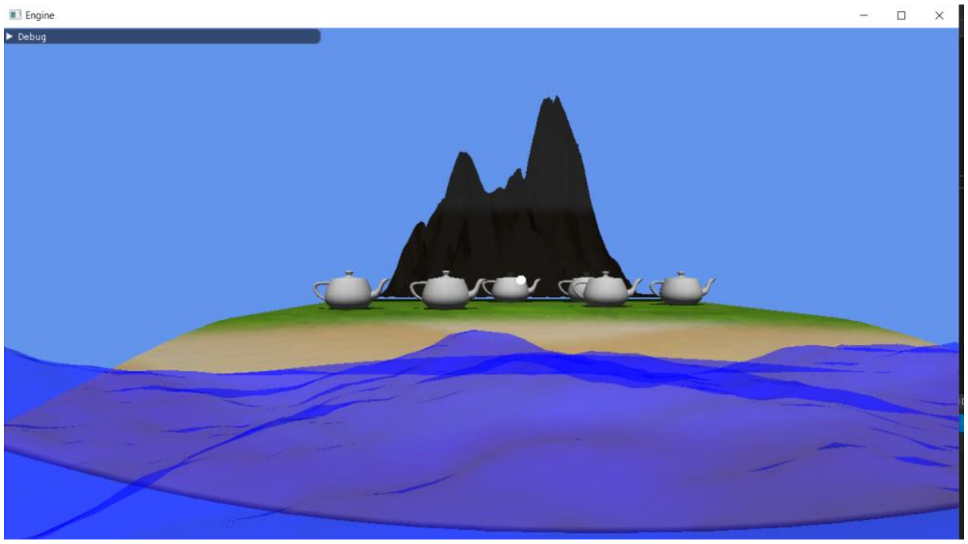
Name: Callum Ahmed

Student number: 1903181

GitHub Username: callum502

Overview

My project is a recreation of “Pelican Town” from the popular game Stardew Valley, the scene contains a mostly flat grassy terrain, some mountains, a beach and an ocean. I planned to add loaded models of houses and shops however unfortunately the model loader was unable to support the models I found online, so I used the teapot models instead to demonstrate the various graphics techniques I have implemented. The scene contains a directional light, a point light and a spotlight. All of the lights can be manipulated using the GUI controls to update direction, position, colour, attenuation, inner/outer angle of spotlight etc. The scene also contains soft shadows which are based on the directional light, the softness of the shadows can be manipulated in the GUI. The ocean in the scene is simply a blue plane which is manipulated in the y axis to create a “wave” effect, I will discuss this technique in more detail later in the report. Both the speed and height of the waves can be altered in the GUI. The scene also contains a toggleable post processing technique which distorts the scene and adds a blue tint. The idea behind this effect is to demonstrate how the scene would look from underwater. The distortion factor of this effect can also be manipulated using the GUI. I have included an image of the scene below.



Techniques

Lighting

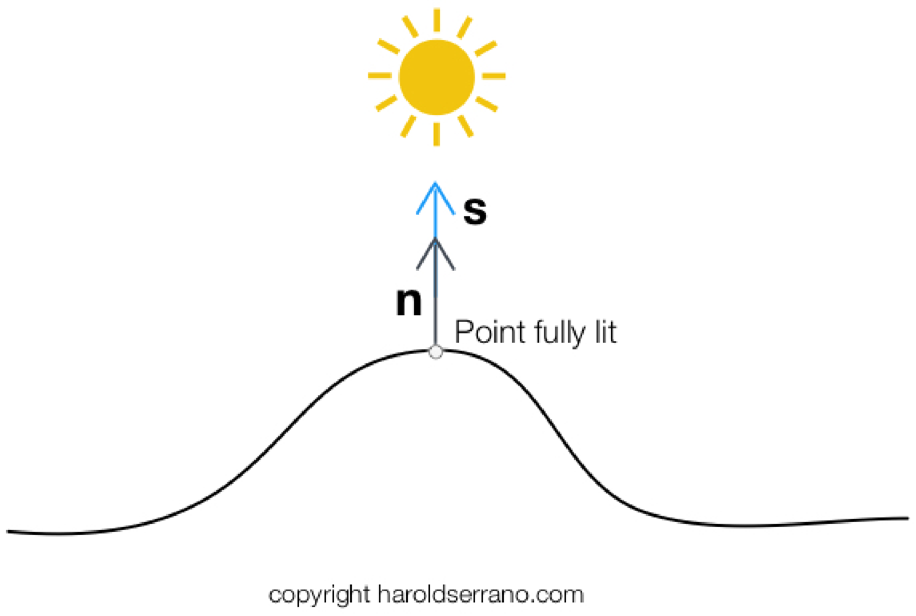
My regular point lights are calculated using the following pseudocode in the pixel shader

```
1. intensity = saturate(dot(normalize(normal), lightVector))
2. distance_factor = saturate(diffuse * intensity)
3. number = smoothstep(pl_distance_falloff, 0, abs(distance))
4. finalcolour = colour * distance_factor
```

