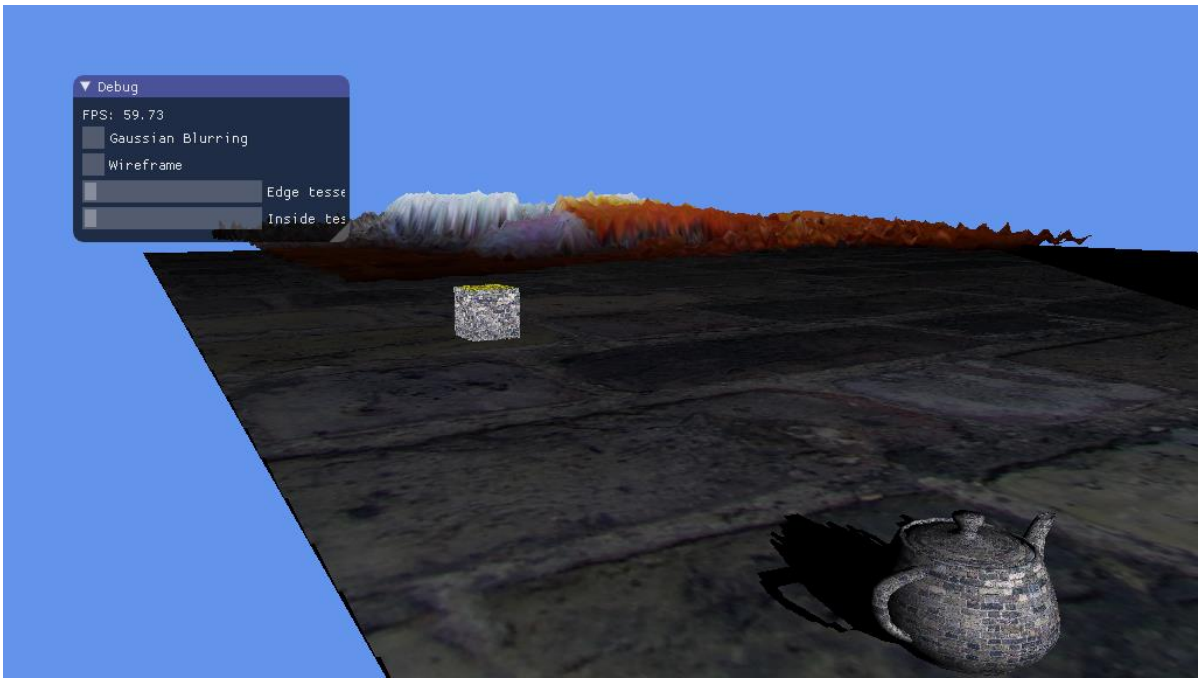
Graphics Programming with Shaders

Andrew Peters

1502220

Coursework Assessment



User Controls:

Accessing the Executable File:

- Go to the debug folder in the projects and open E10_ShadowMaps. exe

Moving the Camera:

- Forward -> w
- Left -> A
- Right -> D
- Backwards -> S
- Up -> Q
- Down -> E
- Look around-> Press space to unlock rotation and move using the mouse position, press space again to lock the camera rotation

ImGui:

- Wireframe -> click the checkbox
- Gaussian Blur -> click the checkbox
- Edge Tessellation Factor -> move slider to change tessellation value
- Inside Tessellation Factor -> move slider to change tessellation value

Code Created:

Different Shader Stages:

Input assembler shader

In this stage, the geometry is specified and the layout which is expected by the vertex shader

Vertex shader

Usually used for transforming the vertex positions from object space into clip space but it can also be used for performing the skinning of the skeletal animation meshes or per vertex lighting .

Hull shader

The hull shader is an optional shader stage and is responsible for determining how much the tessellation stage should tessellate an input control patch.

