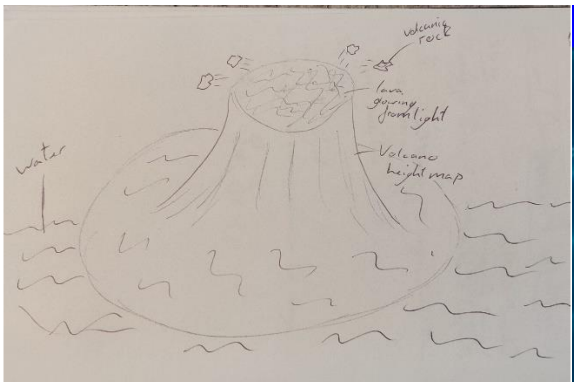
CMP 301 Graphics Coursework Documentation

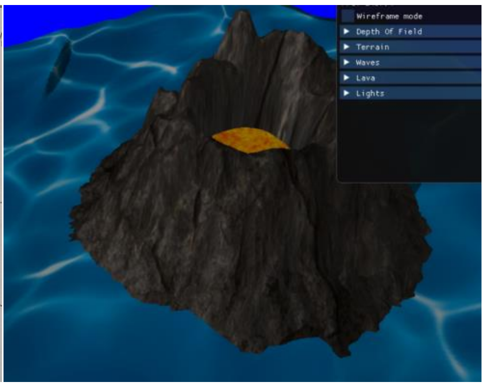
By Joseph Roper

Scene Overview

My scene is of an active volcano in the middle of the ocean. Thus, it contains a heightmap manipulated to look like a volcano surrounded by a plane that has been changed through the vertex shader to represent water. Underneath the water and volcano is a showcase of my spotlight which can be scene from the second camera perspective in the top left of my scene. I have added a post processing depth of field effect which makes things further away blurry and clear up close. Everything in my scene is correctly lit and shadowed.

For my brief I had planned to make a volcano scene surrounded by water with campsite models around the volcano. I believe I have achieved most of what was said as I have a volcano with realistic waves surrounding it. However, I wasn't able to find a good campsite model online which could be imported into DirectX and due to time constraints, I couldn't make one. I also did not have the time to make use of the geometry shader through tessellation however I have made great use of the vertex and pixel shader in my program.





User Controls

I have used imGui for the user controls. With the pop-up menu, you can open the depth of field controls where you can change the how blurry the scene gets with offset and how far you can see without blur with range. In the terrain pop up you can change the volcano's height.

The user can also control the wave and lava properties through the gui controls.

I then have the light controls where each pop up is for a different light within that you can change the diffuse and ambient colour using the colour picker however make sure to only have one open at a time because it can cause the application to crash. You can also change the lights position and direction. For the spotlight specific controls, you can also change the cut off range.

The main camera is controlled using WASD for the camera movement as well as E and Q to control the camera height. The mouse changes the camera view.

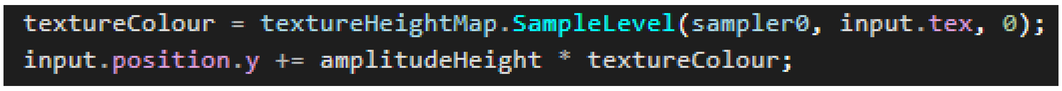
Techniques Demonstrated

Vertex Manipulation

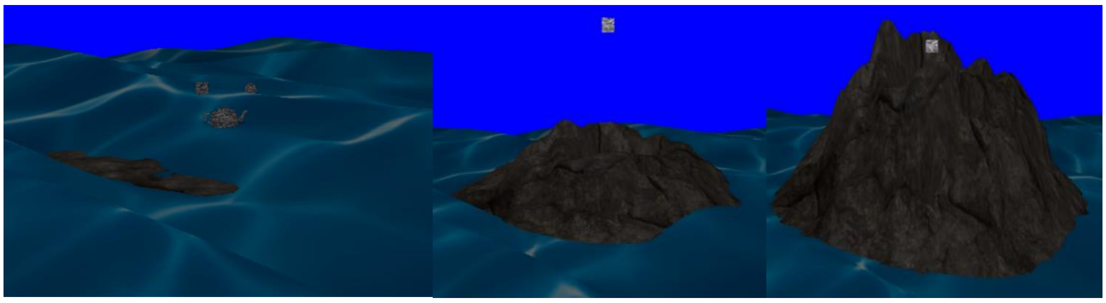
Terrain

The volcano in my scene is made up from a height map texture applied to a plane which has been manipulated.