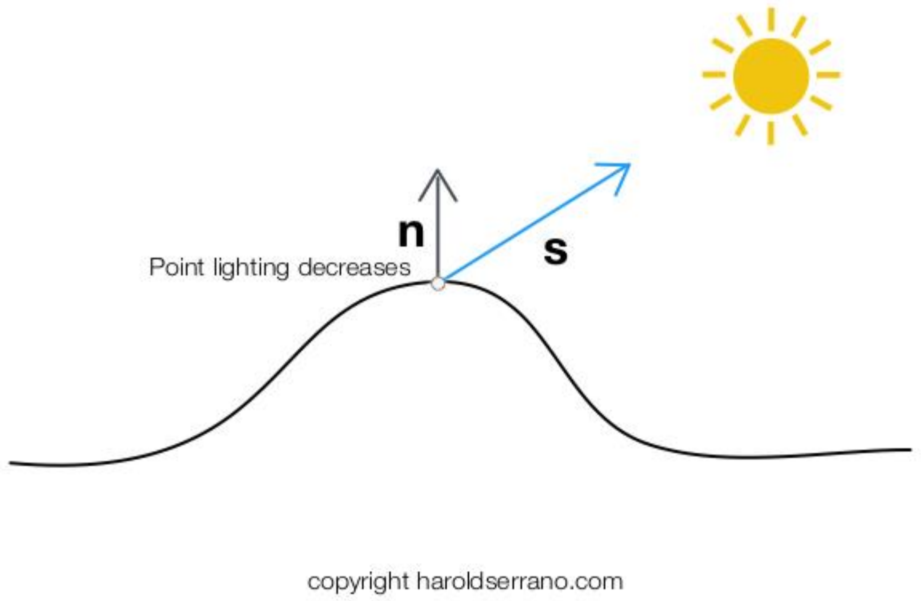
In the code above the light vector is the normalised vector from the pixels position to the position of the point light, this is calculated by subtracting the pixel position from the point lights position. Intensity is then calculated by getting the dot product of the normal and the light vector, intensity can be multiplied by the diffuse to get the final colour from the point light.

The reason we calculated the dot product of the normalised vectors is because the result ranges from 1 to –1, the value ranges from 1 to 0 between the vectors being parallel and perpendicular, and they range from 0 to –1 when the light vector is more than 90 degrees from the normal. This means that as the point light goes from being directly above the normal to being perpendicular to the normal, the dot product will decrease from 1 to 0.

The image below shows how the normal (n) and the light vector (s) would look if point light was directly above the normal, in this case the dot product would be 1. This means in the pseudocode above the intensity would be 1 which would result in the point being fully lit.



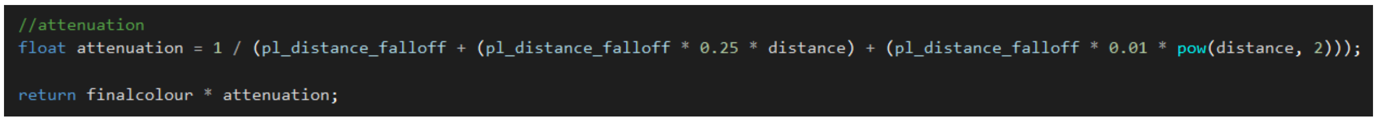
The image below shows the normal and light vector when the point light is further away from the normal of the point, in this case the dot product of the vectors would be between 1 and 0 therefore the pseudocode above would result in the point being partially lit.



Before returning the final colour the program also takes distance into account, it does this by first using the length function to calculate the distance between the point light position and the pixels position. The distance can then be used in the attenuation calculation shown below.

$$\text{Atten} = 1/(\ \text{att0}_i + \text{att1}_i * d + \text{att2}_i * d^2)$$

In the above calculation there is a constant, linear and quadratic attenuation value used. As these values increase the overall attenuation increases and therefore the brightness of the light decreases faster as it gets further away. I implemented this calculation in my pixel shader using the code below.



```
//attenuation
float attenuation = 1 / (pl_distance_falloff + (pl_distance_falloff * 0.25 * distance) + (pl_distance_falloff * 0.01 * pow(distance, 2)));

return finalcolour * attenuation;
```

