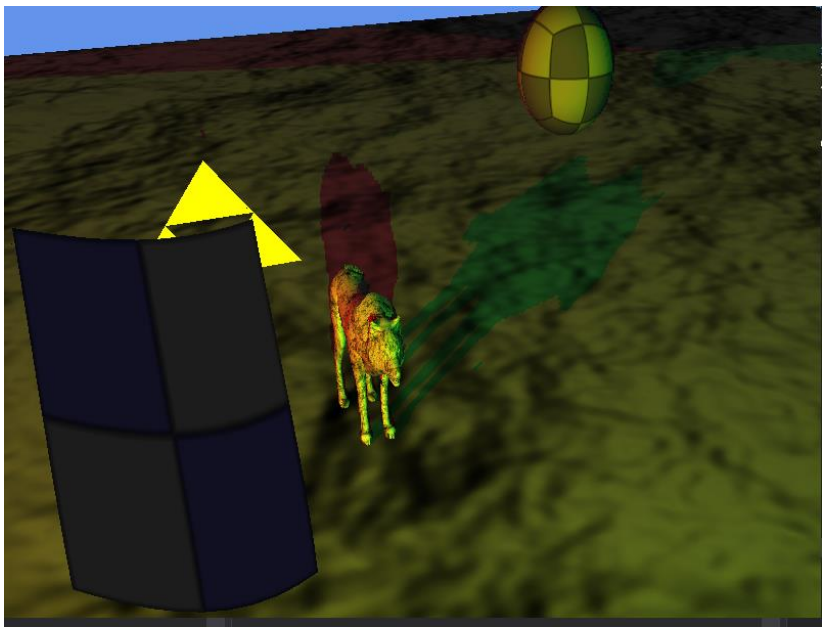
1300869 Neil Donald 3rd Year CGT
AG0904A15 Graphics Programming with Shaders
Coursework submission 2015

This document will explain the processes and techniques used in the application, giving details on the specific shaders used for each aspect of the scene created. This will also describe possible amendments to the code that would solve any problems that occurred in implementation.



User controls for this application are as follows:

Camera controls use the “arrow” keys for camera rotation with the “up” key for looking up, “down” for looking down, “right” for right and “left” for left.
The “QWEASD” keys are used for translating the camera position. “Q” and “E” are for moving the camera down and up, “W” and “S” move forward and back and “A” and “D” move left and right.
One of the planes in the scene is being affected by tessellation and vertex manipulation simultaneously. Tessellation itself is being increased with the “X” key and decreased with the “Z” key. The vertex manipulation has been implemented to create a rippling effect on the plane. The “H” and “N” keys increase and decrease the height of the manipulation respectively, which affects the distance the plane travels forward and backward along the x-axis. The “J” and “M” keys affect the frequency along the z-axis which creates the ripple of the plane as one corner is manipulated at a different rate from the others.
The final user input is the “Space” bar switches the Wireframe mode of the scene on and off, allows the user to see the tessellation of the specific plane more clearly.

Application Implementation

The scene has had shadows implemented for the model and sphere objects as well as vertices of the geometry shape and the tessellated plane meshes, making use of specific pixel and vertex shaders designed to calculate the depth of the scene from the camera and the shadows themselves. As there are two lights in the scene, processing the depth had to be calculated twice for each light with a separate “RendertoTexture” function as well as a separate render textures. The depth data relating to each individual lights were then used together in the shadow shader called in the last “RenderScene” function to produce the mix of shadows for the scene. For simplicity during the explanation of the effects of the depth and shadow shaders, the sphere mesh will be used as an example. Each of the stages discussed are applied to every mesh in the scene alongside any additional shader calls such as calls to the geometry shaders

From the start in the main.cpp, each mesh, specular light and shader class was initialised as well as the two render textures to be used.
The Render Texture is set as the render target for this light’s effect on the scene, with any previous data in the target being cleared.

```
m_RenderTexture->SetRenderTarget(m_Direct3D->GetDeviceContext());
m_RenderTexture->ClearRenderTarget(m_Direct3D->GetDeviceContext(), 0.39f, 0.58f, 0.92f, 1.0f);
```