The terrain shader I have created takes in two textures, the first is a rock texture to be applied to the volcano and the second is the height map. The shader then takes in a variable used to control the height of the volcano named amplitude. The vertex shader then takes the height map and samples it to get the parts of the texture with colour. The plane is then modified along the y axis based off of the amplitude and the texture colour variables to make it, so the planes vertexes increase in the y direction of where there is colour present in the height map.



```
textureColour = textureHeightMap.SampleLevel(sampler0, input.tex, 0);
input.position.y += amplitudeHeight * textureColour;
```

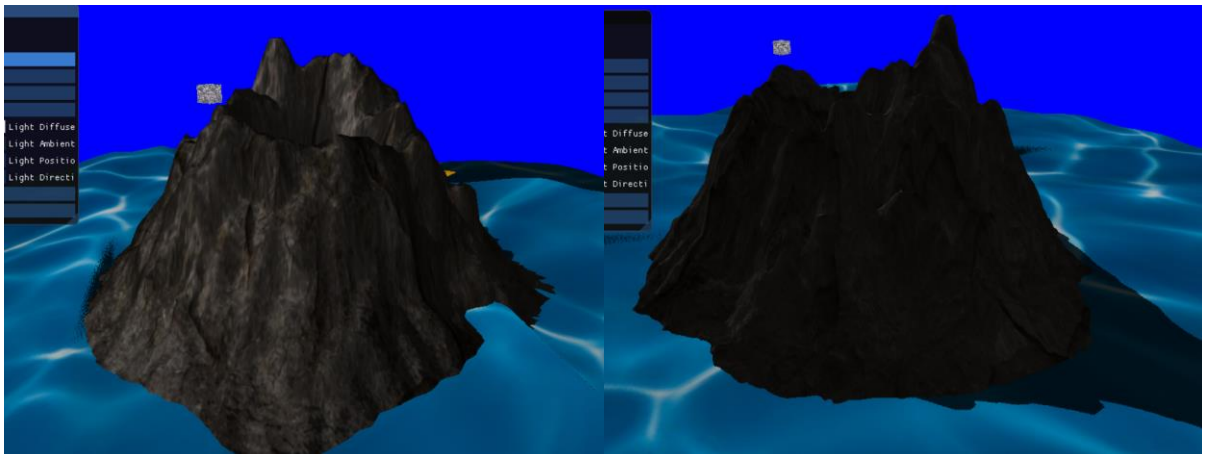


I calculated the normals for the terrains lighting using the parametric equation in the terrain's pixel shader. I did this by sampling the north, east, south, and west of each pixel in the texture. I then needed to calculate the tangent and bitangent by working out the cross product of the values just calculated. The normal is then calculated by getting the cross value of the tangent and bitangent.

$$r_x = \left[1, \frac{2x}{a^2}, 0\right] \quad \text{(tangent)}$$

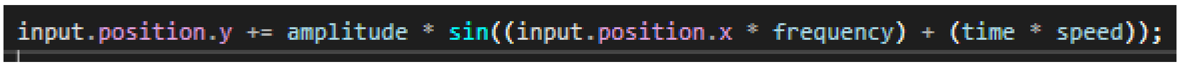
$$r_z = \left[0, \frac{2z}{a^2}, 1\right] \quad \text{(bi-tangent)} \quad \left(\frac{2x}{a^2}, -1, \frac{2z}{a^2}\right) \quad = \text{Our normal}$$

As you can see the normal for my terrain work as the first image demonstrates the front of the volcano where the light is pointing to it is lit up and visible whereas the second image is the backside of the volcano where the light is not visible resulting in this side of the volcano being dark.

