more settings (threshold and strength) are passed using a buffer. To get the brightness level of each pixel, the shader gets the texture colour using the render texture and sampler and performs a grayscale calculation (Figure 17) to get a value from zero to one.

```
Brightness = (textureColour.x+textureColour.y+textureColour.z)/ 3;
```

Figure 17-Brightness Calculation

Using an if statement, the shader returns the texture colour multiplied by the strength variable if the brightness level is higher than the threshold value. If the brightness is less, then the shader returns the colour black. (Figure 18) This means that only the brighter parts of the scene will be returned. The image returned is rendered on a different brightness detection render texture. (Figure 19)

```
if (brightness > threshold)
{
    //Then is bright
    return strength * textureColour;
}
else
{
    //Is not bright
    return float4(0, 0, 0, 1);
}
```

Figure 18-Brightness Detection

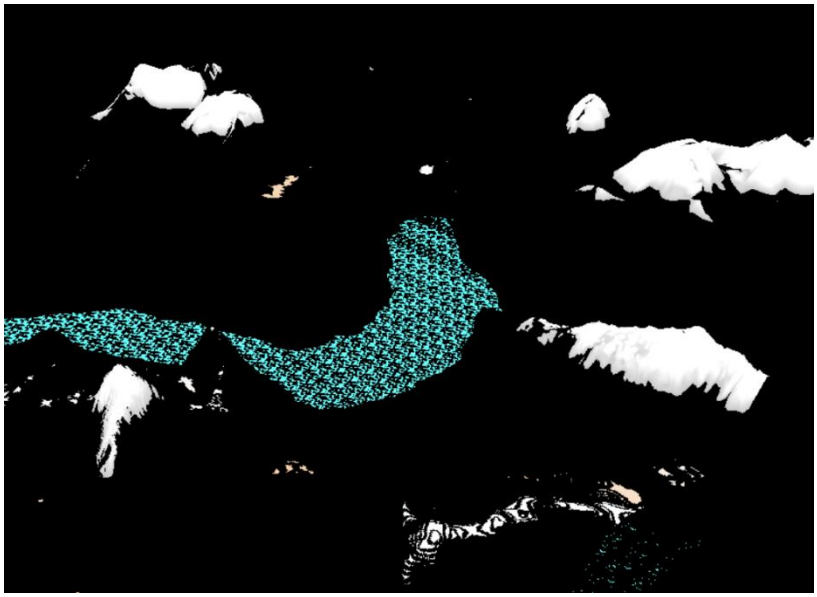

Figure 19-Brightness Shader Result

The next stage is blurring the previously created brightness texture and it's done by firstly using a horizontal blur and then a vertical blur pixel shader. The two shaders are identical except the one is applying a gaussian blur on the x-axis and the other a gaussian blur on the y-axis using 1D kernels. The shaders need the render texture and a sampler along with three more customizable variables (screen width/height, neighbours, and weightings), passed using a buffer. At first, the shaders calculate the weight values for each neighbour, depending on the number of neighbours by using the gaussian blur formula, substituting the standard deviation variable with the weighting variable and the distance variable with the neighbour distance from the current position. (Figure 20)

```
for (int i = 0; i < neighbours; i++)
{
    float p = -(pow(i, 2)) / (2 * pow(weighting, 2));
    weight[i] = (1 / (sqrt(2 * pi * pow(weighting, 2)))) * pow(2.71828, p);

}
```

Figure 20-Gaussian Weighting Calculation

Subsequently, the colour of the current pixel is measured by adding all the neighbours' colour values multiplied by each's weighting. (Figure 21) The horizontal blur shader renders the result on a horizontal blur render texture, which is then taken by the vertical blur shader which renders it's result on a vertical blur render texture.

```
for (int j = 1; j < neighbours; j++)
    {
        if (input.tex.x + texelSize * -j >= 0)
        {
            colour += shaderTexture.Sample(SampleType, input.tex + float2(texelSize * -j,0.0f)) * weight[j];
        }

        if (input.tex.x + texelSize * j <= 1)
        {
            colour += shaderTexture.Sample(SampleType, input.tex + float2(texelSize * j, 0.0f)) * weight[j];
        }
    }
```

