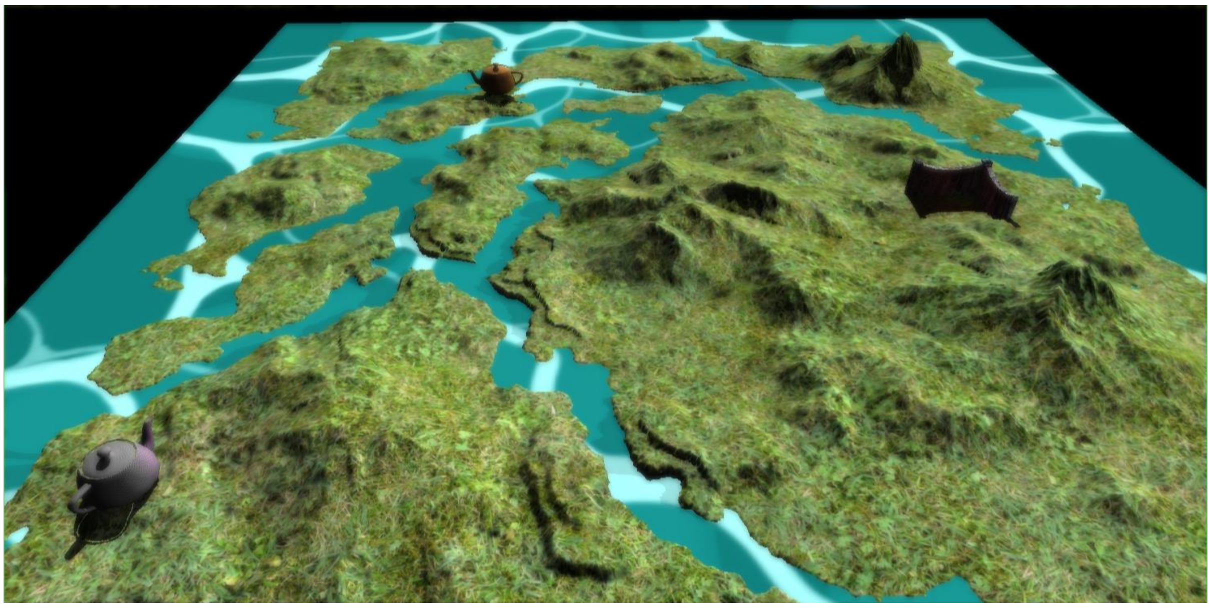
CMP301 COURSEWORK



OVERVIEW

DYNAMICALLY TESSELLATED TERRAIN

Satisfies the vertex manipulation and tessellation through manipulating a plane along a heightmap, then tessellating the said heightmap dynamically based on the distance between the camera and patch.

MULTIPLE SHADOWED DIRECTIONAL LIGHTS

Satisfies the shadow criteria and not handing back the same code that was given. This works by the shadow algorithm taking two maps, sampling each to see if the pixel is lit or in shadow, and only applying one of the directional light calculations of the pixel.

MULTIPLE POINT AND SPOT LIGHTS

In addition to multiple shadowed lights, there are multiple point and spot lights, to which more can be added easily. These lights do not have any shadows.

BLOOM POST PROCESS EFFECT

The post process effect is bloom. It works by getting the brightest parts of the scene, blurring them horizontally and vertically during which the alpha values and strength of the colours is changed by parameters, then merged back into to the original render texture, then finally rendered to the back buffer. This gives the effect of the brightest parts of the scene looking brighter, and darker ones remaining dark with some of the light effects 'bleeding' into the dark spots.

GAME OBJECT SYSTEM

This system allows making adding extra objects to the scene very easy and simplifies the implementation of new objects. It features encapsulation and appropriate getter and setter functions for any variables that need to be exposed for object setup or other purposes.

UI AND CONTROLS

The keyboard controls remain the same to the base application. WASD to move, Space to enable mouse look, arrow keys to change camera perspective.

The user interface has drop down menus, due to a bug in ImGui only one section should be expanded as otherwise the other sections options will be changed/highlighted.

Each drop down contains parameters on a specific effect. They should be relatively self-explanatory, apart from these:

- Tessellated Plane
    - Tessellation At Near – tessellation factor at near.
    - Tessellation At Far – tessellation factor at the near amount.
    - Far Distance – How many units away from the camera is ‘far’.
    - Near Distance – How many units away from the camera is considered ‘near’.
    - The tessellation factor will increase linearly between near and far depending on the arguments specified.
- Bloom Post Process
    - Override wireframe – Disable wireframe when drawing the ortho mesh of the scene.
    - Bloom cut off – Minimum brightness for the bloom to start working.
    - Intensity Modifier – How strong the effect will be (Lower values stronger, higher values weaker).
- To turn off any light set its diffuse to black.

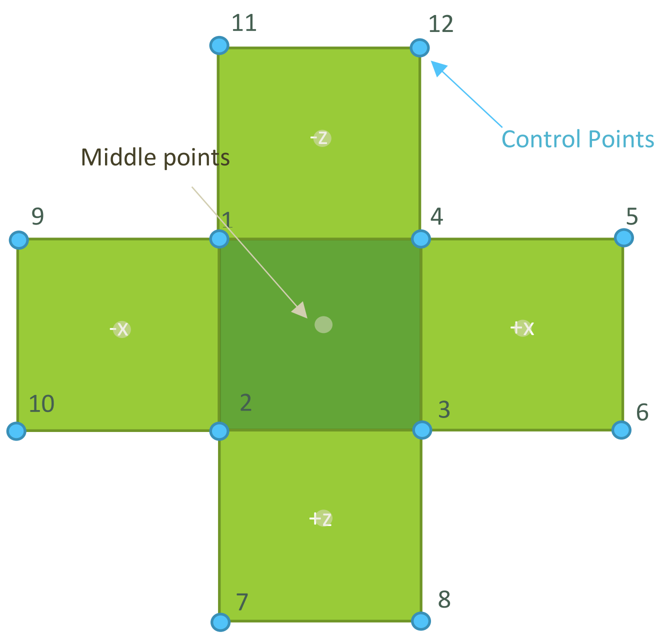
HOW IT WORKS

DYNAMICALLY TESSELLATED TERRAIN

The implemented method was derived from Zink, J., Pettineo, M. and Hoxley, J. (2011). *Practical Rendering and Computation with Direct3D 11*. Natick: CRC Press. With certain adaptations.

The corresponding GameObject for this is TessellatedPlaneObject, and corresponding HLSL files: tessellation_vs, tessellation_hs, tessellationShadows_ds, tessellationShadowDepth_ds, and shadow_ps. The shader class is called ShadowTessellationShader and ShadowTessellationDepthShader.

CONTROL POINTS



Firstly, the control points are generated. There are two extra control points for each side of the patch, which represent the surrounding control points in order to make the terrain watertight. The class responsible for this is TessellatedPlaneMesh.

Then the control points are then passed to the vertex shader, which simply passes the position, normal, and uv's onto the hull shader.

HULL SHADER

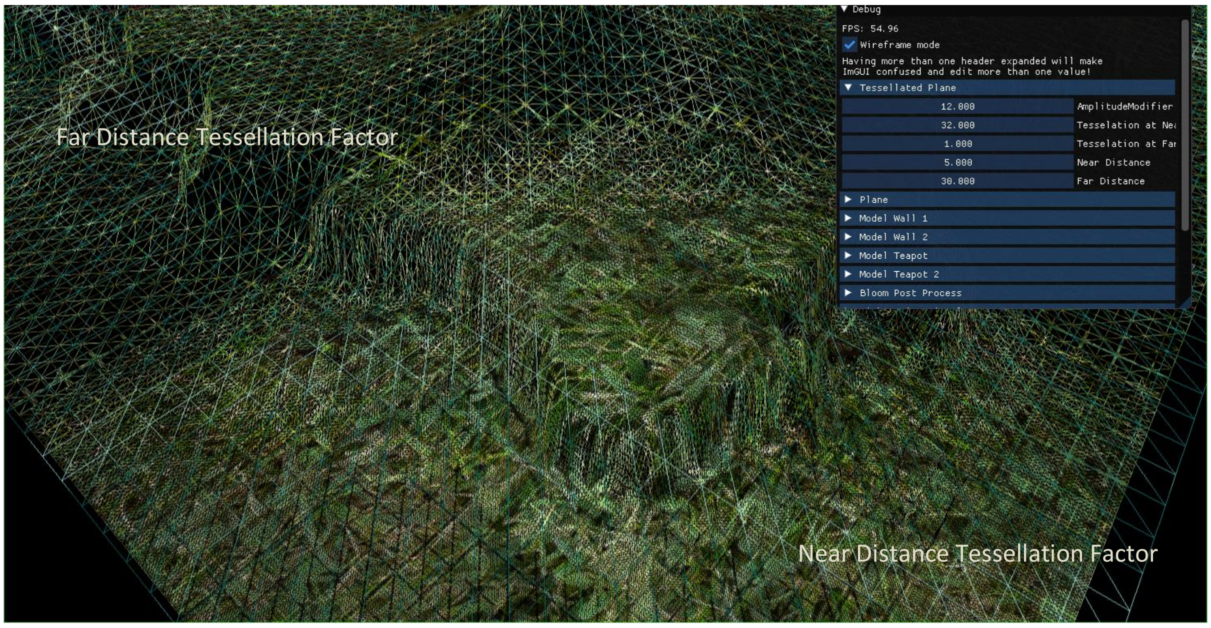
The hull shader then calculates the middle of the four points for the middle patch and surrounding ones, and then the distance between them and the camera. After this the distance between the point and camera is made into a value between 0 and 1, 0 being the 'close' distance, and 1 being the 'far' distance. Then this value is multiped by the close and far tessellation factors.