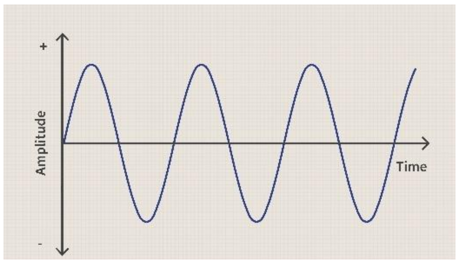


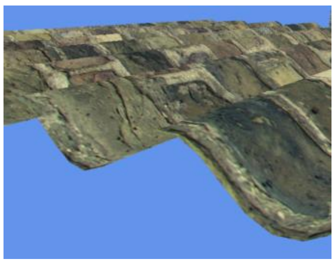
Water

To create the water effect in my scene I first made a plane, using the vertex shader I manipulated the planes y axis to follow the pattern of a sine wave. For this I needed to pass through multiple floats to control the waves effects. Amplitude is used to change the height of the wave, frequency controls the amount of waves present in one cycle and speed controls how fast the waves move this is calculated by using delta time from the programs app.cpp.

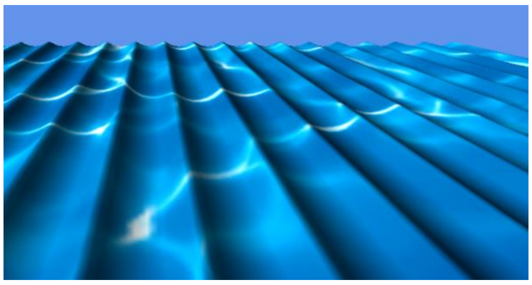


```
input.position.y += amplitude * sin((input.position.x * frequency) + (time * speed));
```

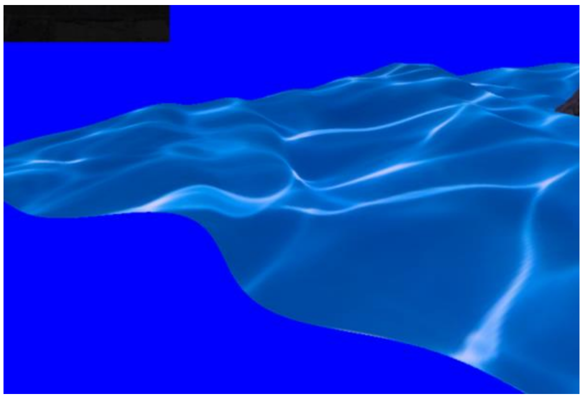




The waves I have in my finsished program use the Gerstner algorithm which was made by Franz Josef Gerstner using fluid dynamics to acurately demonstrate realistic waves. The main change this makes to the sine wave is allowing the wave to also move in the x direction as a sine wave only changes the y and waves in real life move forward in a direction aswell as up and down. Before the vertex shader was changing the y position using sine x but now the planes x value is being manipulated using the co-sine of x. I then made it so you could effect the waves direction based off a float which works by multiplying that float against the x and z vertex manipulation.



Lastly, I made 3 waves and combined them to make it look more realistic. I did this by putting the previous code in a function with 5 parameters, the first takes in the wave information which is a float 4 the x and y of this float are used to calculate the waves direction the z is used for the steepness of the wave and the last value is used for the wavelength. The second parameter is the plane that is being manipulated, the third and fourth parameters are the binormal and tangent used to calculate the waves normals for lighting and finally the fifth parameter is an integer used for a switch statement controlling whether wave or normal data is returned. Using the plane objects input data I run the wave function three times with different wave information, this then results in the complete gerstner waves.

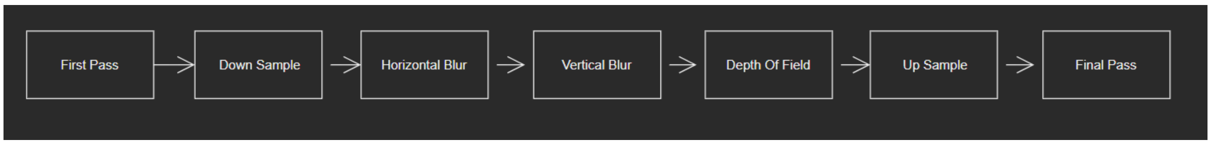


To calculate the normals for my waves I made two float3's the tangent to calculate the normals for the x and bi normal for the z. These then go over the same calculation of the waves although now with steep ness as the top of the wave is more likely to be brighter than the bottom. The cross product of these variables is the calculated and normalised. Resulting in correctly lighted waves.