Figure 21-Blurred Colour Calculation for Horizontal Blur

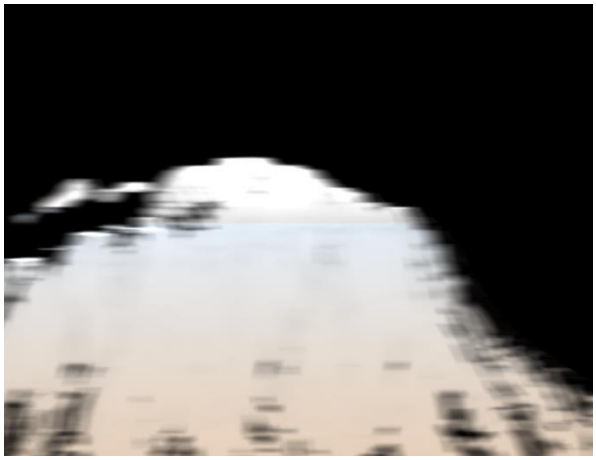

Figure 232-Result of Horizontal Blur

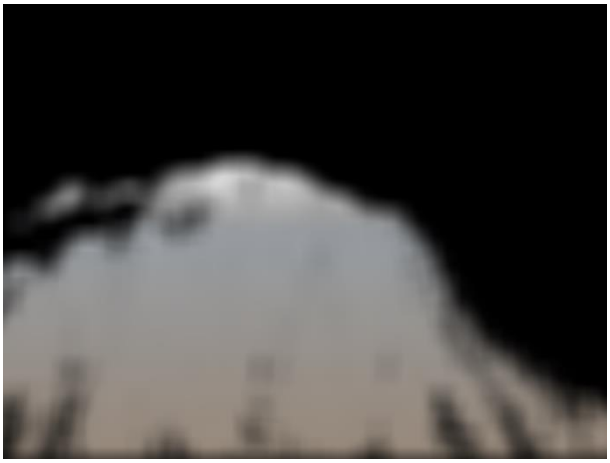

Figure 223-Result of Horizontal and Vertical Blur

If Bloom is enabled, the scene render-texture and the blurred render-texture are both passed into a blend pixel shader. This shader samples both textures and adds their colour values. (Figure 24)

```
float4 textureColour1 = sceneTexture.Sample(Sampler0, input.tex);
float4 textureColour2 = blurredTexture.Sample(Sampler0, input.tex);

//Add them together and return
return (textureColour1 + textureColour2);
```

Figure 24-Blend Textures Calculations

The blended render texture is the one that is displayed to the user. (Figure 25)

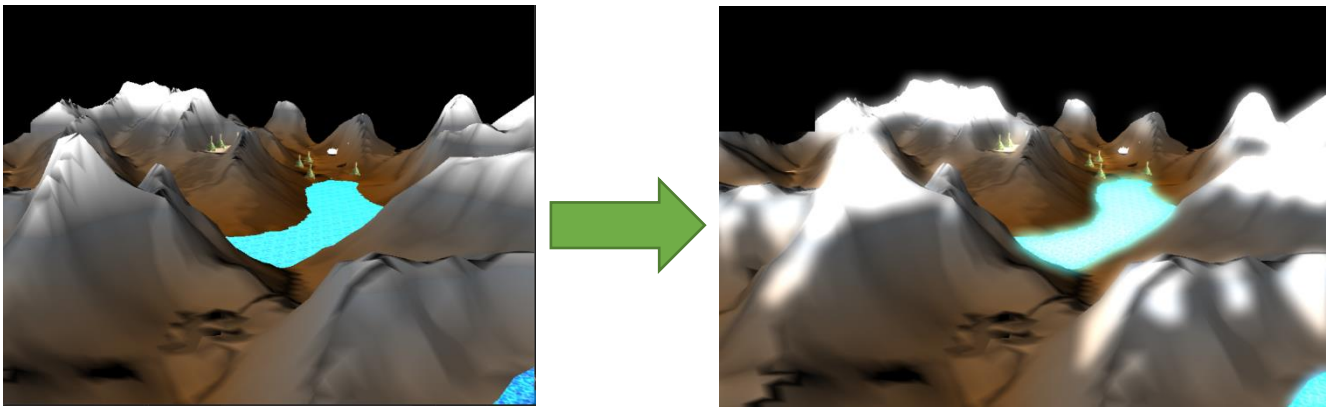

Figure 25-Scene Before and after Bloom

Moreover, post processing is used when creating the mini map. To do this, three stages were needed: rendering the scene using a top down camera, rendering the position of the player on the scene using a red circle and blending the two images together. The first stage is done by just setting a top down render texture as the render target and then rendering every object of the scene using the top down camera's view Matrix and the renderer's ortho matrix, instead of the projection matrix. Then, on a mini map render texture, we use the mini map vertex shader to determine the players position in screen space. The vertex shader uses the position of the camera and the screen size which are both passed using a camera buffer and outputs the camera's position multiplied by the world, view, and projection matrices (Figure 26) and screen size to the pixel shader.

```
output.camera_position = mul(position, worldMatrix);
output.camera_position = mul(output.camera_position, viewMatrix);
output.camera_position = mul(output.camera_position, projectionMatrix);
```