Then the depth values for the first light are calculated in and assigned to a render texture. The world, view and projection matrices for the camera and the light are initialized. The depth shader class then takes in the world matrix from the camera and the view and projection matrices from the light, sets them into buffers for the lights colour values and positioning and then passes them to the depth vertex shader .hlsl file. The world matrix from the camera is modified when processing some meshes at each function so as translate and scale that specific mesh in the world. This modified value is used by the shaders and then reset afterwords for the next mesh

```
worldMatrix *= XMMatrixTranslation(13, 2, 13);
worldMatrix *= XMMatrixScaling(3, 3, 3);
m_SphereMesh->SendData(m_Direct3D->GetDeviceContext());
m_DepthShader->SetShaderParameters(m_Direct3D->GetDeviceContext(), worldMatrix, lightViewMatrix,
lightProjectionMatrix);
m_DepthShader->Render(m_Direct3D->GetDeviceContext(), m_SphereMesh->GetIndexCount());
m_Direct3D->GetWorldMatrix(worldMatrix);
```

The vertex shader .hlsl calculates the specific objects position against each of the passed in matrices, stores this value as its depth position and then send the value down the line to the depth pixel shader .hlsl.

```
output.position = mul(input.position, worldMatrix);
output.position = mul(output.position, viewMatrix);
output.position = mul(output.position, projectionMatrix);
output.depthPosition = output.position;
return output;
```

Once in the pixel shader .hlsl the depth value's z component will be divided by the homogeneous w coordinate. The depth values x,y and new z components are then applied to create a grey scale colour that represents the distance of the shapes from the light or whether the vertices can be seen by the light.

```
depthValue = input.depthPosition.z / input.depthPosition.w;
color = float4(depthValue, depthValue, depthValue, 1.0f);
return color;
```

This process is then repeated in the next "RenderToTexture2" function for the next light, passing that lights view and projection matices and outputting another greyscale map of the scene from the second light's perspective.

After these stages have been calculated and the greyscale depth maps are created and assigned to the appropriate Render Textures, the shadows are then calculated and the scene rendered.
The "RenderScene" function takes the world, view and projection matrices of the camera and both lights of the scene and sends to the shadow shader class.

```
worldMatrix *= XMMatrixTranslation(13, 2, 13);
worldMatrix *= XMMatrixScaling(3, 3, 3);
m_SphereMesh->SendData(m_Direct3D->GetDeviceContext());

m_ShadowShader->SetShaderParameters(m_Direct3D->GetDeviceContext(), worldMatrix, viewMatrix,
projectionMatrix, lightViewMatrix, lightProjectionMatrix, m_SphereMesh->GetTexture(), m_RenderTexture-
>GetShaderResourceView(), m_Light, lightViewMatrix2, lightProjectionMatrix2, m_RenderTexture2-
>GetShaderResourceView(), m_Light2);
m_ShadowShader->Render(m_Direct3D->GetDeviceContext(), m_SphereMesh->GetIndexCount());

m_Direct3D->GetWorldMatrix(worldMatrix);
```

The shader class splits each light's colour values and positioning and sends those to the vertex shader .hlsl. The matrices and buffers for the lights are processed, finding the position of the light, any vertex and their positions from each other, and calculating the normal or any vertex.

```
// 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);

// Calculate the position of the vertice as viewed by the light source.
 output.lightViewPosition = mul(input.position, worldMatrix);
output.lightViewPosition = mul(output.lightViewPosition, lightViewMatrix);
output.lightViewPosition = mul(output.lightViewPosition, lightProjectionMatrix);

output.lightViewPosition2 = mul(input.position, worldMatrix);
output.lightViewPosition2 = mul(output.lightViewPosition2, lightViewMatrix2);
output.lightViewPosition2 = mul(output.lightViewPosition2, lightProjectionMatrix2);

// Stores the texture coordinates for the pixel shader.
 output.tex = input.tex;
```

```
// Calculate the normal vector against the world matrix only.
 output.normal = mul(input.normal, (float3x3)worldMatrix);

// Normalize the normal vector.
 output.normal = normalize(output.normal);

 // Calculate the position of the vertex in the world.
 worldPosition = mul(input.position, worldMatrix);

 // Determine the light position based on the position of the light and the position of the vertex in the world.
output.lightPos = lightPosition.xyz - worldPosition.xyz;

// Normalize the light position vector.
 output.lightPos = normalize(output.lightPos);o
utput.lightPos2 = lightPosition2.xyz - worldPosition.xyz;o
utput.lightPos2 = normalize(output.lightPos2);

return output;
```