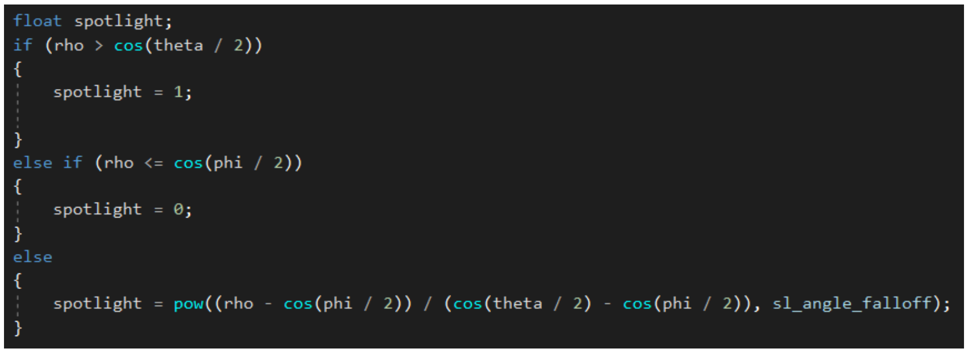
The “pl_distance_falloff” is a value passed into the pixel shader which can be manipulated using the GUI controls. This value is used in the constant, linear and quadratic components of the attenuation calculation therefore when the value is altered using the GUI controls the overall attenuation of the light will be affected. This allows the user to increase or decrease how quickly the light loses its brightness as it gets further away.

The most complex lighting technique I implemented was the spotlight, the spotlight works similarly to the point light except the return value is multiplied by an additional factor calculated from the following equation.

$$spot_i = \begin{cases} 1 & \text{for non- spotlights or if } rho_i > \cos(\frac{theta_i}{2}) \\ 0 & \text{if } rho_i \le \cos(\frac{phi_i}{2}) \\ \left[\dfrac{rho_i - \cos(\frac{phi_i}{2})}{\cos(\frac{theta_i}{2}) - \cos(\frac{phi_i}{2})}\right]^{falloff} & \text{otherwise} \end{cases}$$

In the above equation theta represents the inner angle of the spotlight in radians, phi represents the outer angle and rho represents the dot product of the normal and the light vector which is calculated the same way as intensity from the point light calculation. There is also a falloff value used to alter how quickly the brightness fades from the inner angle to the outer angle of the spotlight.

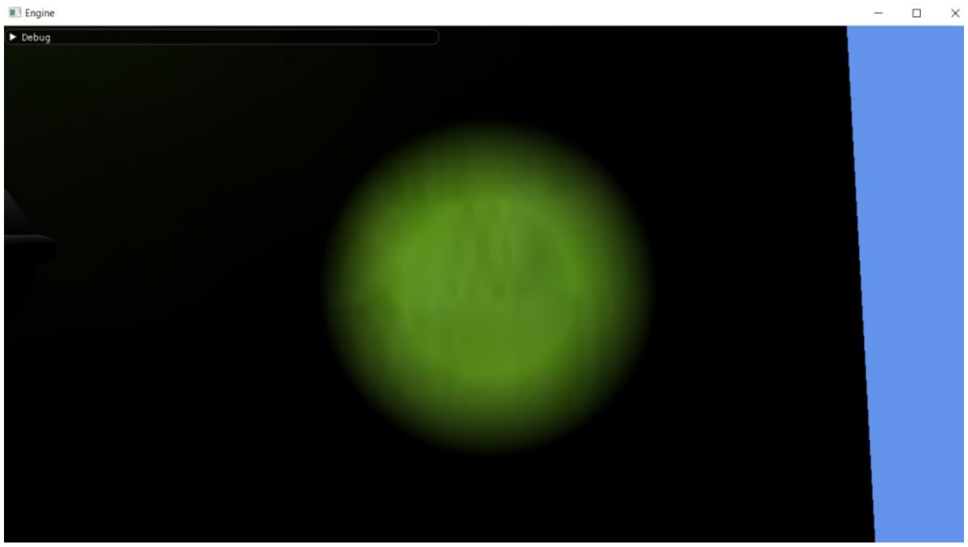
When calculating the spotlight value if rho is greater than cos(theta/2) then the point is within the inner angle of the spotlight and therefore it should be fully lit, so the value is set to one. When rho is less than or equal to cos(phi/2) the point is outside the outer angle of the spotlight and therefore completely unlit so the spotlight value is set to 0, otherwise the point is somewhere between the inner and outer angle and therefore it should be partially lit, in this case the value can be calculated using the equation from the above image. Shown below is how this can be calculated in code.