Figure 26-Camera Position Multiplied by Matrices

To calculate the camera's position into screen space, the pixel shader firstly divides the camera's xyz-coordinates with its w-coordinate, making them homogeneous. The xy-coordinates are then halved and added 0.5 to put them in the range of 0-1. They are then multiplied by the screen size to place the xy-coordinates in the range 0-screen space. (Figure 27)

```
input.camera_pos.xyz /= input.camera_pos.w

input.camera_pos.x *= 0.5
input.camera_pos.y *= -0.5

input.camera_pos.x += 0.5
input.camera_pos.y += 0.5

input.camera_pos.x *= input.screen_space.x
input.camera_pos.y *= input.screen_space.y
```

Figure 27-Camera Position to Screen Space

After finding the position, the shader draws a red pixel if the current pixel is in the range of the radius a circle around the camera's position. (Figure 28)

```
if (abs(sqrt(pow((input.position.x - input.camera_pos.x), 2) +
pow((input.position.y - input.camera_pos.y), 2))) <= 5)
{
   textureColour = float4(1, 0, 0, 1);
}
```

Figure 28-Draws a red pixel if position is near to camera's position

The two render textures are then blended using the same pixel shader used to blend the textures when creating the bloom effect. The result is then displayed to the user. (Figure 29)

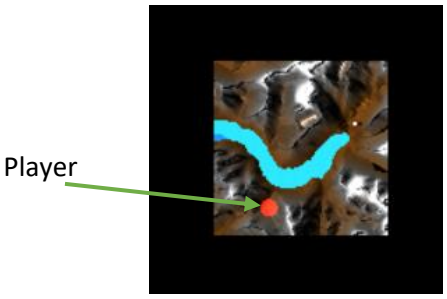

Figure 29- The Resulting Mini Map

Lights and Shadows

The scene includes three customizable light sources each one with its own shadow map and mesh. Everything, in the scene, is affected by light and cast shadows except the billboarded trees. Therefore, in each object's pixel shader, the lighting and shadow calculations take place. Each light can be turned off or changed into a directional light, point light or spotlight. The buffer is packed so that 3 of each of the following variables are passed to the pixel shaders using 16 byte chunks (1 for each light): ambient, diffuse, position, direction, specular colour, specular power, type, cut off angle, constant factor, linear factor, and quadratic factor. (Figure 30)

```
cbuffer LightBuffer : register(b0)
{
    float4 ambient[3];
    float4 diffuse[3];
    float4 position[3];
    float4 direction[3];
    float4 specular[3];
    float4 specularPower[3];
    int4 type[3];
    float4 cutOffAngle[3];
    float4 constantFactor[3];
    float4 linearFactor[3];
    float4 quadraticFactor[3];
};
```

Figure 30-Packed Light Buffer

A world position, a camera view vector and three light view positions are needed to calculate the lighting in the pixel shader. These are calculated in the vertex shader for non-tessellated objects and in the domain shader for tessellated objects. Three light view and projection matrices are passed along with the world, view, and projection matrices to the shader. The world position is calculated by multiplying the vertex position with the world matrix to get this vertex's world position. The camera view vector is the direction vector from the vertex's world coordinates to the camera's coordinates and is calculated by normalizing the vertex world position subtracted by the camera's position. Finally, the three light view positions are calculated by multiplying the vertex position using the world, light view, and light projection matrix for each light. (Figure 31)

```
//World Position
output.worldPosition = mul(vertexPosition, worldMatrix);

//Camera View Vector
output.viewVector = cameraPosition - output.worldPosition;
output.viewVector = normalize(output.viewVector);

//Light View Positions
for (int i = 0; i < 3;i++)
{
    output.lightViewPos[i] = mul(vertexPosition, worldMatrix);
    output.lightViewPos[i] = mul(output.lightViewPos[i], lightView[i]);
    output.lightViewPos[i] = mul(output.lightViewPos[i], lightProjection[i]);
}
```