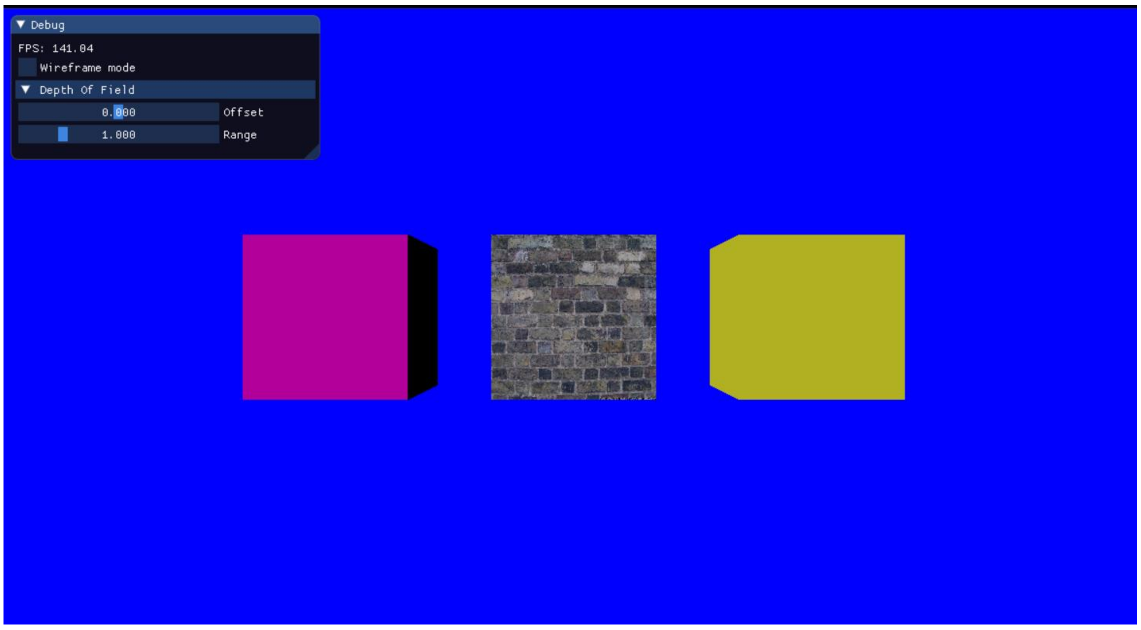
Post Processing

For the post processing of my scene, I have created a depth of field effect. To make this I created multiple render textures It first starts with the scene texture which is used in the programs first pass. The first pass renders all my scene's geometry then applies it to the render texture which is then used in the down sample pass which creates an ortho mesh to be used within the texture shader. The down sample pass uses a render texture which is half the screens width and height and basically zooms in the scene to then have a horizontal and vertical blur applied to it using the same method, however this time it uses the appropriate shaders. The horizontal and vertical blur stages for the post processing effect mainly make use of the pixel shader. In the pixel shader it applies a blur effect to the texture by manipulating the colour value (go to the presentation to see how it's done.