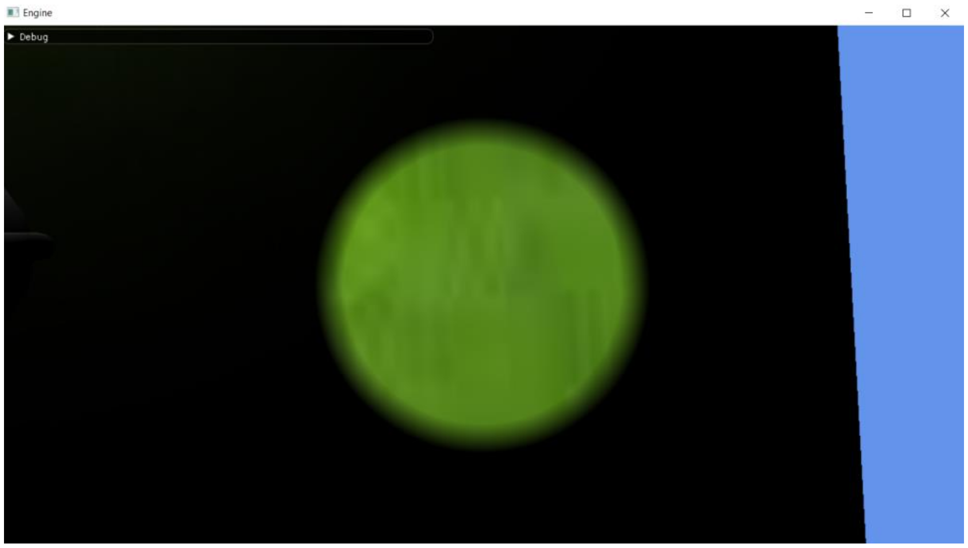
```
float spotlight;
if (rho > cos(theta / 2))
{
    spotlight = 1;

}
else if (rho <= cos(phi / 2))
{
    spotlight = 0;
}
else
{
    spotlight = pow((rho - cos(phi / 2)) / (cos(theta / 2) - cos(phi / 2)), sl_angle_falloff);
}
```

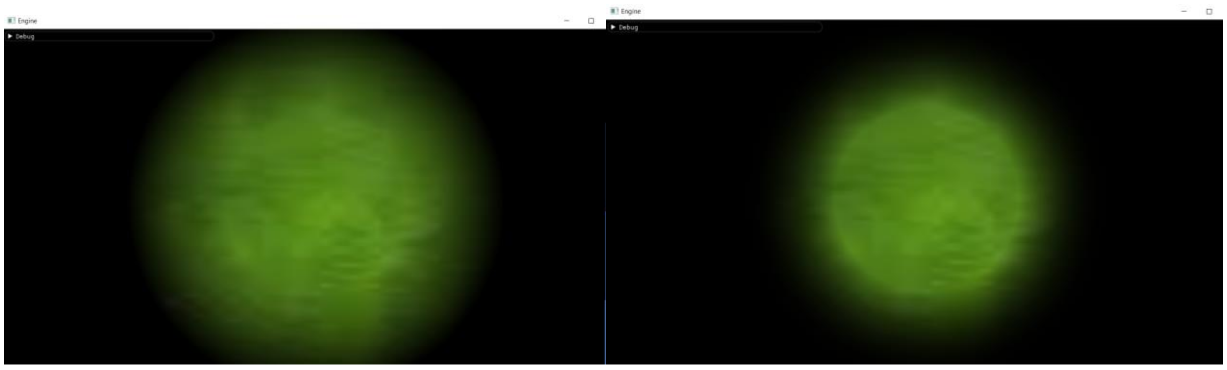
In the above code the values for phi, theta and angle falloff are passed into the pixel shader, these values can all be manipulated using the GUI controls. In the image shown below you can see the spotlight when the inner angle is set to 15 degrees and the outer angle is 30 degrees.



As you can see from the above image, pixels within the inner angle are fully lit, the brightness then decreases as the angle increases towards the outer angle of the spotlight. In the image below you can see how the spotlight is affected when we increase the inner angle to 25 degrees.