Figure 31-Light Variable Calculations

In the pixel shader, an if statement checks what type each light is and calculates the lighting on the object accordingly. Firstly, if the light is turned off, no lighting calculations are done, rendering the object as black. If the light is a directional light, the diffuse, specular, and ambient light must be calculated. For diffuse lighting, the intensity is calculated by finding the dot product of the normal and the light vector, which for the directional light is the direction. Then, the diffuse colour is multiplied by the intensity and saturated to the range 0-1. When calculating specular lighting, the light vector and view vector are added and normalized. If the dot product of the normal vector and the normalized sum of the vectors are less than 0, then it becomes 0, so that it is not a negative value, if not then it remains as it is. That value is raised to the specular power variable. This returns the specular intensity at the current position. The specular light can be found by calculating the specular intensity with the specular colour. (Figure 32)

```
//Calculate Specular
float3 halfway = normalize(lightVector + viewVector);
float specularIntensity = pow(max(dot(normal, halfway), 0.0), specularPower);
return saturate(specularColour * specularIntensity);}
```

Figure 32-Specular Lighting Calculations

The ambient light RGB values are being multiplied by the alpha value and then added to the lighting, so that the user can change the opacity of the ambient lighting using the alpha channel. If the type of the light is a point light, the same lighting calculations are used with some differences. The distance between the position and the world position needs to be calculated, along with a light vector variable, which is calculated using the normalized result of the subtraction of the world position and the position, returning the direction vector between the light source and the current position. (Figure 33)

```
float3 dist = length(position[i].xyz - input.worldPosition);
float3 lightVector = normalize(position[i].xyz - input.worldPosition);
```

Figure 33- Distance and light vector calculations

With the point light and spotlight the attenuation also needs to be calculated. Three variables affect the attenuation: constant factor, linear factor, quadratic factor, and the distance (calculated before). To find the attenuation value, the constant factor is added with the linear factor multiplied by the distance and added with

the quadratic factor multiplied by the distance squared. (Figure 34) If the factor sum is less than 1 then its set to 1, since the attenuation is the inverse of the factors and if they are less than one, attenuation will act in the opposite way it is supposed to.

```
float factors = constantFactor + (linearFactor * dist) + (quadraticFactor * pow(dist, 2));
```

Figure 34-Attenuation Factor Calculation

While calculating the diffuse and specular light, of the point light, instead of the direction vector, the light vector is used. The diffuse colour is also multiplied by the attenuation factor, so that when the factors are higher, the less light will be shown. Finally, when the light is a spotlight all lighting calculations are the same as the point light's but, are only done if the angle from the light to the current position is less or equal to the cut off angle variable. The angle is found by a function which takes in the light vector and the direction of the light. That function uses the formula for finding an angle between two vectors: $\cos\theta = \frac{u \cdot v}{\|u\|\|v\|}$ where u = lightVector and v = lightDirection. (Figure 35)

```
//Calculate angle between Light Vector and Direction Vector
float angle = acos((lightVector.x * lightDirection.x + lightVector.y * lightDirection.y +
lightVector.z * lightDirection.z) /(sqrt(pow(lightVector.x, 2) + pow(lightVector.y, 2) +
pow(lightVector.z, 2)) *sqrt(pow(lightDirection.x, 2) + pow(lightDirection.y, 2) +
pow(lightDirection.z, 2))));

//Convert the angle to radians
angle *= 0.01745329252;
```

Figure 35-Calculating the angle for spotlight

To simulate shadows three more functions are needed, the first one getting the projective coordinates using the light view position variable. That function divides the light view position's xy-coordinates with the light view w-coordinate to make them homogeneous. Then, by multiplying with 0.5 and adding 0.5 the coordinates returned are texture coordinates from 0-1. (Figure 36)

```
float2 projTex = lightViewPosition.xy / lightViewPosition.w;

projTex *= float2(0.5, -0.5);
projTex += float2(0.5f, 0.5f);
return projTex;
```

Figure 36-Projective Texture Coordinates Calculation

The projected texture coordinates are then used in another function returning true if texture coordinates are in the range 0-1. The projected texture coordinates will also be used in a function checking if the current position is in shadow. That function samples a shadow map at the projected texture coordinates to get the depth value. Then, using the light view position z-coordinate and dividing it by w, the light depth value is returned; later, the bias is subtracted by it which helps eliminate shadow acne. Thereafter, the function returns true if the light depth value is higher than the depth value of the shadow map. (Figure 37)

```
// Sample the shadow map (get depth of geometry)
float depthValue = sMap.Sample(shadowSampler, uv).x;

// Calculate the depth from the light.
float lightDepthValue = lightViewPosition.z / lightViewPosition.w;
lightDepthValue -= bias;

// 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 (depthValue > lightDepthValue)
{
    return false;
}
else
{
    return true;
}
```

Figure 37-Function Checking If Point is in Shadow

Before rendering the scene, each shadow map is created by rendering all objects on the shadow map, using the lights view and projection matrix. These shadow maps are later passed in the pixel shaders as 2D textures. (Figure 38)

```
Texture2D depthMapTexture[3] : register(t0);
```