The calculated tessellation factors of the surrounding patches correspond to the edge tessellation factors of the main patch (marked in dark green). Because the plane is processed in quads, they have two interior tessellation factors, which are both set to the distance between the centre and the camera.

```
/* Get a the tessellation factor for the point based on the camera position. */
float getLODForPoint(float3 p, float3 cameraPos)
{
    float distance = length(cameraPos - p);

        //[Enhancement] TODO: MOVE ALONG EXPONENTIAL/TAN LINE TO MIMIC DISTANCE IN PERSPECTIVE
        float percentBetweenMinMax = 1.f - ((distance - nearDistance) / (farDistance - nearDistance));

        return clamp(lerp(farLOD, nearLOD, percentBetweenMinMax), farLOD, nearLOD); // Make sure that the
tessellation factor doesn't go out of bounds. FarLOD is smaller than nearLOD.
}
...
/* Calculate the mid points of this patch and surrounding ones. */
    float3 midPoints[] = {
        getAverage(inputPatch[0].position, inputPatch[1].position, inputPatch[2].position,
inputPatch[3].position), //Main patch
                   getAverage(inputPatch[2].position, inputPatch[3].position, inputPatch[4].position,
inputPatch[5].position), //+x
                   getAverage(inputPatch[1].position, inputPatch[3].position, inputPatch[6].position,
inputPatch[7].position), //+z
```

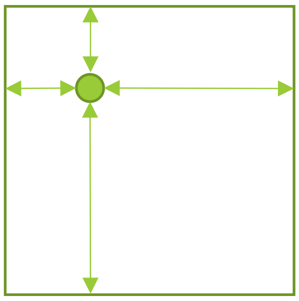
```
                   getAverage(inputPatch[0].position, inputPatch[1].position, inputPatch[8].position,
inputPatch[9].position), //-x
                   getAverage(inputPatch[0].position, inputPatch[2].position, inputPatch[10].position,
inputPatch[11].position) //-z
    };

         /* Determine LOD for patch and surrounding ones. */
    float patchLODs[]= {
        getLODForPoint(midPoints[0], cameraPos), //Main patch
        getLODForPoint(midPoints[1], cameraPos), //+x
        getLODForPoint(midPoints[2], cameraPos), //+z
        getLODForPoint(midPoints[3], cameraPos), //-x
        getLODForPoint(midPoints[4], cameraPos), //-z
    };

    // Set the tessellation factors for the four edges of the quad to match the the LOD's for the four adjacent
ones.
    output.edges[0] = min(patchLODs[0], patchLODs[4]); //top edge
    output.edges[1] = min(patchLODs[0], patchLODs[3]); //left edge
    output.edges[2] = min(patchLODs[0], patchLODs[2]); //bottom edge
    output.edges[3] = min(patchLODs[0], patchLODs[1]); //right edge

    // Set the tessellation factor for tessallating inside the quad.
    output.inside[0] = patchLODs[0]; //0 is the mid point
    output.inside[1] = patchLODs[0];
...
```
Code excerpt from tessellation_hs showing the process for finding the surrounding tessellation factors.

This then gets passed onto the tessellator stage, which tessellates the patches accordingly.


Far Distance Tessellation Factor
Near Distance Tessellation Factor

```
▼ Debug
FPS: 54.96
✓ Wireframe mode
Having more than one header expanded will make
ImGUI confused and edit more than one value!
▼ Tessellated Plane
        12.000      AmplitudeModifier
        32.000      Tesselation at Nea
         1.000      Tesselation at Far
         5.000      Near Distance
        30.000      Far Distance
▶ Plane
▶ Model Wall 1
▶ Model Wall 2
▶ Model Teapot
▶ Model Teapot 2
▶ Bloom Post Process
```

DOMAIN SHADER



Domain Shader Vertex/UV Calculation.

The domain shader then reassembles the points, as the output of the tessellation stage is just a UV coordinate between 0 and 1 from the original patches' edges. The reassembly is done by finding the distance between the edges of the quad - first the left and right and then between top and bottom.

```
    /*Find the position of the new vertex*/
```

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

        /*Find the new postition of the texture with similar technqiue to above*/
    float2 uv1 = lerp(patch[0].tex, patch[1].tex, uvwCoord.y);
    float2 uv2 = lerp(patch[3].tex, patch[2].tex, uvwCoord.y);
    float2 uvPos = lerp(uv1, uv2, uvwCoord.x);
    output.tex = uvPos;
```
Code excerpt from tessellationShadow_ds and tessellationShadowDepth_ds for the lerping process described.

After finding the world position of the vertex, the y position of the vertex is set from sampling the heightmap, and then amplified according to the inputted values. This means that the more detail there is, the more points will be sampled on the heightmap, making the terrain appear more detailed as it is approached.

```
    /*Calculate the position of the new vertex against the world, view, and projection matrices*/
    output.position = mul(float4(vertexPosition, 1.0f), worldMatrix);

        /*Add the Y offset before view and projection matrices*/
    output.position.y = output.position.y + heightMapTex.SampleLevel(samplerState, output.tex, 0.0).r * amplitude;

    output.position = mul(output.position, viewMatrix);
    output.position = mul(output.position, projectionMatrix);
```
Code excerpt from tessellationShadow_ds and tessellationShadowDepth_ds for the matrix multiplicaton.

Then the value is multiplied by the world, view, and projection matrices to get them ready for the rasteriser.

The UV is calculated in an almost identical way – except the matrix multiplication and height transformation is left out as UV's don't need to be transformed after they are found.

NORMALS

The normal of the vertex is calculated using edge detection on the height map. This is because without adding a way to calculate the adjacent triangles (effectively implementing the maths for the tessellator) or proving a normal map it would prove either computationally expensive or require extra textures for the heightmap.

The edge detection algorithm, commonly called Sobel, finds the difference in the averaged colour value (grayscale) in the pixels, on the X and Y axes on the image. This corresponds to X and Z axes in the world, and the filters have been named as such.

This is done by applying a filter on the image.

$$x\ filter = \begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix} \quad z\ filter = \begin{matrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{matrix}$$

The filter is weighted to one side, to which an image sample is applied.

$$image\ sample = \begin{matrix} 0.9 & 0.8 & 0.4 \\ 0.9 & 0.7 & 0.5 \\ 0.8 & 0.4 & 0.4 \end{matrix}$$

The image sample contains the grayscale of the surrounding pixels, and the middle pixel.

$$x\ filter * image\ sample = edge_x = \begin{matrix} -1*0.9 & 0*0.8 & 1*0.4 \\ -2*0.9 & 0*0.7 & 2*0.5 \\ -1*0.8 & 0*0.4 & 1*0.4 \end{matrix} = \begin{matrix} -0.9 & 0 & 0.4 \\ -1.8 & 0 & 1 \\ -0.8 & 0 & 0.4 \end{matrix}$$

To which then one of the filters is applied. 1,1 of the output is equal to 1,1 of the filter * 1,1 of the sample. This is then applied to every frame.

$$\rightarrow\ -0.9 + 0 + 0.4 - 1.8 + 0 + 1 - 0.8 + 0 + 0.4 = \ -2.8$$

Next the set of values is decomposed and each element is added together to find the total difference. In this case the change in X is negative, that means that there is an edge going in the left direction (due to negative values being on the left of the X filer). An identical process is applied to the image sample, except with a z filter. In this case Z refers to Y on the image, as the plane is based on the XZ.

$$edge_x = -2.8$$
$$edge_z = -0.1$$

$$y = \tan^{-1}\left(\frac{edge_x}{edge_z}\right) \approx 1.23$$
$$y = \sqrt{\left(1 - {edge_x}^2 - {edge_z}^2\right)} \approx 1.29$$

Then the Z component is calculated by doing the inverse tan of $edge_x$ over $edge_z$ which yields the angle in radians. In code this is done by using a tan approximation which utilises a square root (last equation listed). This was adapted from Zink, Pettineo and Hoxley (2011) Practical Rendering book, as their method also includes values for weighting the respective x,y,z components to make the normal look accurate.

In summary, the above process effectively generates equivalent of a normal map at runtime.

CODE WISE (TESSELLATIONSHADOW_DS AND TESSELLATIONSHADOWDEPTH_DS)

The code equivalent of this process is:

1. Sample the image.