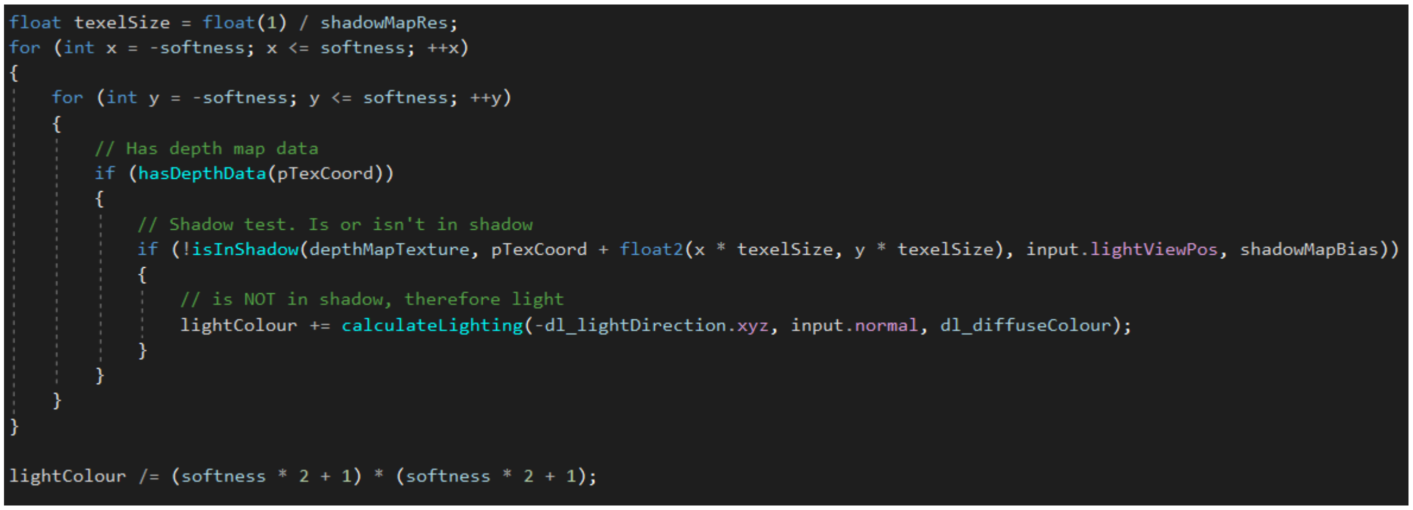
The angle falloff can also be increased in the GUI which simply increases how quickly the brightness falloff when between the inner and outer angle of the spotlight. The image below on the left shows the spotlight with inner angle 25, outer angle 50 and a falloff value of 1. The image on the right shows the spotlight after increasing the falloff value to 3.



As you can see there is a clear decrease in brightness between the inner and outer angles when the falloff value is increased.

Shadows

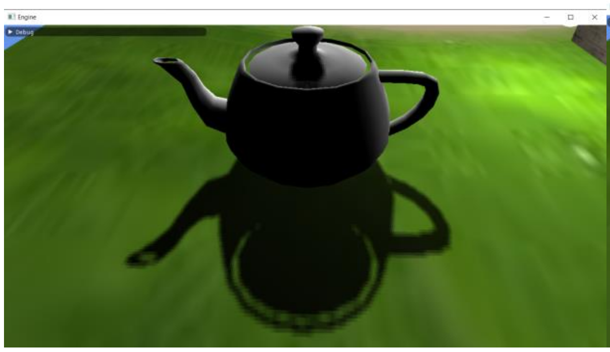
My scene also contains shadows created by the directional light. To render shadows I first need to create a shadow map, I did this by setting the render target to my shadow map, then rendering the scenes depth data from the perspective of the directional light. I then passed this shadow map into my pixel shader, from here I can compare the depth of the shadow map and the depth of the light. If the depth value of the light is less than the shadow map then the point is not in the shadow. Using this knowledge, I implemented percentage closer filtering (PCF) soft shadows. This can be done by sampling the neighboring texels of the targeted texel and then averaging the resulting direction light colour values. This technique is demonstrated in the following snippet of code from my pixel shader.

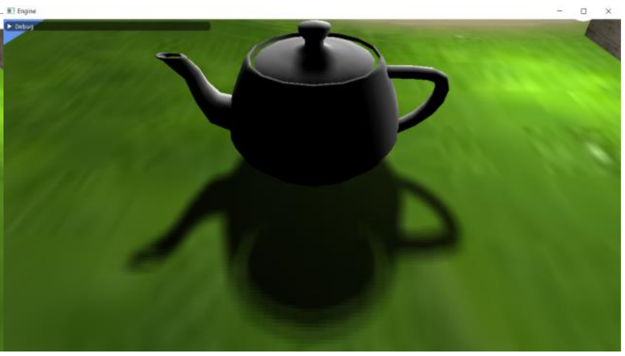


```
float texelSize = float(1) / shadowMapRes;
for (int x = -softness; x <= softness; ++x)
{
    for (int y = -softness; y <= softness; ++y)
    {
        // Has depth map data
        if (hasDepthData(pTexCoord))
        {
            // Shadow test. Is or isn't in shadow
            if (!isInShadow(depthMapTexture, pTexCoord + float2(x * texelSize, y * texelSize), input.lightViewPos, shadowMapBias))
            {
                // is NOT in shadow, therefore light
                lightColour += calculateLighting(-dl_lightDirection.xyz, input.normal, dl_diffuseColour);
            }
        }
    }
}

lightColour /= (softness * 2 + 1) * (softness * 2 + 1);
```