Figure 38-Shadow Maps passed to the Pixel Shaders

Combining everything together in the pixel shader, for each light, the ambient light is calculated and added to the light colour. Then, the projective texture coordinates are found and tested; if the current position lands on the shadow map, the function that checks the current position for shadows returns if the current position will be lit. The lighting calculations are only completed if the position is not in shadow. Finally, all three light colours are summed and returned by the pixel shader. (Figure 39)

```
for (int i = 0; i < 3;i++)
{
    //Calculate ambient Lighting
    float4 newAmbient = float4(ambient[i].xyz * ambient[i].w, ambient[i].w);

    // Calculate the projected texture coordinates.
    float2 pTexCoord = getProjectiveCoords(input.lightViewPos[i]);

    // Shadow test. Is or isn't in shadow
    if (hasDepthData(pTexCoord))
    {
        // Has depth map data
         if (!isInShadow(depthMapTexture[i], pTexCoord, input.lightViewPos[i], shadowMapBias))
            {
                DO LIGHTING CALCULATIONS
            }
        }
    }
}
    finalColour = lightColour[0] + lightColour[1] + lightColour[2];

    return saturate(textureColour * finalColour);
```

Figure 39-General Lighting and Shadows

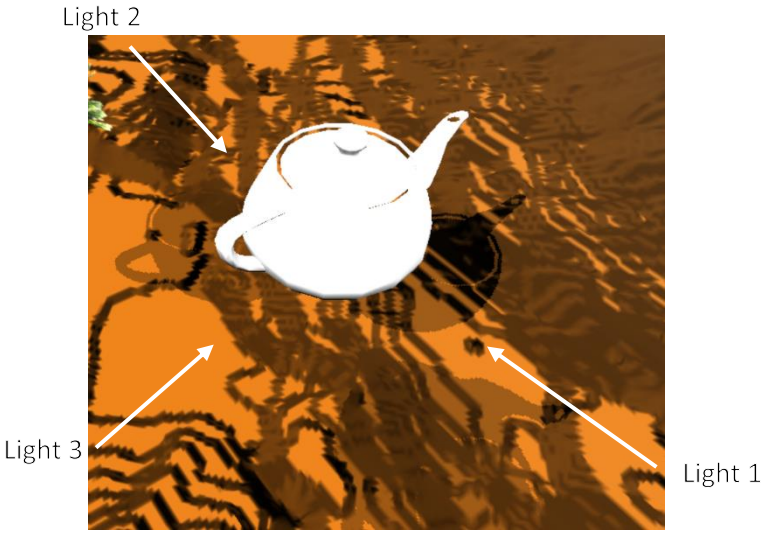

Figure 40-Multiple lights, with multiple shadows

Tessellation

Dynamic tessellation is used when rendering the water and mountains. The mesh must be more tessellated the closer it is to the player; therefore, the position of the camera must be transferred to the hull shader, using a buffer. In the same buffer, two more variables can be packed to increase the customizability of the tessellation: level of detail and distance of tessellation. (Figure 41)

```
cbuffer TessalationFactorBuffer : register(b0)
{
    float4 position; //16 bytes
    float2 minMaxLOD; //8 bytes
    float2 minMaxDist; //8 bytes
};
```

Figure 41-Tessellation Factor Buffer

The midpoint position of the patch is also needed to calculate the distance from the player and the level of detail. The edges of two adjacent patches must be tessellated the same amount so that no holes will appear. (Figure 42)

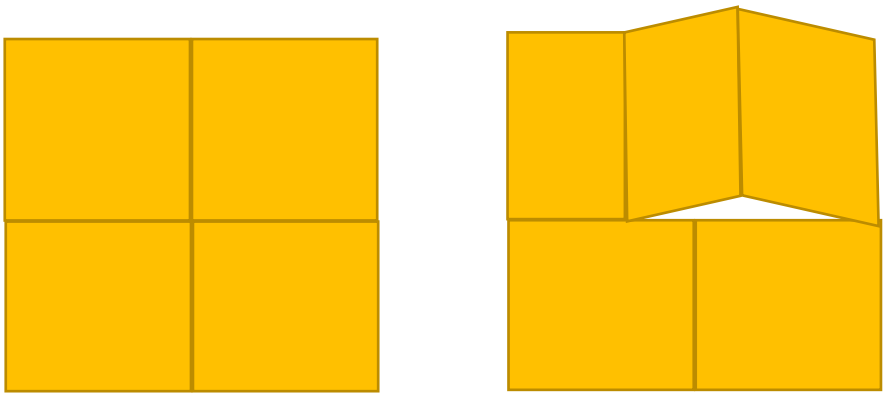

Figure 42- Left: Adjacent patched are tessellated by the same amount. Right: Adjacent patches are tessellated by a different amount