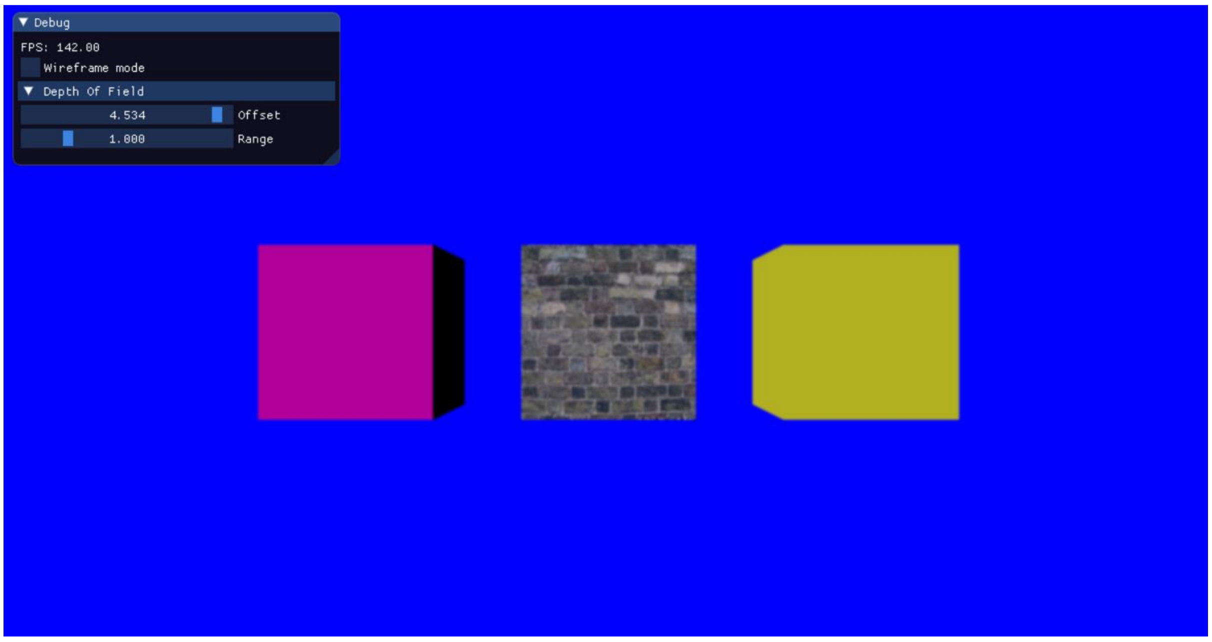
The scene texture is then sent to the depth of field shader which takes in two changeable floats as its parameters, depth range and the offset which the user can control during the application runtime to affect the depth of field blur effect. It also takes in the far and near plane of the scene as well as the first scene texture.



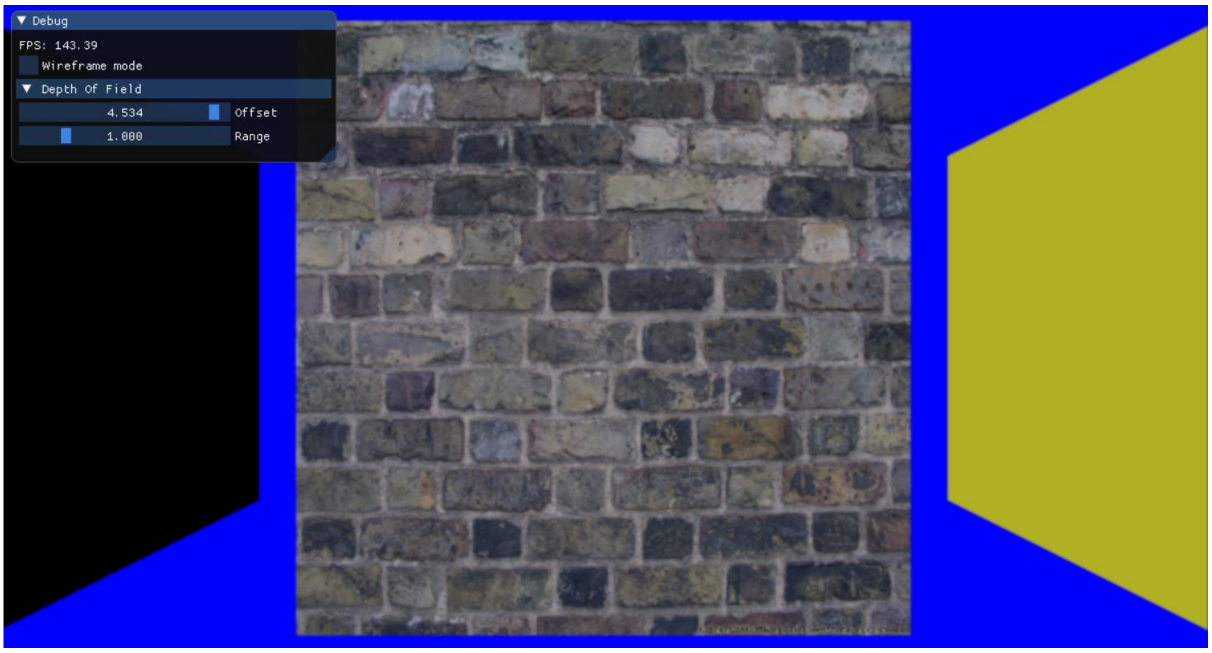
The depth of field affect is calculated by using the first scene texture with nothing applied to it as well as the blurred texture and an empty texture. These are sampled based on the texture coordinates of the ortho mesh assigned to the shader. The empty textures coordinates, and the centre of the empty texture is then saved as floats and the value is set to be in the range of 0 – 1. These floats are then multiplied by far plane minus the near plane to make them useable for the scene. They are then used to calculate the blur factor which is used to switch between the normal scene texture and the blurred one based off the cameras viewpoint, resulting is close objects being clear and objects far away being blurry. Finally, the image is sampled by up to the original screen size and rendered.