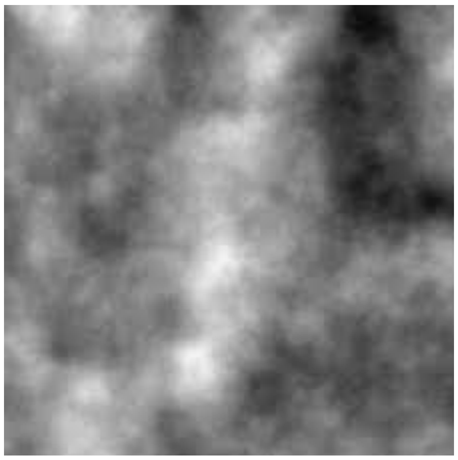
In the above code the values for the “shadowMapRes” and the “softness” are passed into the pixel shader, I simply calculate one divided by the shadow map resolution to find the shadow texel size which is then used to calculate the uv coordinates of the neighboring texels. The softness value can be adjusted using the GUI controls. As the softness increases the pixel shader will sample a higher number of surrounding texels and therefore the shadow will appear more blurred or “soft”, this is demonstrated in the following screenshots where the image on the left shows a shadow with the softness set to 1 and the image on the right shows the shadow with a softness of 3.





Vertex Manipulation

I also used vertex manipulation on a transparent blue plane to simulate water waves, this is done by first passing a gradient noise texture into the vertex shader, this texture is shown below.



The vertex shader of the water then samples this image twice using different directions and speeds for the uv offset. The average value of these 2 samples determines the y value of the targeted vertex. This is demonstrated in the code below.

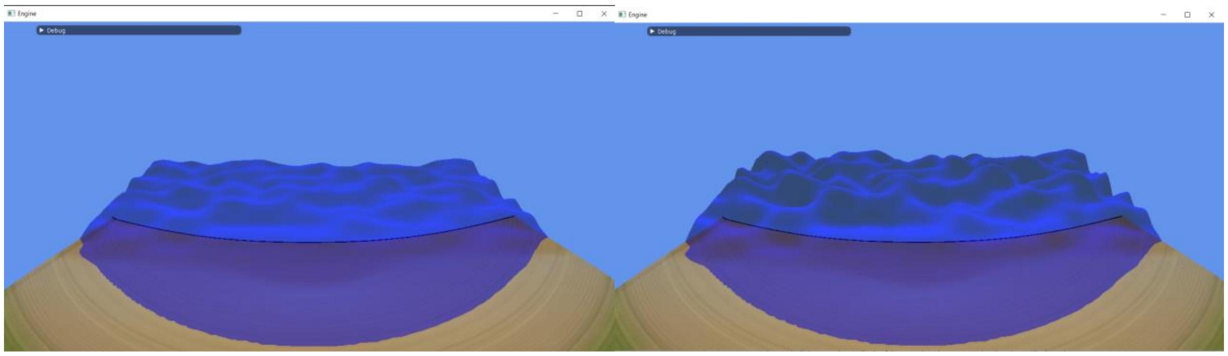
```
 sample = texture0.SampleLevel(sampler0, float2(uv.x, uv.y - time * speed), 0)

sample2 = texture0.SampleLevel(sampler0, float2(uv.x - time * speed * -0.25, uv.y - time *
speed * -0.25), 0)

input.position.y = ((sample + sample2) / 2) * amplitude * height_offset;
```

The code shown above samples the noise texture using the texture coordinates of the targeted vertex however, the coordinates are offset by time multiplied by speed. Time is used to ensure that the y position is dynamic and therefore creates the effect of moving waves, speed is simply a value passed into the vertex shader which can be manipulated using the GUI to make the waves appear faster or slower.

In the above calculation the amplitude is another value passed into the shader which can be manipulated in the GUI. The image below on the left shows the water with an amplitude value of 5 while the image on the right has the amplitude value set to 10.



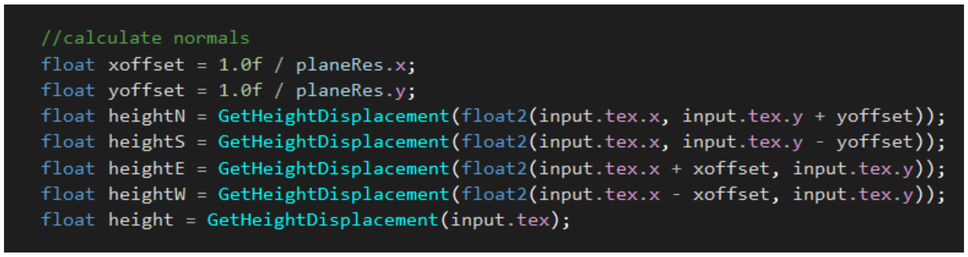
As you can see the height of the waves is much higher when the amplitude value is increased in the GUI. You may also notice that the height increases as the waves get further from the terrain, this is due to the height offset value used in the calculation of the y position shown earlier. The height offset value is used to gradually increase the height of the waves in order to ensure that they do not clip through the terrain. The height offset value is simply calculated using the smoothstep function, this calculation is shown below.

height_offset = smoothstep(0, -30, output.worldPosition.z)

The smoothstep function returns a value between zero and one based on where the z position of the targeted vertex is between zero and negative thirty. This means that as the z positions decrease and therefore the vertices are getting further from the terrain, the height offset should increase from zero towards one. Therefore, factoring this value into the y position calculation will result in the height of the waves increasing as they get further from the terrain.