To prevent this issue, 12 control point tessellation is done where the four control points are the current patch's control points and the rest are the control points of adjacent patches. This is achieved using a custom tessellation plane where the indexes passed for each patch are 12 (Figure 43)

```
indices[0] = (z + 0) + (x + 0) * (TERRAIN_X_LEN + 1); //0
indices[1] = (z + 1) + (x + 0) * (TERRAIN_X_LEN + 1); //1
indices[2] = (z + 0) + (x + 1) * (TERRAIN_X_LEN + 1); //2
indices[3] = (z + 1) + (x + 1) * (TERRAIN_X_LEN + 1); //3

indices[4] = (z + 0) + (x + 2) * (TERRAIN_X_LEN + 1); //4
indices[5] = (z + 1) + (x + 2) * (TERRAIN_X_LEN + 1); //5

indices[6] = (z + 2) + (x + 0) * (TERRAIN_X_LEN + 1); //6
indices[7] = (z + 2) + (x + 1) * (TERRAIN_X_LEN + 1); //7

if (x > 0)
{
   indices[8] = (z + 0) + (x - 1) * (TERRAIN_X_LEN + 1); //8
   indices[9] = (z + 1) + (x - 1) * (TERRAIN_X_LEN + 1); //9
}
else
{
   indices[8] = (z + 0) + (x - 0) * (TERRAIN_X_LEN + 1); //8
   indices[9] = (z + 1) + (x - 0) * (TERRAIN_X_LEN + 1); //9
}

if (z > 0)
{
   indices[10] = (z - 1) + (x + 0) * (TERRAIN_X_LEN + 1); //10
   indices[11] = (z - 1) + (x + 1) * (TERRAIN_X_LEN + 1); //11
}
else
{
   indices[10] = (z - 0) + (x + 0) * (TERRAIN_X_LEN + 1); //10
   indices[11] = (z - 0) + (x + 1) * (TERRAIN_X_LEN + 1); //11
}
```

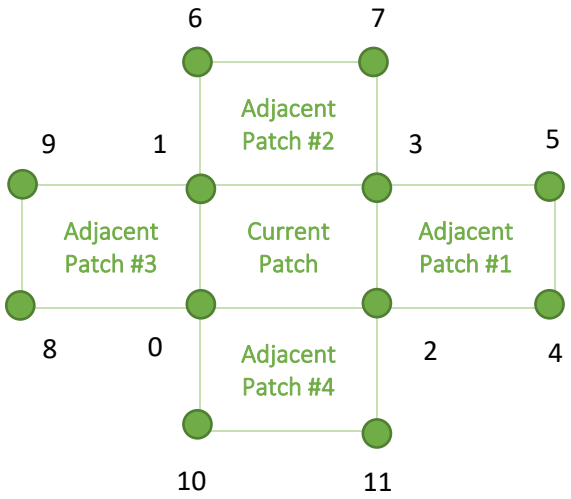

Figure 43-The 12 Control Points

Then, the midpoints of each patch's control points are found. The level of detailed for each is later calculated using the distance of the x and z axis between the patch's midpoint and camera's position, and applying a range calculation, so that the distance can be found in the range 0-1. (Figure 44)

```
float d = distance(f.xz, t.xz);
float maxD = minMaxDist.y;
float minD = minMaxDist.x;

return ((d-minD) / (maxD - minD));
```

Figure 44-Distance calculations for tessellation

Later, the distance value subtracted by 1 is used to linearly interpolate between the minimum level of detail variable and the maximum level of detail variable. (Figure 45)

```
lerp(minMaxLOD.x, minMaxLOD.y, saturate(1-d));
```

Figure 45-Level of detail linear interpolation

In the patch constant function, the level of detail for the current patch is calculated and set as the tessellation value for the 2 interior factors. Then, for each edge factor, the level of detail for the current patch and the level of detail of the adjacent patch are compared and the minimum is chosen as that edge's tessellation value. After having all the tessellation values, the quads are drawn in the domain shader using the patches initial four control points.