Debug
FPS: 141.04
Wireframe mode
Depth Of Field
0.000 Offset
1.000 Range



Debug
FPS: 142.00
Wireframe mode
Depth Of Field
4.534 Offset
1.000 Range



```
FPS: 143.39
Wireframe mode
Depth Of Field
4.534 Offset
1.000 Range
```

Lighting & Shadows

Directional Lights & Shadows

I have three lights is my scene stored within an array, there are two directional lights and one spotlight which can all be controlled whilst the application is running. Each directional light has its own shadow map generated using the depth shaders based on what object is passed to it. I have separate depth and shadow shaders for the water and terrain as the vertex manipulation must be accounted for when calculating the depth and forming the shadow map. Within the depth hlsl files the depth value is calculated in the pixel shader based off the objects position within the world space in relation to the light.

This information is applied to the shadow map which is then used in the shadow pass which I have placed in the programs first pass. The shadow pass sends the object data to their specific shadow shader for example the water plane will go to the water shadow shader. The shadow shaders take in the shadow map created in the depth pass along with other variables needed to calculate the objects vertex manipulation. In the shadow pixel shader hlsl file it uses the shadow map to define which areas of the object need to be darkened to represent the shadow. For the light passed through the shadow shader parameters, the projected texture coordinates are calculated. If the projected texture coordinates are outside of the 0 -1 range, then lighting is not calculated for those pixels.

Spotlight

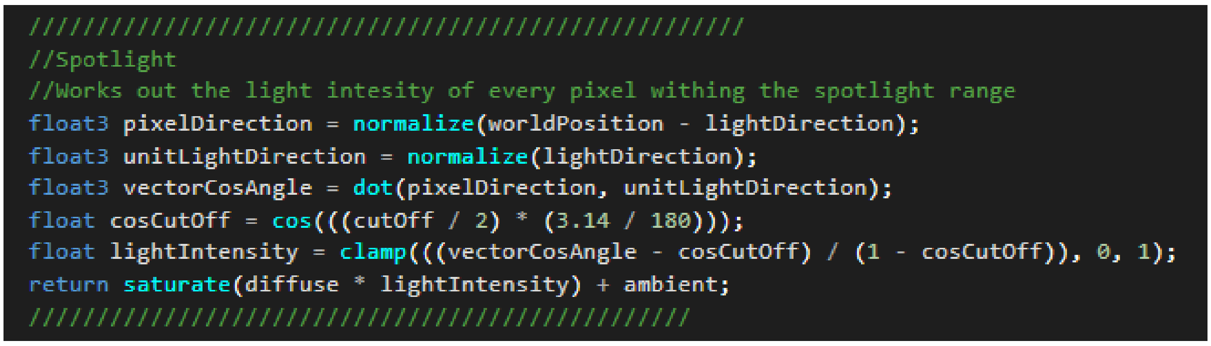
For my application I have created a spotlight which is displayed in the top left of the programs screen on a separate screen.

To make the second screen I first needed to create a second camera and ortho mesh along with another render texture. The second cameras position and angle are initialised in the app1.cpp along with the ortho mesh set to be a quarter of the screen size. I created a function called camera2Pass which is ran in the render within the function I have a plane which is rendered using the spot light shader onto the render texture created. Within the final pass this render texture is applied to the other mesh created.