After calculating the height of the waves, the normals must be calculated, for each vertex the normal is set to the average of the adjacent vertices to the north, east, south and west.

To calculate the adjacent normals the height at these points must first be calculated, this can be done using the same vertex manipulation calculation from earlier but using offset texture coordinates, this is shown in the code below.



```
//calculate normals
float xoffset = 1.0f / planeRes.x;
float yoffset = 1.0f / planeRes.y;
float heightN = GetHeightDisplacement(float2(input.tex.x, input.tex.y + yoffset));
float heightS = GetHeightDisplacement(float2(input.tex.x, input.tex.y - yoffset));
float heightE = GetHeightDisplacement(float2(input.tex.x + xoffset, input.tex.y));
float heightW = GetHeightDisplacement(float2(input.tex.x - xoffset, input.tex.y));
float height = GetHeightDisplacement(input.tex);
```

In the above code the plane resolution has been passed into the vertex shader, this allows for the texture coordinate offset values to be calculated using one divided by the resolution. This calculation is performed to get the offset in uv space as uv coordinates range from zero to one. Now that the heights have been calculated the tangents and bitangents can be obtained from the following set of equations;

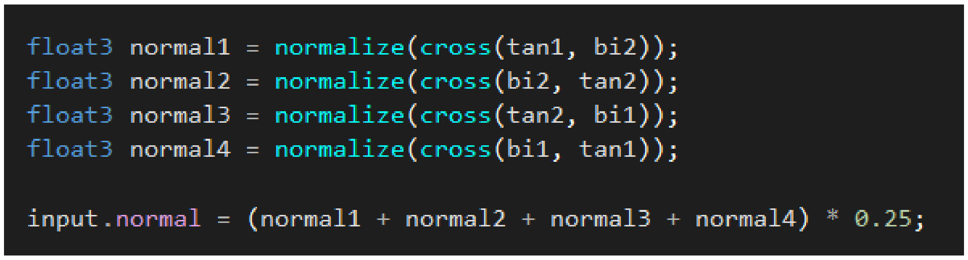
tan1 = normalize(float3(1, heightE - height, 0))

tan2 = normalize(float3(-1, heightW - height, 0))

bi1 = normalize(float3(0, heightN - height, 1))

bi2 = normalize(float3(0, heightS - height, -1))

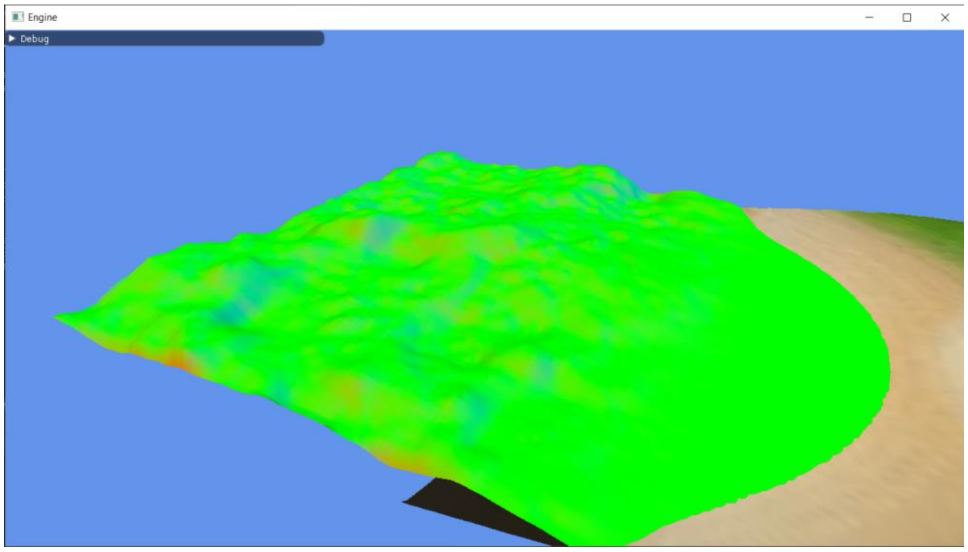
The code below shows how these tangents and bitangents can then be used to calculate the normals.



```
float3 normal1 = normalize(cross(tan1, bi2));
float3 normal2 = normalize(cross(bi2, tan2));
float3 normal3 = normalize(cross(tan2, bi1));
float3 normal4 = normalize(cross(bi1, tan1));

input.normal = (normal1 + normal2 + normal3 + normal4) * 0.25;
```