Tessellator stage

Subdivides a patch primitive according to the tessellation factors specified by the hull shaders

Domain shader stage

This is an optional shader stage and it computes the final vertex attributes based on the output control points from the hull shader and the interpolation coordinates from the tessellator stage

Multiple Lights:

In order to create multiple lights that will rotate around a cube you need to use a light shader class that would have data sent through to a set of pixel and vertex shaders. In the LightShader.h you

need to create a light buffer which would hold an array of the different ambient, diffuse, and position values of each of the lights.

```
{
private:
    struct LightBufferType
    {
        XMFLOAT4 ambient[2];
        XMFLOAT4 diffuse[2];
        XMFLOAT4 position[2];
    };
```

The same buffer would must be sent to the light pixel shader where you would calculate the light direction, it's intensity and colour for each of the pixels in each array. Padding should be removed from the buffer and in the cpp file the light should use an float 4 in order to get the position.

```
lightPtr = (LightBufferType*)mappedResource.pData;
lightPtr->ambient[0] = light[0]->getAmbientColour();
for (int i = 0; i < 3; i++)
{

    lightPtr->diffuse[i] = light[i]->getDiffuseColour();
    lightPtr->position[i] = XMFLOAT4(light[i]->getPosition().x, light[i]->getPosition().y, light[i]->getPosition().z, 0.0f);
}
```

In app1 each of the lights should be initialized and set their position ambient and diffuse colours. This will allow the object being light to use the lights created within the array to affect it.

```
m_Light[0] = new Light;
m_Light[0]->setAmbientColour(0.2f, 0.2f, 0.2f, 1.0f);
m_Light[0]->setDiffuseColour(1.0f, 0.0f, 0.0f, 1.0f);
m_Light[0]->setPosition(30.0f, 10.0f, 20.0f);

m_Light[1] = new Light;
m_Light[1]->setAmbientColour(0.0f, 1.0f, 0.0f, 1.0f);
m_Light[1]->setDiffuseColour(0.0f, 1.0f, 0.0f, 1.0f);
m_Light[1]->setPosition(20.0f, 10.0f, 20.0f);

m_Light[2] = new Light;
m_Light[2]->setAmbientColour(0.2f, 0.2f, 0.2f, 1.0f);
m_Light[2]->setDiffuseColour(1.0f, 0.0f, 0.0f, 1.0f);
m_Light[2]->setPosition(10.0f, 10.0f, 20.0f);
```

Shadow Mapping:

We need to use two sets of shaders to make this work. One to handle getting the depth information and one to handle the shadows. The depth shader will need to have access to its own vertex and pixel shaders. We need to render the scene from the lights position and also use a different view and projection matrix in order to see the shadow from the lights perspective. We store the depth values through the pixel shader after the vertex shader passes in the information and this will allow us to compare the depth in the shadow map with the depth in the shader.

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

float4 main(InputType input) : SV_TARGET
{
    float depthValue;
    float4 color;

    // Get the depth value of the pixel by dividing the Z pixel d
    depthValue = input.depthPosition.z / input.depthPosition.w;
    color = float4(depthValue, depthValue, depthValue, 1.0f);
    return color;
}
```

After we have this information we need to do another render pass but this time we will be rendering the scene as normal. The difference with this though is that we need to add in some extra

calculations so that it renders with the deoth values that we just periously calculated and it will render the scene with a shadow.

```
//// Send geometry data (from mesh)
mesh->sendData(renderer->getDeviceContext());
shadowShader->setShaderParameters(renderer->getDeviceContext(),
    worldMatrix, viewMatrix, projectionMatrix, textureMgr->getTexture("brick"), renderTexture->getShaderResourceView(), shadow_Li
shadowShader->render(renderer->getDeviceContext(), mesh->getIndexCount());
```

Gaussian Blurring:

To do the gaussian blurring we need to have five different render to textures, two ortho meshes and four shaders. We will have five different render functions to complete this, one to render the scene to a texture, one to downscale the image, two to blur the scene and a last one to upscale the image. The reason that we use two blurring functions is that it is a much higher resource drain to do the blurring all in one go than it is to do it separately.

```
void App1::GausianRender()
{
    downScale();

    horizontalBlur();

    verticalBlur();

    upScale();

    RenderToOrtho();
}
```

So, for the blurring vertex shaders we need to pass in the position and texture coordinates to the vertex shader. In the vertex shader we calculate the surrounding pixels in a five-pixel line and output them ready to be used by the pixel shader. The pixel shader than takes these values and changes the colours of the surrounding pixels by the weights set previously in the pixel shader.

During the time that we down scale the mesh we use the texture shader in order to set the image. When this happens, and the texture is passed through to the texture shader, it goes through its usual process storing the texture coordinates for the pixel shader to use. But the interesting thing here is that, when it gets passed to the pixel shader, we return the colours to be used in the final rendering and we are inverting them to give a different effect.