```
float3x3 inputPixels = {
    sampleHeightmapAt(float2(texel.x - onePx.x, texel.y - onePx.y)), sampleHeightmapAt(float2(texel.x - onePx.x, texel.y)), sampleHeightmapAt(float2(texel.x - onePx.x, texel.y + onePx.y)),
    sampleHeightmapAt(float2(texel.x, texel.y - onePx.y)),           sampleHeightmapAt(float2(texel.x, texel.y)),           sampleHeightmapAt(float2(texel.x + onePx.x, texel.y)),
    sampleHeightmapAt(float2(texel.x + onePx.x, texel.y - onePx.y)), sampleHeightmapAt(float2(texel.x + onePx.x, texel.y)), sampleHeightmapAt(float2(texel.x + onePx.x, texel.y + onePx.y))
};
```

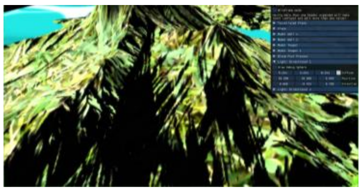
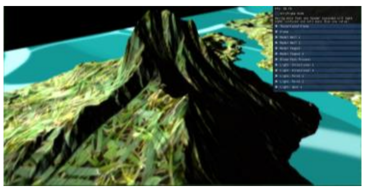
2. Calculate $edge_x$ and $edge_z$.

The calculation is simplified to remove unnecessary multiplication/addition.

```
float xTotal = inputPixels[0][0] - inputPixels[2][0] + 2.0f * inputPixels[0][1] - 2.0f * inputPixels[2][1] + inputPixels[0][2] - inputPixels[2][2];
float yTotal = inputPixels[0][0] + 2.0f * inputPixels[1][0] + inputPixels[2][0] - inputPixels[0][2] - 2.0f * inputPixels[1][2] - inputPixels[2][2];
```

3. Calculate the Y.

This deviates from the mentioned method, as the results of the mentioned method are too strong. Zink, Pettineo and Hoxley (2011) have specified values which when multiplied with the respective components make the normal magnitudes correct.


Original method on left.


```
float z = sqrt(max(0.0f, 1.0f - xTotal * xTotal - yTotal * yTotal));
/*Calculate normalize vetex*/
float xAndYIntensity = 2.f;
float zIntensity = 0.01f;
float3 n = float3(xAndYIntensity * xTotal, zIntensity * z, xAndYIntensity * yTotal);
```

VARIANTS

There are two variations of the domain shader, one for the depth shader (tessellationShadowDepth_ds.hlsl), and the other for the shadow pixel shader (tessellationShadow_ds.hlsl). Their differences are the output types, which are made to match with the inputs for their respective shaders.

The shadowDepth variant simply passes on the calculated position as 'depthPosition' to the shadow depth pixel shader (shadowDepth_ps.hlsl).

Whereas the shadow variant calculates the calculated vertices’ position in the directional light’s height map using each directional light’s view and projection matrices. This is then passed on to the shadow pixel shader (shadow_ps.hlsl). More is mentioned about this in the next section.

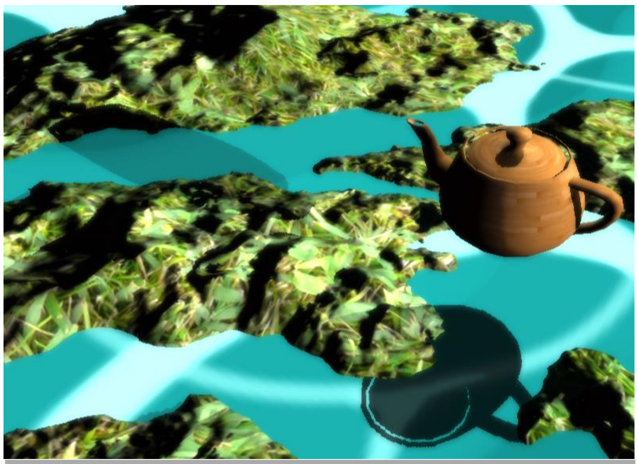
```
struct OutputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
    float4 lightViewPos[NO_OF_SHADOWED_LIGHTS] : TEXCOORD1;
    float4 worldPosition : TEXCOORD10;
};
```

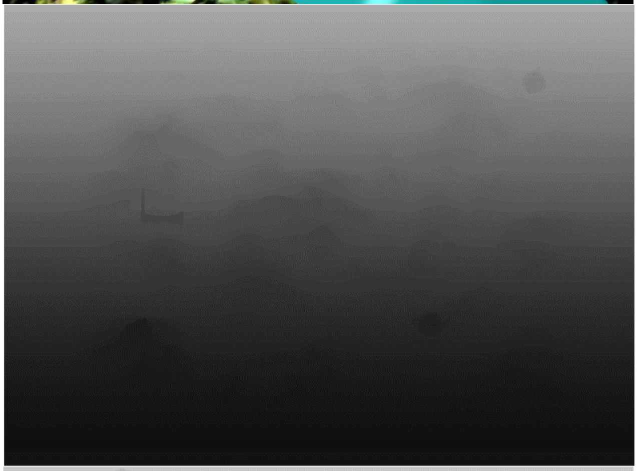
Output type for tessellationShadow_ds and shadow_vs.

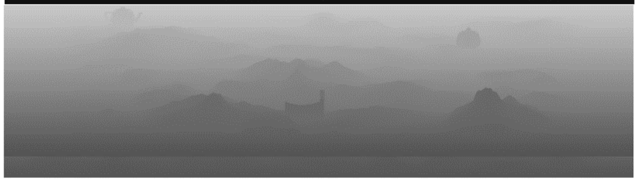
```
struct OutputType
{
    float4 position : SV_POSITION;
    float4 depthPosition : TEXCOORD0;
};
```

Output type for tessellationShadowDepth_ds and shadowDepth_vs.

MULTIPLE SHADOWED DIRECTIONAL LIGHTS







The scene features multiple directional shadowed lights. This requires both a depth and a regular render pass.

DEPTH PASS

Each directional light has a depth pass of its own. This is as each light has a separate 'perspective' that needs the scene to be rendered from. Despite directional lights not technically having a position they are required to have one as the scene has to be rendered from the perspective of the light for the calculation of the shadow.

For this calculation an ortho matrix is used as it renders the whole scene irrespective of the distance and makes it easier to fit the whole scene into one render. The scene is rendered onto a render texture for each light. The way these render textures look like are on the left.

For the depth pass the vertices are multiplied inside of the vertex stage by the world, view, and ortho matrices from the mesh, and then stored in a separate output variable, that is of a different semantic (TEXCOORD0 specifically) as it doesn't get affected by the rasteriser.

In the case of the tessellated plane this happens inside of the domain shader as that where the vertices get reassembled after tessellation. The code is practically identical except the vertices' position relative to the original point also gets calculated.

VERTEX DEPTH SHADER

The vertex shader stage is as described above. The input type is simply the what the mesh provides (position, normal and texels) however the normal and texels are disregarded because textures or normals aren't necessary for calculating the distance inside of the pixel shader. The tessellationShadowDepth_ds is identical, except it also reconstitutes the vertices, the method is mentioned in the last section.