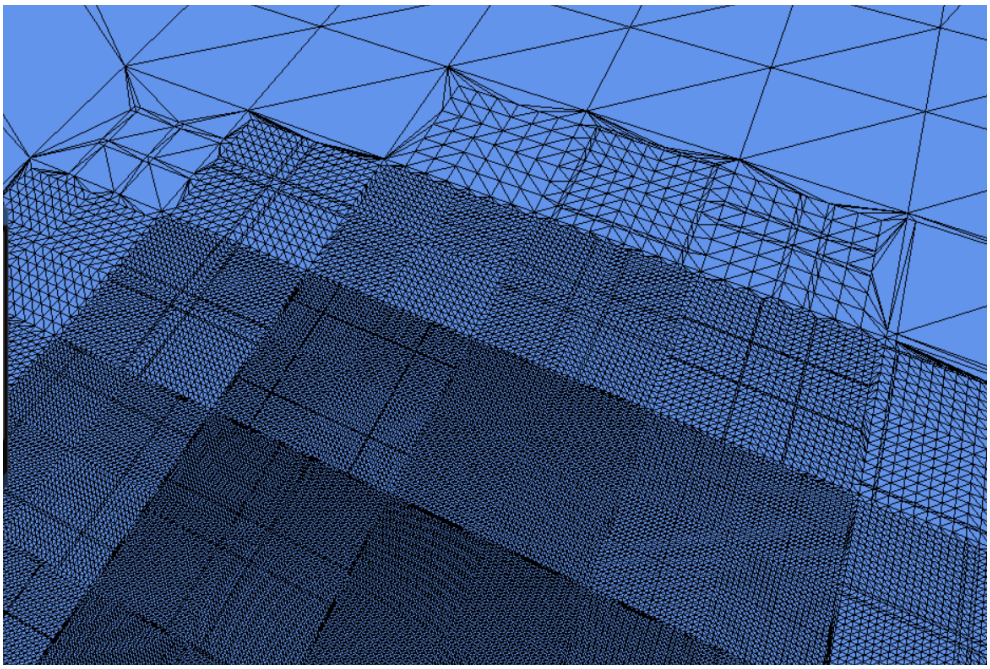

Figure 46-Active Dynamic Tessellation

Geometry Shader

The geometry shader is used when creating billboarded trees. A custom point mesh is used with a point where a tree is desired. Then the geometry shader creates two intersecting quads, textured as trees at each point's positions. The positions used for each vertex are held in a buffer in an array with a size of 8 (4 vertices for each quad). (Figure 47)

```
cbuffer PositionBuffer
{
    static float3 g_positions[8] =
    {
        float3(-1, 2, -1),
        float3(-1, -2, -1),
        float3(1, 2, 1),
        float3(1, -2, 1),
        float3(-1, 2, 1),
        float3(-1, -2, 1),
        float3(1, 2, -1),
        float3(1, -2, -1)
    };
};
```

Figure 47-Positions for new vertices

Since the trees must rotate depending on the players position, the direction vector from the player's position to the point position must be calculated. Using the arc tan of the direction vectors x and z coordinates (because we only want the trees to rotate in the y-axis), the angle of rotation is found. That angle is used to build a y-axis rotation matrix which is later multiplied by each "g_position" along with the world, view, and projection matrices. (Figure 48)

```
//Calculate Direction Vector
float3 dir = normalize(input[0].position.xyz - playerPos);

//Angle of Rotation
float angleY = atan2(dir.x, dir.z);
//Cos and Sin of angle
float c = cos(angleY);
float s = sin(angleY);

//Rotational Matrix
float4x4 rotYMatrix;
rotYMatrix[0].xyzw = float4(c, 0, -s, 0);
rotYMatrix[1].xyzw = float4(0, 1, 0, 0);
rotYMatrix[2].xyzw = float4(s, 0, c, 0);
rotYMatrix[3].xyzw = float4(0, 0, 0, 1);
```

Figure 48-Building the rotational matrix

The texture coordinates are calculated by dividing the "g_positions" by two (since the highest value is two) and added 0.5 so that the range of the texture coordinates is 0-1. (Figure 49)

```
//Calculate new Texture Coordinate
output.tex = g_positions[i] / 2 + 0.5;
output.tex.y = 1 - output.tex.y;
```

Figure 49- Texture Coordinate Calculation

The normals are initialized to point towards the positive x and y-axis and the negative z-axis for the first quad and towards the positive y-axis and negative x and the z-axis for the second. Then, it is rotated using the rotation matrix, created before, and the world matrix. (Figure 50)

```
//Calculate Normal
input[0].normal = float3(1, 1, -1);
input[0].normal = mul(float4(input[0].normal, 1), rotYMatrix);

output.normal = mul(input[0].normal, (float3x3) worldMatrix);
output.normal = normalize(output.normal);
```

Figure 50-Normal Calculation for the first quad

In the geometry shader the world position is also calculated and passed to the pixel shader for lighting calculations. (Figure 51)

```
//Calculate World Position
output.worldPosition = mul(vposition, worldMatrix).xyz;
```