```
    // Sample the pixel color from the texture using the sa
    textureColor = texture0.Sample(Sampler0, input.tex);

    return 1-textureColor;
```

You will need to note that when you are doing the downscaling, blurring and upscaling functions the scene needs to have already been processed as this is post processing, so you need to have these functions come last in the rendering process.

Tessellated Displacement Map:

This part is probably the most interesting thing about this project. The reason for this is because we are using one set of shaders to do multiple things We are first doing vertex manipulation and then tessellating the new displacement map. The actual render function for the tessellation is fairly small, we are only translating and scaling the plane to where we want it and calling on the tessellationShader.

We will be using four shaders to achieve what we are aiming for in this. The vertex shader, hull shader, domain shader, and the pixel shader. In the vertex shader we are passing in the data for the position of the vertices and the coordinates of the texture so that we can modify the vertices based on the colour of each texture coordinate the vertex is attached to.

The shader texture and sampler type as to be put at the top of the vertex shader. We take in the height, this will be the maximum height that the vertex will go to. Declare the initial colour to show what the ground level would be. Sample by using the sample level and use the sample type taking in the texture. Since we are only dealing with a plane we will only need to offset the y axis. We do this this by multiplying the height by the texture colour on the x channel.

```
float height = 10.0f;

input.position.w = 1.0f;

//offset the values absed on the colour of the texture
float4 textureColor = (0.0f, 0.0f, 0.0f, 1.0f);
textureColor = shaderTexture.SampleLevel(SampleType, input.tex, 0);
input.position.y += height* textureColor.x;
```

In the hull shader we are setting a constant function that receives all of the control points for a patch based on the output of the vertex shader. In here we will be changing the tessellation factors by a value in the buffer which will be made by the imGui.

```
ConstantOutputType PatchConstantFuncti
{
    ConstantOutputType output;

    // Set the tessellation factors fo
    output.edges[0] = tessEdges;
    output.edges[1] = tessEdges;
    output.edges[2] = tessEdges;

    // Set the tessellation factor for
    output.inside = tessInside;
```

The Domain shader then takes in the data from the Hull shader and the tessellator and will modify the position, normal and texture coordinates based on the patch control points.

It needs to be noted that all of this will not show when the program is running unless we set the shader texture resource in the pixel shader vertex shader AND the domain shader.

```
// Set shader texture resource
deviceContext->PSSetShaderResources(0, 1, &texture);
deviceContext->VSSetShaderResources(0, 1, &texture);
deviceContext->DSSetShaderResources(0, 1, &texture);
```

In order to modify the values of the tessellation in real time within the project we need to create a buffer that can hold these values and pass them through the hull shader and into the main program, so they can be modified.

```
struct TessellationBufferType
{
    float tessInside;
    float tessEdges;
    XMFLOAT2 padding;
};
```

Critical Reflection:

One of the things that I would have liked to have done would have been to do my vertex manipulation inside of the domain shader instead of the vertex shader. This would have allowed me to tesselate first, so I would have more vertices to manipulate and the overall look would have come out much smoother. I would like to have done more with shadows as I only have basic shadows, and possibly manipulate them using the imGui. One of the things that could have been interesting to do would be soft shadows, so it blends more and aren't as harsh and I think the ultimate goal I would like to set for myself would be to be able to do global illumination. Unfortunately, I was unable to get my geometry shader working properly to allow billboarding to be put in. It would have made a nice contribution if it hadn't made the program crash, so I had to remove it for the time being. The project would have been better if I managed to create a super shader (a shader that does everything) however one of the main issues I had wasn't making the shaders the themselves but putting the shaders together this shows that it wasn't the fact that I didn't have the knowledge to do this and more of a lack of understanding of how to put this together.

References:

https://johnsgamedev.com/2017/01/20/directx11-programming-with-shaders/