```
struct OutputType
{
        float4 position : SV_POSITION;
        float4 depthPosition : TEXCOORD0;
};
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;
}
```
Excerpt from shadowDepth_vs

PIXEL SHADER

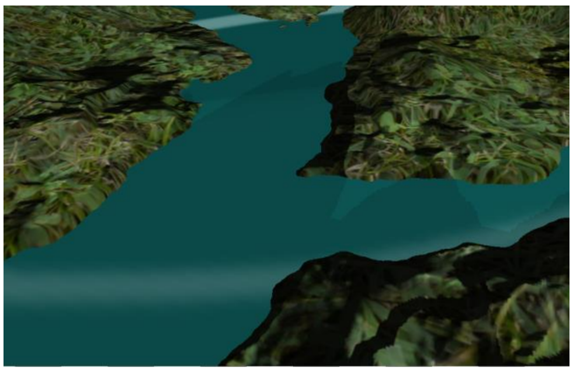
The pixel shader divides the z depth position by the w component, making the vector homogenous, then outputting the depth value into a colour.

This whole process is rendered on to a render texture, which is later used in the 'regular' pass.

Due to the way the render texture is set up (as a depth buffer that gets made visible) the smallest (closest) point gets stored, where as other get disregarded if a smaller value is found in its place.

```
struct InputType
{
	float4 position : SV_POSITION;
	float4 depthPosition : TEXCOORD0;
};

float4 main(InputType input) : SV_TARGET
{
	/* Get the 0-1 depth value of the pixel by dividing the Z pixel depth by the homogeneous W coordinate.*/
	float depthValue = input.depthPosition.z / input.depthPosition.w;
	return float4(depthValue, depthValue, depthValue, 1.0f);
}
```
The shadowDepth_ps.

REGULAR PASS



The regular pass simply renders the object with the shadows. This is done by both transforming the vertices and their normals and texels as normal, but also transforming the specific points by the directional light matrices (view and projection) in order to see where they would be on the depth map.

This results in the output of the vertex shader (or domain shader for the tessellated plane) containing the mesh data, and the points' position on the depth map.

```
#define NO_OF_SHADOWED_LIGHTS 2
...
struct OutputType
{
	float4 position : SV_POSITION;
	float2 tex : TEXCOORD0;
	float3 normal : NORMAL;
	float4 lightViewPos[NO_OF_SHADOWED_LIGHTS] : TEXCOORD1;
	float4 worldPosition : TEXCOORD10;
};
```
The output type for the shadow_vs and tessellationShadow_ds shaders.

Inside of the vertex/domain shader the view and projection matrices from all of the lights are passed in, and then the point is multiplied by them, and then stored as an array in the OutputType for the pixel shader to process. As each array element is treated as another TEXCOORD semantic (Element 1 would be TEXCOORD1, element 2 as TEXCOORD2 etc.), the world position has to be moved forward a few semantic values inside the output type. This does impose a limit on the maximum amount of 9, unless the worldPosition semantic was updated across shadow_vs, tessellationShadow_ds, and shadow_ps. However, this value was decided on as the performance would be further made detrimental. More on this subject is mentioned in the improvements section.

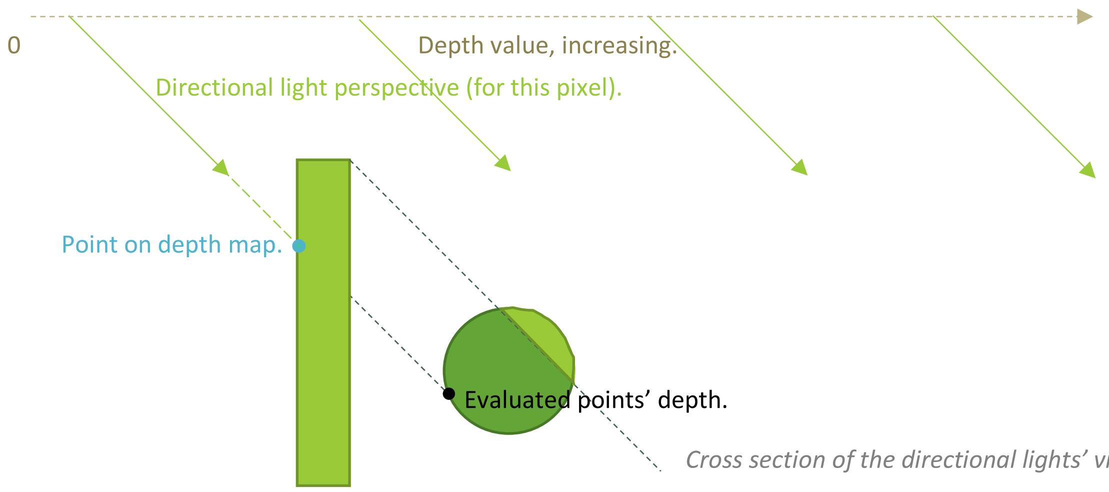
```
	for (int i = 0; i < NO_OF_SHADOWED_LIGHTS; i++)
	{
		/* Calculate the position of the point from the light point. */
```

```
                    output.lightViewPos[i] = mul(input.position, worldMatrix);
                    output.lightViewPos[i] = mul(output.lightViewPos[i], lightViewMatrix[i]);
                    output.lightViewPos[i] = mul(output.lightViewPos[i], lightProjectionMatrix[i]);
         }
```
Calculation of the point relative to the view matrices, for each light.

PIXEL SHADER

The pixel shader where the comparison happens. It takes in a depth map for each of the directional lights and finds the position on the relevant depth map. Then samples the pixel and retrieves the value from it at that point. If the depth value of the point is less than on the map then the pixel is lit.

This is because if the value on the map is more than the calculated points' distance, it indicates that the sampled point is behind something. Whereas if the sampled point is less than the heuristic indicates that the point is in front.



Cross section of the directional lights' view.

This diagram shows how the depth value is calculated. From the perspective of the light the value in the depth map is less than the value calculated, therefore it must be in shadow and is not lit by that directional light.