Figure 51-World Position Calculation

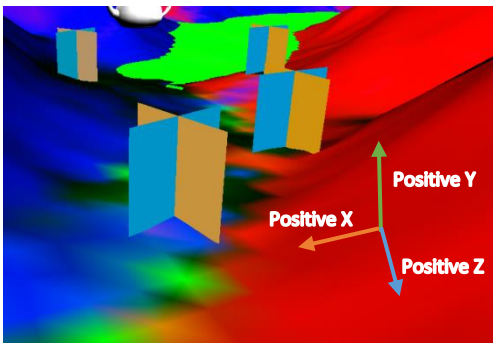

Figure 52-Billboarded quad normals

The billboarded pixel shader is responsible for printing the 2D texture on each quad and for calculating lighting. The lighting calculations on the trees are the same as all other geometry but without using specular lighting since trees wouldn't receive specular lighting. Moreover, the trees don't cast or receive shadows, because if they were included in the shadow map they would have been shown as two black rectangles instead of trees.

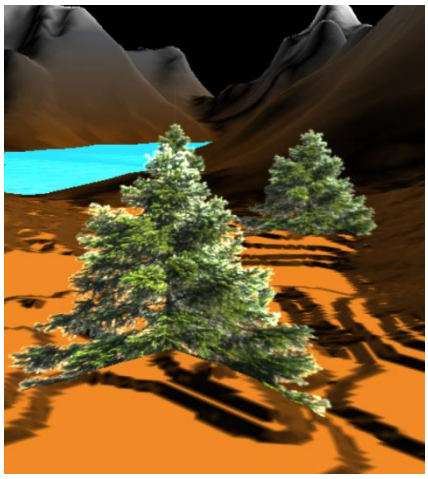

Figure 53-Textured Billboarded Trees

Reflection

I believe that the tessellated mountains and water, with their vertex manipulation, normal and texture coordinate calculations worked very well. I was also pleased with the bloom effect and the mini map. I spent a lot of time working on the lights and shadows, and I believe they also worked as intended. I was not very impressed with the billboarded trees, because I didn't have time to implement a way that they can cast shadows as trees instead of quads. I believe all techniques worked very nicely together to create a beautiful scene. All scene meshes can cast and receive shadows, and be lit correctly, showing that everything works in unison. Due to time limitations, I couldn't add realistic water to my scene. I would love for the water to create a reflection-refraction illusion and have used stacked normal maps to create the waves. (Pharr, 2005) Moreover, I would have loved to add some grass moving and giving the illusion of air, using the geometry shader. (Marco, 2020)

I tried making the scene as customizable as possible. The user can change:

- The tessellation factors (LOD and distance).
- The scale, position, and rotation of the models.
- The state of the bloom post processing effect and the bloom factors.
- Amplitude, speed, and frequency of waves.
- The type and state of each light
- All other light and frustum settings

The buffers transferring all the user inputs to the shaders were packed in 16 byte packs. The light buffers transferring the light settings to the shaders are using only float4 and int4 data types, so that all information is has the correct number of bytes, but this is wasteful when transferring information like the attenuation factors, who only need a single float. This could have been improved by adding padding and dividing the information into smaller buffers.

If I had more time, I would improve the shadowing techniques used. I did a lot of reading on soft shadows and cascaded shadow maps, and I don't think they would be very hard to implement. For the soft shadows, I would need a different shader to render the geometry in white, and shadows in black onto a render texture. Then, I would have blurred it and projected it from the camera's viewpoint to create the soft shadows. (RasterTek, 2016) The cascaded shadow maps would be a bit more complex. Each light would have needed three shadow

maps, each with a different resolution. The frustum would have been partitioned to find out where each shadow map would be (more resolution closer to the camera). More calculations would have had to be done for the position of the shadow maps depending on the light and camera orientation.

Finally, I have learned a lot while developing this scene and I am excited to work on shaders without a time limit to explore more complicated and efficient techniques.

References

Gold, B., 2018. Theme: Cartoon Fantasy. [Art].

Mapzen, 2021. Tangram Heightmapper. [Online]
Available at: https://tangrams.github.io/heightmapper/

Marco, G., 2020. Grass Shader. [Online]
Available at: https://giordi91.github.io/post/grass/

MART, P., 2019. Christmas Pine Tree PNG Image. [Art] (PNG MART).

Pharr, M., 2005. Chapter 18. Using Vertex Texture Displacement for Realistic Water Rendering. [Online].

RasterTek, 2016. Tutorial 42: Soft Shadows. [Online]
Available at: http://www.rastertek.com/dx11tut42.html