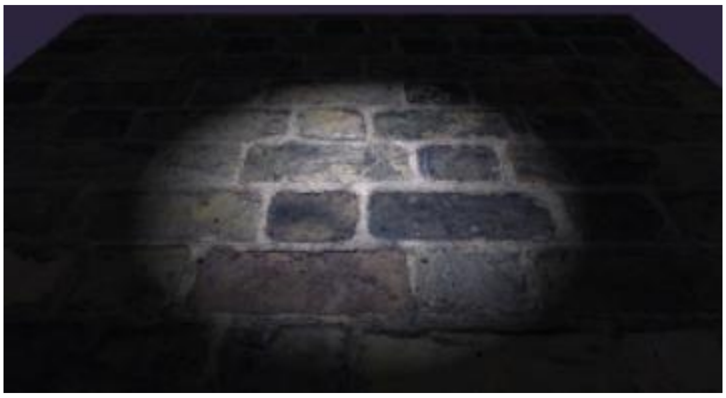
The spot light shader takes in a texture for the object the light is being applied to have a texture. It then takes in the light that is acting as the spot light and a changeable float named spotlightCutof as parameters. The shader then passes this information off to the vertex and pixel shaders for the spotlights hlsl files to use. Not much is done in the vertex shader it is all calculated in the pixel shader as I am wanting to change the objects brightness and colour within a specific area.

In the pixel shader I use the lights position and direction as well as the objects position to make a cone shape with the lights radius based off of the changeable float. To do this I first calculate the pixels direction by normalizing the world position of the object minus the lights direction.

Secondly I make a vector called vectorCosAngle and input the dot product of the pixels direction and the lights direction. This will provide the cone shape with the cutoff variable used to increase and decrease the radius of the lights cone shape.



```
/////////////////////////////////////////////////////
//Spotlight
//Works out the light intesity of every pixel withing the spotlight range
float3 pixelDirection = normalize(worldPosition - lightDirection);
float3 unitLightDirection = normalize(lightDirection);
float3 vectorCosAngle = dot(pixelDirection, unitLightDirection);
float cosCutOff = cos(((cutOff / 2) * (3.14 / 180)));
float lightIntensity = clamp(((vectorCosAngle - cosCutOff) / (1 - cosCutOff)), 0, 1);
return saturate(diffuse * lightIntensity) + ambient;
/////////////////////////////////////////////////
```



Critical Reflection

During this module I have learned a lot and gained a new understanding of the graphics rendering pipeline and what it does especially at the vertex and pixel shader stages. I am very proud of the waves I have created using the gerstner wave algorithm however I struggled to get the waves normals working along side the shadows as the shadow shader renders another plane on top of the one with correctly lit normals making the user unable to see the normals in effect. To solve this, I would need to change the shadow shader used in the lab work however I did not have the time. I also was unable to render the spotlight on objects which already made use of the pixel shader as the code couldn’t overlap without rendering the objects twice, so I chose to keep the water and volcano planes as they are and highlight the spotlight effect on a normal unchanged plane. To solve this issue, I believe I would need outside help to get a better understanding however due to going back home for winter I didn’t have the means to do so.

I wish I could have made use of the geometry shader and implemented tessellation so I would have a clear understanding and capability to replicate the technique in future projects. However, I believe with a bit more time I could have fit it into the program.

In my next code project, I will definitely make use of imGui more whilst testing as to save time between changing variable numbers by constantly restarting the program. When starting future projects, I will take into consideration what can be done with the graphics pipeline to create low poly geometry and post processing effects which can add to the user experience and reduce computing power.

References

www.youtube.com. (n.d.). *Coding Adventure: Gerstner Waves*. [online] Available at: https://www.youtube.com/watch?v=V4yZigMSLiU&ab_channel=Renatus [Accessed 13 Jan. 2022].

Ezra (n.d.). *Island Map | Dinosaur Island*. [online] Available at: http://dinosaur-island.com/tag/island-map/ [Accessed 13 Jan. 2022].

Unsplash (n.d.). *1500+ Water Texture Pictures | Download Free Images on Unsplash*. [online] unsplash.com. Available at: https://unsplash.com/s/photos/water-texture.

www.filterforge.com. (n.d.). *Lava (Texture)*. [online] Available at: https://www.filterforge.com/filters/11415.html [Accessed 13 Jan. 2022].

www.deviantart.com. (n.d.). *Seamless Rock Face Texture by hhh316 on DeviantArt*. [online] Available at: https://www.deviantart.com/hhh316/art/Seamless-Rock-Face-Texture-271675185.