```
...
float depthValue = depthMapTexture[i].Sample(shadowSampler, pTexCoord).r;
float lightDepthValue = input.lightViewPos[i].z / input.lightViewPos[i].w; // Make the value homogenous to match
the scale on the depth map.
...
if (lightDepthValue < depthValue)
{
         /*Its lit!*/
...
```
Section in shadow_ps responsible for deciding if a value is lit or not.

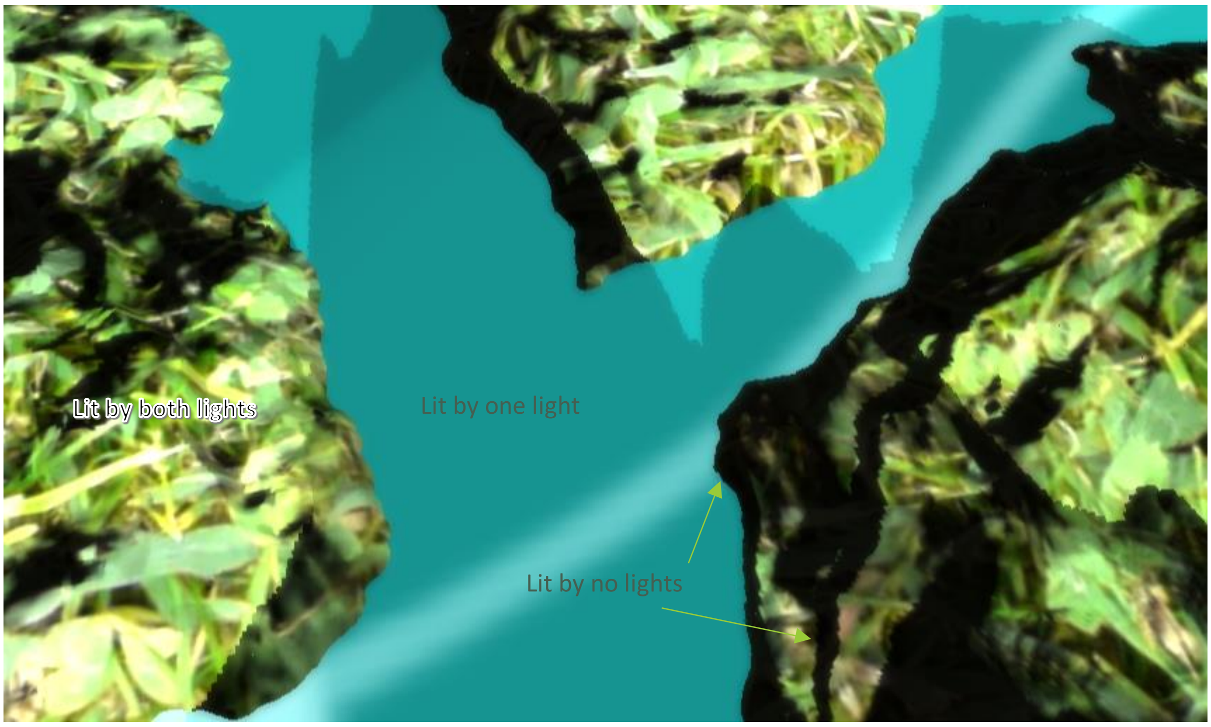
DIRECTIONAL LIGHT

The lightness of the specific section is determined by a regular directional light calculation, which determines how much light is added to the specific pixel by doing the dot product of the normal and light direction (both unit vectors).

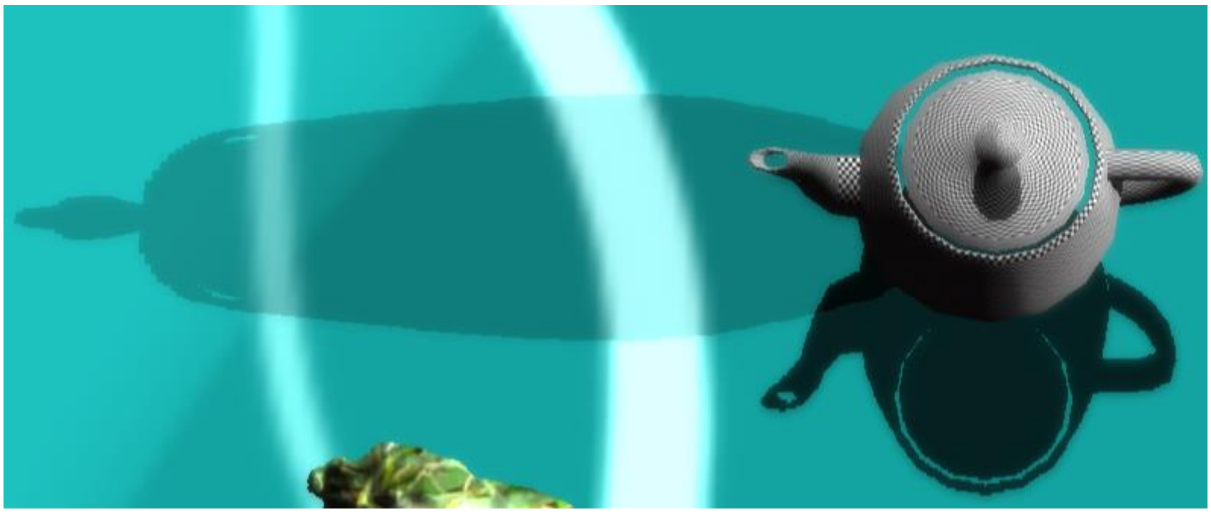
```
/// Calculate directional lighting based on the normal.
float4 calculateLightingDirectional(float3 lightDirection, float3 normal, float4 diffuse)
{
         float intensity = saturate(dot(normal, lightDirection));
         float4 colour = saturate(diffuse * intensity);
         return colour;
}
```
Directional Lighting code in shadow_ps.

Given that there are multiple directional lights in the scene, each pixel needs two sets of calculations, given that it is lit from both sides. If the pixel is only lit by one light (like only one part is in shadow) the only calculation that takes place is for the light that is lighting it. As a result, the partly lit sections appear comparatively bright to any ones that aren't.

If the light is lit by more than one directional light the effect is reduced, as otherwise the scene would be too bright – especially with bloom enabled.



Another effect to note is that because there are two directional with different directions (one pointing down and in negative x at 45 degrees, and the other pointing down and backward in negative z) one of the shadows appears to be darker than the other. This is to do with the angle of incidence where its calculated.



After the light is determined that the pixel is lit the colour value is stored in an array and whether the pixel was lit in another. The values from the array are then added to the general colour and the amount of shadowed lights are stored in a separate variable. After adding the colours together, they are divided by the total amount of lights that light that pixel in order to curb the overall light intensity, and then finally added to the output colour.

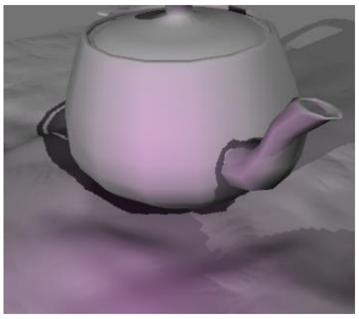
```
	/* Add the shadowed lights to the pixel. */
	int noOfShadowedLights = 0;
	float4 totalShadowedColour = float4(0.f, 0.f, 0.f, 1.f);
	for (int i = 0; i < NO_OF_SHADOWED_LIGHTS; i++)
	{
		if (isShadowed[i])
			noOfShadowedLights++;

		totalShadowedColour += shadowedColours[i];
	}

	if (noOfShadowedLights > 0) //decrease the strengh of the directional light if there is more than 1.
		totalShadowedColour /= noOfShadowedLights;

	colour += totalShadowedColour;
```
Code in shadow_ps corresponding to adding the colour together.

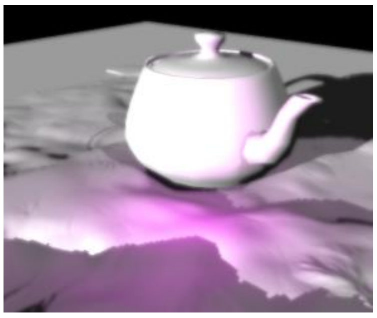
MULTIPLE POINT AND SPOT LIGHTS

There are two spot, and two point lights scattered across the scene. Their position can be marked by a 'debug sphere'. Please note that the debug sphere has no shadows or lighting performed on it because it is not intended to be part of the scene.

The multiple point lights have two places where their calculation takes place - inside the vertex (or domain) and pixel shader. The vertex/domain shader is responsible for passing the world position onto the pixel shader, and the pixel shader for calculating the intensity of the diffuse colour on light direction in relation to the normal.

POINT LIGHTS





The point and light require the world position of the pixel to be passed to it in order to calculate the angle of incidence between them and a certain pixel. The world position calculation is done inside of the vertex/domain shader stage, which is simply the vertex point multiplied by the world matrix. This is stored in a TEXCOORD semantic value in order for the rasterizer to not modify the value.

The effect of the point light is far more pronounced on darker areas, as the combined directional lights leave little room for extra colour information, as well as the bloom effect, which may oversaturate the area to where very little colour (i.e. non white) is left. On the left is the scene with more appropriate directional lighting, whereas on the left the directional lighting was turned up and bloom was enabled.

```
struct OutputType
{
...
	float4 worldPosition : TEXCOORD10;
};
...
output.worldPosition = mul(input.position, worldMatrix);
...
```

Code inside shadow_vs and tessellationShadow_ds responsible for calculating the world position of the pixel, and the semantic on the output type.

```
float4 calculateLightingPoint(...)
{
float3 pxToLightVec = worldPosition.xyz - lightPos; // Get vector between pos and light.
	float distance = length(pxToLightVec); // Get the distance between the two.
	pxToLightVec = normalize(pxToLightVec); // Now make it unit after getting length.

	float normalIntensity = saturate(dot(normal, -pxToLightVec));
	float4 lightColour = saturate(normalIntensity * diffuse); // Add the intensity to the diffuse value.

	lightColour /= attenuationConstant + (attenuationLinear * distance) + (attenuationExp * (distance *
distance)); // Divide the output colour at the pixel by the attenuation factor.
…
```

Code inside shadow_ps responsible for calculating the point.

Both due to simplicity and performance, the code for the point and directional lights was put into the shadow_ps, as it has to take place after the shadow calculations but also is easier than doing an extra pass and blending the colours back into the scene.

SPOT LIGHT

The spot light works in a similar fashion except it only projects around the centre, in this case the light is set to only project 90 degrees around its self. It checks whether the light to pixel vector is between the certain angle using the cosine rule (both vectors are made unit, and the dot product is taken, which is equal to the cosine of the angle – and as the angle is stored in cosine form it doesn't need to be inverted). If the vector is within the angle it can be lit.