The pixel shader .hlsl then receives this data and using the matrices for both lights to calculate projected texture co-ordinates for each mesh. Calculations are done on whether a pixel is seen by the light and the distance from the light this pixel has using the depth map calculated previously.

```
// Determine if the projected coordinates are in the 0 to 1 range.  If so then this pixel is in the view of the light.
if((saturate(projectTexCoord.x) == projectTexCoord.x) && (saturate(projectTexCoord.y) == projectTexCoord.y))
{
	// Sample the shadow map depth value from the depth texture using the sampler at the projected
texture coordinate location.
	depthValue = depthMapTexture.Sample(SampleTypeClamp, projectTexCoord).r;

	// Calculate the depth of the light.
	lightDepthValue = input.lightViewPosition.z / input.lightViewPosition.w;
	// Subtract the bias from the lightDepthValue.
	lightDepthValue = 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 the light is in front of the object then light the pixel, if not then shadow this pixel since an object
	(occluder) is casting a shadow on it.
	if(lightDepthValue < depthValue)
	{
		// Calculate the amount of light on this pixel.
		lightIntensity = saturate(dot(input.normal, input.lightPos));
		if(lightIntensity > 0.0f)
		{
		// Determine the final diffuse color based on the diffuse color and the amount of light
	intensity.

		color += (diffuseColor * lightIntensity);
		}

		}
```

```
}

projectTexCoord.x = input.lightViewPosition2.x / input.lightViewPosition2.w / 2.0f + 0.5f;
projectTexCoord.y = -input.lightViewPosition2.y / input.lightViewPosition2.w / 2.0f + 0.5f;
```

Then the same calculations are done for the second light, carrying on the colour value the first light had on this pixel.

```
// Determine if the projected coordinates are in the 0 to 1 range.  If so then this pixel is in the view of the light.
if ((saturate(projectTexCoord.x) == projectTexCoord.x) && (saturate(projectTexCoord.y) == projectTexCoord.y))
        {
        // Sample the shadow map depth value from the depth texture using the sampler at the projected texture coordinate location.
        depthValue = depthMapTexture2.Sample(SampleTypeClamp, projectTexCoord).r;
        lightDepthValue = input.lightViewPosition2.z / input.lightViewPosition2.w;
        lightDepthValue = lightDepthValue - bias;

                if (lightDepthValue < depthValue)
                 {
                // Calculate the amount of light on this pixel.
                lightIntensity = saturate(dot(input.normal, input.lightPos2));
                if (lightIntensity > 0.0f)
                {
                color += (diffuseColor2 * lightIntensity);
                }
                }
                }
```

Then finally the pixel shader .hlsl uses the colour value carried on through the previous calculations and applies it to the texture colour affecting that pixel of this mesh to produce the final colour and shading affected on this pixel.

```
        color = saturate(color);//setting final colour
        // Sample the pixel color from the texture using the sampler at this texture coordinate location.
        textureColor = shaderTexture.Sample(SampleTypeWrap, input.tex);
        // Combine the light and texture color.
        color = color * textureColor;
         return color;
```

The colour calculated with reference to both lights is now applied to the scene and with using the depth maps, shadows now appear for each mess, with pixels behind other pixels in either of the light view not having their colour value affected by that light.

There is a mesh in the scene that is affected by shader .hlsl that allow for this mesh to be tessellated and manipulated by varying amounts inputted by the user. The “TessMesh” as it is called in the application is made up of quad shapes, the number of which is affected by the manipulating tessellation shader class, or “ManiTessShader.cpp”.
The mesh is initialized and then goes through the shadowing process previously described. Then in

the “RenderScene” function of the applications main .cpp as well as being affected by the shadow shader .cpp the “ManiTessShader.cpp” is applied to it also.

m_TessMesh->SendData(m_Direct3D->GetDeviceContext());

m_ManiTessShader->SetShaderParameters(m_Direct3D->GetDeviceContext(), worldMatrix, viewMatrix, projectionMatrix, m_TessMesh->GetTexture(), TessAmount, ourTime, Height, Freq);

The last four values being taken in by this are user inputs that affect the effect the hull and domain shaders have on this mesh, of which will be discussed shortly.

m_ManiTessShader->Render(m_Direct3D->GetDeviceContext(), m_TessMesh->GetIndexCount());