```
float4 calculateLightingSpot(...)
{
	float halfAngle = cos(0.785f / 2.f); //0.785 radians to cos which is 45 
degrees.
	...
	float cosOfAngleBetweenPxAndLight = dot(lightDirection, -pxToLightVec);

	if (cosOfAngleBetweenPxAndLight > halfAngle) //check if the normals
	{
		Its lit!
	}
...
```

Excerpt of code from shadow_ps showing key differences in a spot light.

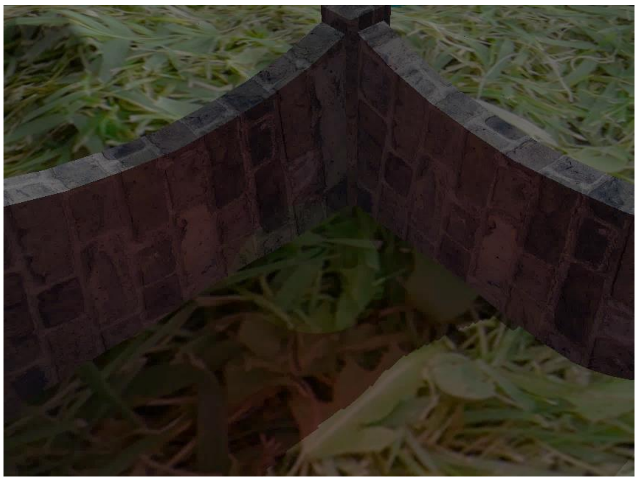
MULTIPLE LIGHTS

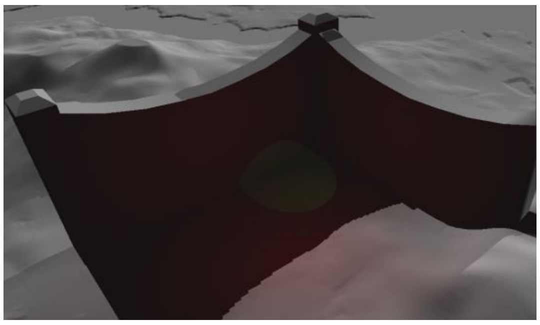
Multiple spot/point lights are achieved by performing each of the light calculations for each pixel, and then adding their colour return values to the overall output colour of the pixel.

```
	/*Add point light colour.*/
	float4 pointColor =float4(0.f, 0.f, 0.f, 1.f);
	for (int i = 0; i < NO_OF_POINT_LIGHTS; i++)
	{
		pointColor += calculateLightingPoint(input.normal, positionPoint[i], diffusePoint[i], 
input.worldPosition.xyz, 1.f, 2.f, 1.f);
	}

	colour += pointColor;

	/*Add spot light colour.*/
	float4 spotColour = float4(0.f, 0.f, 0.f, 1.f);
	for (int i = 0; i < NO_OF_SPOT_LIGHTS; i++)
	{
		spotColour += calculateLightingSpot(-directionSpot[i].xyz, positionSpot[i].xyz, input.normal, 
diffuseSpot[i], input.worldPosition, 2.f, 4.f, 1.f);
	}
	colour += spotColour;
```

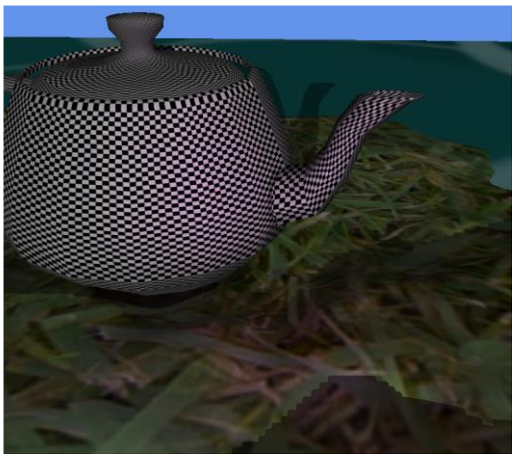
Excerpt of code from shadow_ps showing how lights are added.

Below are some examples of multiple lights, and in the scene. One is without a texture or directional light as its easier to see the effect it has. Some of the images have the directional light 'turned off' (setting diffuse to black as the lights work by adding the light diffuse to the pixel) as it makes it easier to see.

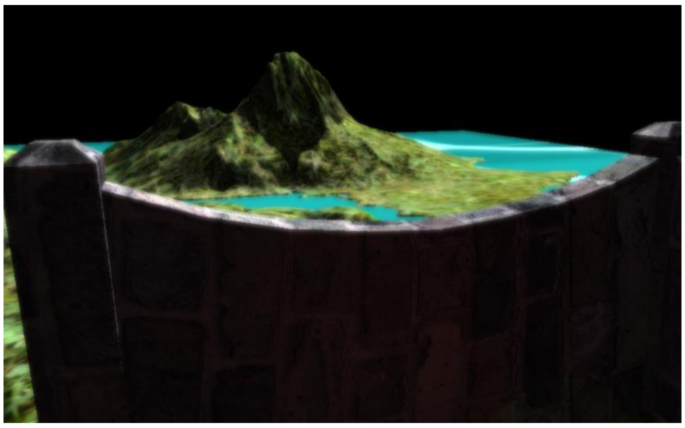




Other incidences of lights in the scene:



BLOOM



The bloom post process effect exaggerates highly lit areas and causes a haze effect that slightly 'bleeds' into the darker areas.

It does this by rendering the scene to a texture, cutting the darkest parts out ('Bloom Cut Off' setting in GUI), then blurring the scene horizontally and vertically, and finally merging back into the render texture and rendering the it onto the screen size ortho mesh. Each of these stages has its own render texture, except from the last one which reuses the original render texture scene.

Each render texture stage has a screen size ortho mesh passed in, as some values are required for the vertex shader to pass onto the pixel shader.

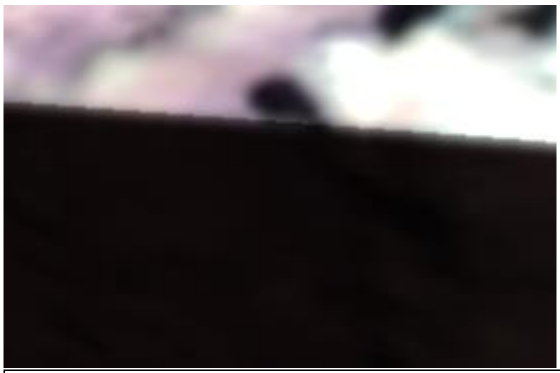
The bright part cut off is done through getting the length of the colour, which can be done as it can be reinterpreted as a vector. This means that the cut off is done in grayscale, as the overall colour strength is taken.

```
float4 main(InputType input) : SV_TARGET
{
	/*Sample texture*/
    float4 textureColour = sceneTex.Sample(sampler0, input.tex);

	/*Get overall intensity*/
    float overallIntensity = length(textureColour.xyz);

	/*Cutoff above a certain intensity*/
    if (overallIntensity < intensityCutOff)
        return float4(0.f, 0.f, 0.f, 1.f);

    return float4(textureColour.xyz, 0.f);
}
```
Excerpt of bloomGetBrightest_ps showing the cutoff logic.

During the blur passes the alpha is changed to the overall intensity of the pixel to make sure that its blended well into the scene. Otherwise if the alpha is set to 1 or set to a constant value the blended scene doesn't blend in as smoothly as using this technique.

```
	/* Sample around texture using the weights .*/
    for (int i = -(noOfWeights - 1); i < noOfWeights; ++i)
    {
        color += sceneTex.Sample(sampler0, float2(input.tex.x, input.tex.y + (onePx.y * ((float) i)))) *
weights[abs(i)];
    }
	color.w = saturate(length(color.xyz) / 3.f); // Scale the alpha based on the color strength (more needs
to be blended in the more color there is).

    color.xyz = saturate(color.xyz / intensityModifier);
```
Excerpt from bloomBlurVertical_ps and bloomBlurHorizontal_ps with the calculation of alpha values.

The blur pass also uses custom distribution values, in this case to sample 24 pixels around the pixel that will be returned. A script to generate values was made by me (link) in Ruby. The script generates values on a normal distribution curve, using the Gaussian equation, which is based on the deviation. The number of pixels outputted is affected by the 'stdev' variable, which represents the standard deviation. The smaller the value the larger the number of pixels.

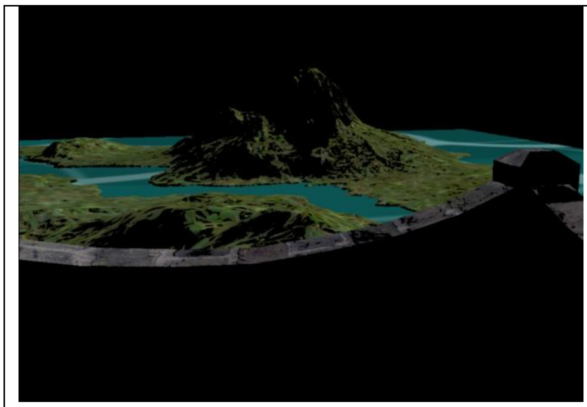
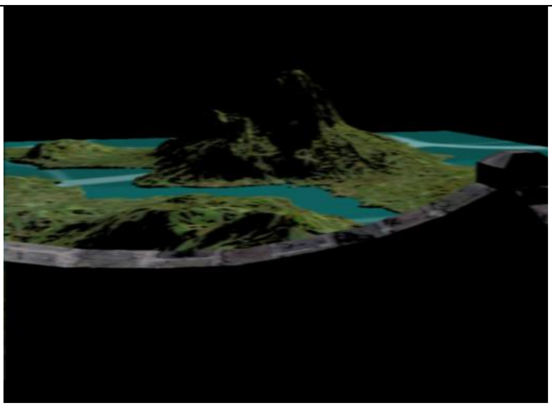
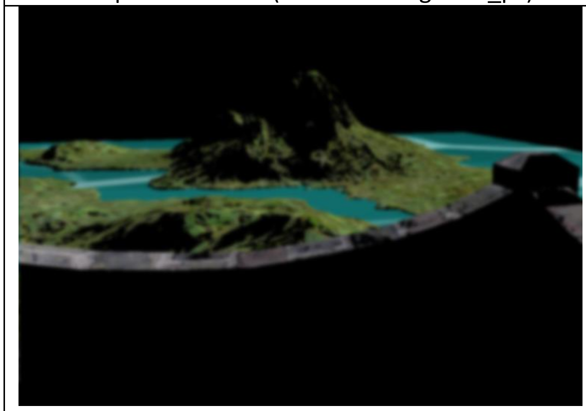
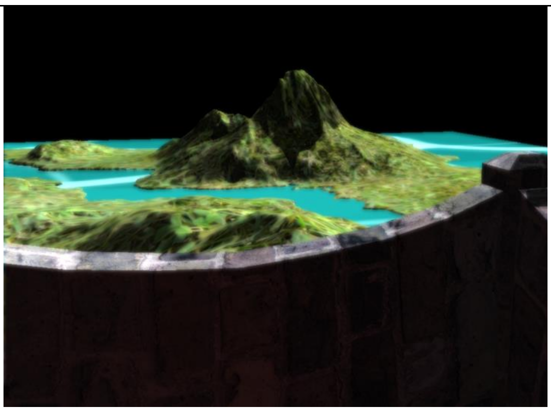


Finally, the two scenes (one regular, one blurred) are merged, which makes the overall scene also appear brighter (due to effectively adding the colour of two scenes on top of each other). In addition, a bit of the lighter parts blur into the darker parts near the edge/where the cut off happens. The aforementioned effect is pictured to the left.