The shader.cpp takes the camera’s matrices and the users inputs, puts them into appropriate buffers and sends them to the vertex shader .hlsl.
The user input for the amount of tessellation applied to the shape is taken here and applied in the hull shader hlsl

tessellationPtr->tessellationFactor = Tess;

The vertex position is declared as well as texture co-ordinates and then sends the data next to the pixel shader .hlsl which itself calculates the colour of each pixel according to texture and ambient lighting. The data is then sent to the hull shader .hlsl where the tessellation is implemented to do so in quad shapes made of two internal triangles. The tessellation input from the user is applied here as the variable “tessellationFactor” and describes how many quads will be applied to the mesh.

```
// Set the tessellation factors for the four edges of the quad.
 output.edges[0] = tessellationFactor;
output.edges[1] = tessellationFactor;
output.edges[2] = tessellationFactor;
output.edges[3] = tessellationFactor;
 // Set the tessellation factor for tessallating inside of the quad.
 output.inside[0] = tessellationFactor;
output.inside[1] = tessellationFactor;
```

```
[domain("quad")]
[partitioning("integer")]
[outputtopology("triangle_ccw")]
[outputcontrolpoints(4)]
[patchconstantfunc("PatchConstantFunction")]
```

The control point for this mesh where tessellation is set to and the manipulation of the mesh centres around is also defined. This is what shadowing has an effect on at the moment.
Once the hull shader hlsl has finished the data is transferred to the final domain shader .hlsl, where manipulation is calculated and the user input is referenced.

```
float3 v1 = lerp(patch[0].position, patch[1].position, 1 - uvwCoord.y);
float3 v2 = lerp(patch[2].position, patch[3].position, 1 - uvwCoord.y);
float3 n1 = lerp(patch[0].normal, patch[1].normal, 1 - uvwCoord.y);
float3 n2 = lerp(patch[2].normal, patch[3].normal, 1 - uvwCoord.y);
```

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

vertexPosition = lerp(v1, v2, uvwCoord.x);
normalPosition = lerp(n1, n2, uvwCoord.x);
texPosition = lerp(t1, t2, uvwCoord.x);
```

The user inputs “height” and “frequency” are called here, and affects the translation of the vertices of the mesh.

```
//offset position based on sine wave: divide position by frequency before adding time
vertexPosition.z += height *sin(vertexPosition.x / frequency + time);
vertexPosition.z += height *cos(vertexPosition.y / frequency + time);

//modify normal
normalPosition.x = 1 - cos(vertexPosition.x + time);
normalPosition.y = abs(cos(vertexPosition.x + time));

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

This data is then sent back to the scene and a tessellated plane will appear moving forward and backward in a flag like motion at varying degrees as decided by the user.

Shortfalls

The main shortfall for this application would be that the Depth and Shadow shaders could have been combined with other shaders used in this application for relevant processes. For example applying shadow shader functionality to the manipulation and tessellation pixel shader .hlsl could allow for lighting to affect the shapes vertices and create appropriate shadows. This could also be applied to the Geometry shader also so as to create shadowing for the shapes created rather than just the original vertices.

When tessellating, it would be beneficial to the user if the frequency could only be increased and decreased within the range of values that has a visual effect on the planes manipulation. A solution to this would be to having an additional If() statement when taking the user input that only affects the frequency values within this range. This solution could also be used for the height values as well.

Applying blurring to the shadows of the scene would have been an appropriate solution to the harsh artefacts of the shadows as they are in the scene, and would have created more appealing soft shadows.

The Geometry shader could have been worked upon to provide far more functionality, such as to create particle affects. The implementation of terrain using a manipulated plane and perlin noise would have been appropriate for creating a immersive scene. In future adaptation of this application these will be main features to work towards.

It has become apparent that the tessellated plane mess does not appear on certain computers. It seems to be a hardware issue however on further investigation it would be appropriate to find a solution to this predicament with amendment to code if possible.

References

Wolf 3d model .obj
“3dregenerator”. 2012. TF3DM. http://tf3dm.com/3d-model/wolf-61313.html

Perlin noise .bmp
JTippetts. 2ND December 2007. GameDev.com. http://www.gamedev.net/blog/33/entry-1605586-isometric-art/

Techniques for specifically soft shadowing
RasterTek. Latest update 2015. Rastertek.com. http://www.rastertek.com/tutdx11.html