```
textureColour = saturate(textureToMerge.Sample(sampler0, input.tex) + sceneTexture.Sample(sampler0, input.tex));
```

Code excerpt from bloomMergeToScene_ps, showing how the merging takes place (additive).

VISUAL SUMMARY

 1. Render scene to texture and get brightest parts of scene (bloomGetBrightest_ps).	 2. Blur vertically (bloomBlurVertical_ps).
 3. Blur horizontally (bloomBlurHorizontal_ps).	 4. Merge the blurred values back to scene (bloomMergeToScene_ps).

GAME OBJECT SYSTEM

The scene is divided into GameObjects. All items in the scene, and even rendering stages and effects (shadows and bloom) have their own object as it encapsulates their functionality neatly, as well as makes them easy to add to the existing program.

SCENE

The scene is a ‘wrapper’ around App1.cpp – it separates from some extras that the App1 class contains such as calling render, and update, updating the camera and setting up the window. This makes the code in the Scene class fully relevant to the scene.

All of the GameObjects in the scene are initialised inside the constructor. They are assigned any parameters that need to be changed, such as position and texture.

The scene also contains the GameObjectManager, which manages the lifespan, initialisation and evocation of the update(), gui(), render(), renderDepth(..), renderWithShadows(..) functions on the GameObject.

Inside of the render function different draw calls are made depending on whether the bloom post process is enabled – either render to render texture then to ortho, or just render straight to backbuffer.

GAME OBJECTS AND GAME OBJECT MANAGER

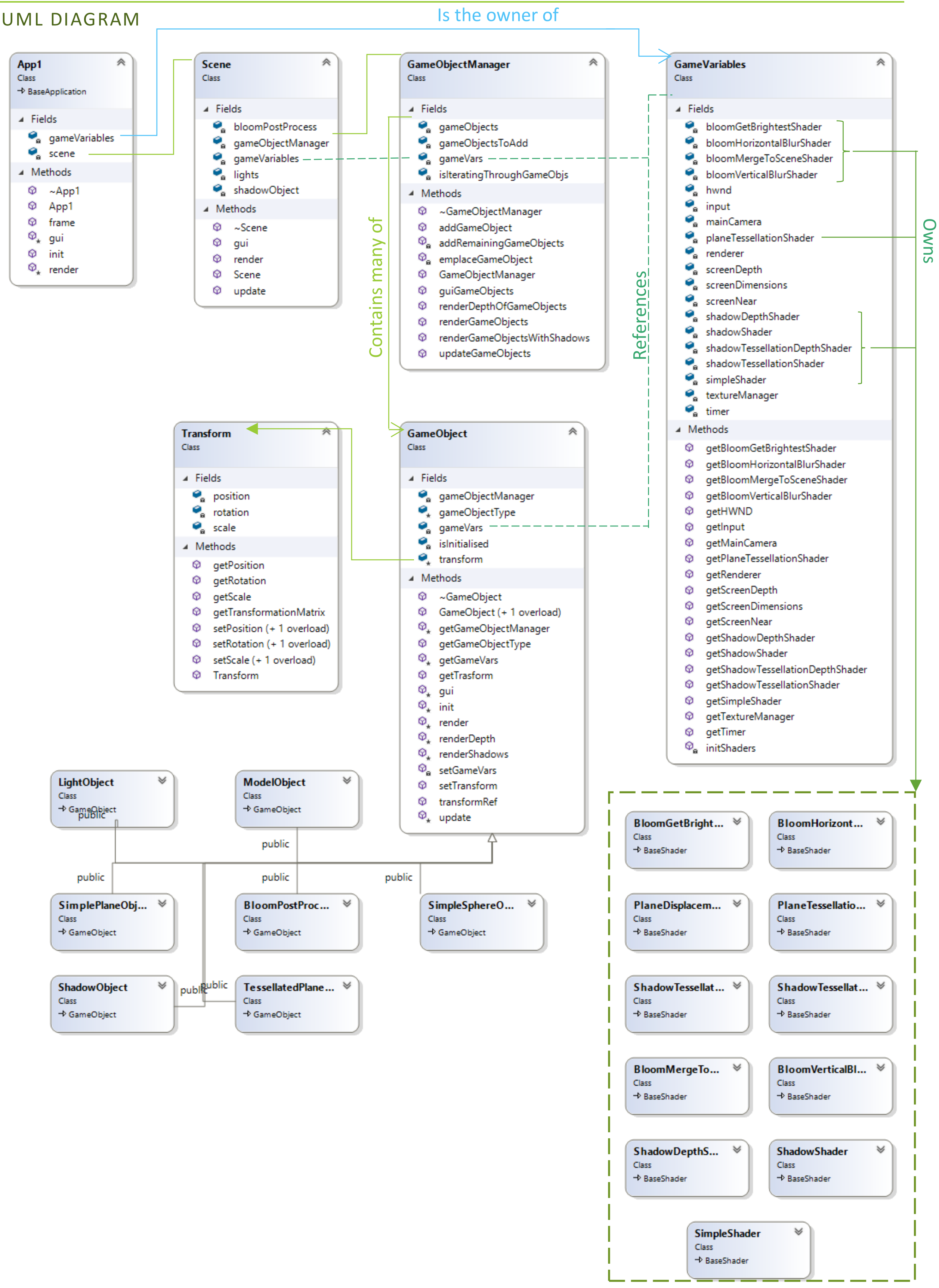
A GameObject is designed to encapsulate all of the relevant functions of an object. As a result, there are multiple overridable functions. The GameObject is controlled by the GameObjectManager which is a friend class to make sure that functions such as init() are not called before the object has been initialised.

The GameObjectManager contains a vector of shared_ptrs to make adding new objects easier and require less syntax than a unique_ptr.

The GameObject also contains a Transform object which aims to make the calculation of the world matrix easier. It involves setting the rotation/transform/scale for the GameObject, and during any shader calls that take the world matrix, multiplying it by the renderer’s world matrix during the setShaderParams() call.

In addition, there is a GameVariables class inside the GameObject which allows access to any necessary variables such as the renderer, shaders, main camera, and screen dimensions. This makes the GameObject class substantially neater as it only requires one reference to the GameVariables class. The GameVariables class also is the owner of the shader objects, as it makes them easy to share across multiple objects.

UML DIAGRAM


Is the owner of
Contains many of
References
Owns
App1
Class
BaseApplication
Fields
gameVariables
scene
Methods
~App1
App1
frame
gui
init
render
Scene
Class
Fields
bloomPostProcess
gameObjectManager
gameVariables
lights
shadowObject
Methods
~Scene
gui
render
Scene
update
GameObjectManager
Class
Fields
gameObjects
gameObjectsToAdd
gameVars
isIteratingThroughGameObjs
Methods
~GameObjectManager
addGameObject
addRemainingGameObjects
emplaceGameObject
GameObjectManager
guiGameObjects
renderDepthOfGameObjects
renderGameObjects
renderGameObjectsWithShadows
updateGameObjects
GameVariables
Class
Fields
bloomGetBrightestShader
bloomHorizontalBlurShader
bloomMergeToSceneShader
bloomVerticalBlurShader
hwnd
input
mainCamera
planeTessellationShader
renderer
screenDepth
screenDimensions
screenNear
shadowDepthShader
shadowShader
shadowTessellationDepthShader
shadowTessellationShader
simpleShader
textureManager
timer
Methods
getBloomGetBrightestShader
getBloomHorizontalBlurShader
getBloomMergeToSceneShader
getBloomVerticalBlurShader
getHWND
getInput
getMainCamera
getPlaneTessellationShader
getRenderer
getScreenDepth
getScreenDimensions
getScreenNear
getShadowDepthShader
getShadowShader
getShadowTessellationDepthShader
getShadowTessellationShader
getSimpleShader
getTextureManager
getTimer
initShaders
Transform
Class
Fields
position
rotation
scale
Methods
getPosition
getRotation
getScale
getTransformationMatrix
setPosition (+ 1 overload)
setRotation (+ 1 overload)
setScale (+ 1 overload)
Transform
GameObject
Class
Fields
gameObjectManager
gameObjectType
gameVars
isInitialised
transform
Methods
~GameObject
GameObject (+ 1 overload)
getGameObjectManager
getGameObjectType
getGameVars
getTrasform
gui
init
render
renderDepth
renderShadows
setGameVars
setTransform
transformRef
update
LightObject
Class
GameObject
ModelObject
Class
GameObject
public
public
public
public
public
public
SimplePlaneObj...
Class
GameObject
BloomPostProc...
Class
GameObject
SimpleSphereO...
Class
GameObject
ShadowObject
Class
GameObject
TessellatedPlane...
Class
GameObject
BloomGetBright...
Class
BaseShader
BloomHorizont...
Class
BaseShader
PlaneDisplacem...
Class
BaseShader
PlaneTessellatio...
Class
BaseShader
ShadowTessellat...
Class
BaseShader
ShadowTessellat...
Class
BaseShader
BloomMergeTo...
Class
BaseShader
BloomVerticalBl...
Class
BaseShader
ShadowDepthS...
Class
BaseShader
ShadowShader
Class
BaseShader
SimpleShader
Class
BaseShader

IMPROVEMENTS

I am quite happy with the effects that I have achieved, although I wish I had spent more time adding extra features, such as the cube fountain (geometry shader) mentioned in my milestone and proposal. In addition, I wish I had done a depth-based water shader that not only displaced the water but also made a 'depth haze' when looking inside – so the deeper the water is, making the water less transparent the further you go down.

Almost everything from the coursework milestone (revised proposal) was implemented except water and geometry shader. Overall, if I had started the coursework earlier and worked faster (worrying less about the initial architecture – although I have improved on that aspect since last year), and as a result mostly likely I would've gotten the geometry shader done, and possibly the water shader. Moreover, the Perlin noise terrain would've been an excellent learning experience.

Due to generous time allocation on the revised (milestone) proposal I was able to meet the deadline on time. Admittedly the original proposals' timescale was very ambitious. Although I dropped the specular from the implementation as each object would've needed a way to tell what one needs it, at what level. This is in addition to making sure that the specular to reacts with shadows correctly.

I'd like to improve the final texture used on the plane – It would've ideally been generated based on the height of the terrain and blended in the relevant textures for the height. This would make the scene much more visually appealing. And make the lights easier to see!

While the edge detection method is great for finding normals, an actual normal displacement map would've been ideal, as it could've created a surface roughness, giving the illusion of more vertices, which in conjunction with the tessellation would give an even more visually appealing effect.

I'd also like to have added shadows to the point and spot lights, which wouldn't require many changes to the current way lights are done, mostly passing in extra values, shadow maps, and modifying the shadow calculation slightly to take the directional light be taken into account. In addition to pushing back the semantic further to allow more lights. I would've also liked to implement soft shadows, which would've required rendering them to a texture then blurring them, then rendering them back into the final scene.

I also would like to improve the efficiency of the pixel shader by removing some arrays and loops, as it would've sped up the process overall. The depth map values would also be better as a texture array instead of using a hardcoded array. This is in addition to every pixel having its relation to each light calculated, which means that each pixel would have an extra calculation if another light was added. A solution is deferred rendering, which would store which item is lit by which light, and only perform lighting calculations on that. However, implementing deferred rendering would be a substantial task requiring many changes to the fundamental way the coursework works – it would probably be a whole coursework within its self.

The GameObject system helped speed up the implementation and improve code cleanliness, I would've added a component system, as some code repeats throughout the GameObjects, such as the shader calls, where the fundamental differences being the matrix values and meshes passed in. Even though this system has memory locality issues there are not that many objects in the scene so the effect would have been negligible – however this could be resolved by placing each of the components close in memory along with the values they have to modify on the GameObject. There are some redundant things like an object tagging system and a shadow-free rendering function, and runtime GameObject addition support, but otherwise most of the features are used.

Lastly the ShaderHelpers class was very helpful in setting up buffers far less error prone and substantially neater.

REFERENCES

Chapter 9 of Zink, J., Pettineo, M. and Hoxley, J. (2011). *Practical Rendering and Computation with Direct3D 11*. Natick: CRC Press.

- Overview of dynamically tessellated terrain and using Sobel to calculate normal edge detection.

Chapter 21 of Fernando and Fernando, Randima (2004) GPU gems : programming techniques, tips, and tricks for real-time graphics. London: Addison-Wesley.

- General overview of how to make a glow shader.

Braynzarsoft.net. (2019). *DirectX 11 - Braynzar Soft Tutorials [Collection]*. [online] Available at: https://www.braynzarsoft.net/viewtutorial/q16390-braynzar-soft-directx-11-tutorials [Accessed Oct. 2019].

- Explanation of how point lights and spot lights works, as well as attenuation.

MODELS, TEXTURES

Texturelib.com. *Green Mixed Grass Texture*. [online] Available at: http://texturelib.com/texture/?path=/Textures/grass/grass/grass_grass_0101 [Accessed Nov. 2019].

- Grass texture used, downsized and tiled from the original.

Deviantart.com. *Seamless Cartoon-styled Water Texture*. [online] Available at: https://www.deviantart.com/berserkitty/art/Seamless-Cartoon-styled-Water-Texture-743787929 [Accessed Dec. 2019].

- Water texture.

OpenGameArt.org. *Wall*. Available at: https://opengameart.org/content/wall [Accessed Dec. 2019].

- The two walls seen, modified from original to have the side patched in.

Unreal Engine Forums. *Duke's Free Heightmaps*. [online] Available at: https://forums.unrealengine.com/development-discussion/modding/ark-survival-evolved/92497-duke-s-free-heightmaps [Accessed Dec. 2019].

- Heightmap texture. Scaled down.

Any remaining textures or models were not made by me and were provided to me with the coursework.

RUBY SCRIPT SOURCE

Published on Github Gist. Created by me (my Github account is Giodestone).

https://gist.github.com/giodestone/f1e04262bf467436ff745fd180ec12d1

```
stdev = 0.35 #change me to change the amount of pixels and the curve (try gaussian equation on
desmos)

arr = Array.new

step = 1.0
y = 0.0
x = 0.0
```

```
loop do
        y = (1 / Math.sqrt(2 * 3.14159 * (stdev ** 2))) * (2.718 ** -(x ** 2 / 2 * (stdev ** 2)))
        arr << y;
        x = x + step
        puts "x: #{x}, y: #{y}"
        break if y <= 0.0001
end

for i in 0..arr.length - 1
puts "#{arr[i]},"
end

puts "#{arr.length}"
```

SPECIAL THANKS

Paul Robertson and Erin Hughes for teaching and assistance during this project and module.