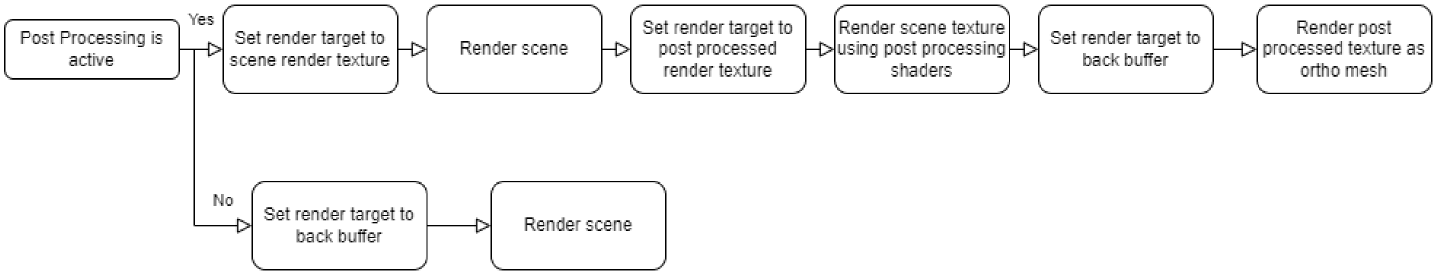
In the above code the normals are calculated by getting the cross product of the previously calculated tangents and bitangents, this works because the cross product returns the line perpendicular to the tangent and bitangent. The normal of the targeted vertex can then be obtained by averaging the four surrounding normals. The image below shows the water when outputting the normals as the colour value in the pixel shader.



The above screenshot is evidence of correct normals as it shows mostly green pixels meaning upward normals however there are small red and blue sections which represent normals positive in the x and z axes respectively. These colours seem to match up with the expected results, which suggests that the normals are being calculated correctly.

Post Processing

My scene also features a toggleable post processing effect, when this effect is not active the scene is simply rendered to the back buffer. However, when the effect is active the render target is set to the scene render texture before rendering the scene. The render target is then set to the post processed render texture, the scene render texture is then rendered using the post processing shaders, this process should result in the post processed render texture being set to a texture of the whole scene after applying the post processing effect. Now the program simply needs to render the post processed render texture to the back buffer, this can be done by creating and rendering an ortho mesh of the post processed render texture. This process is represented in the diagram below.


Post Processing is active
Yes
Set render target to scene render texture
Render scene
Set render target to post processed render texture
Render scene texture using post processing shaders
Set render target to back buffer
Render post processed texture as ortho mesh
No
Set render target to back buffer
Render scene

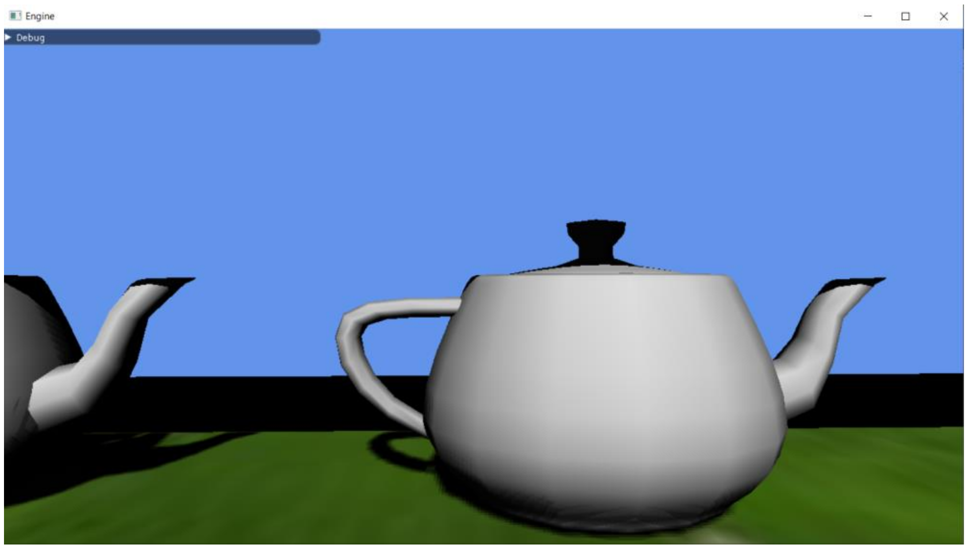
The purpose of the post processing shaders is to create the effect that the user is underwater, it does this by applying a distortion effect as well as a basic blue tint. To apply distortion the pixel shader first samples a noise texture using the texture coordinates of the targeted pixel multiplied by a factor of time, this obtains a random value ranging from 0 to 1 which can be used as the distortion strength. The reason it factors in time is so the distortion values are dynamic, without this the pixel shader would always apply the same static distortion to the screen. The pixel shader then samples the scene render texture using the following calculation to obtain the uv coordinates.

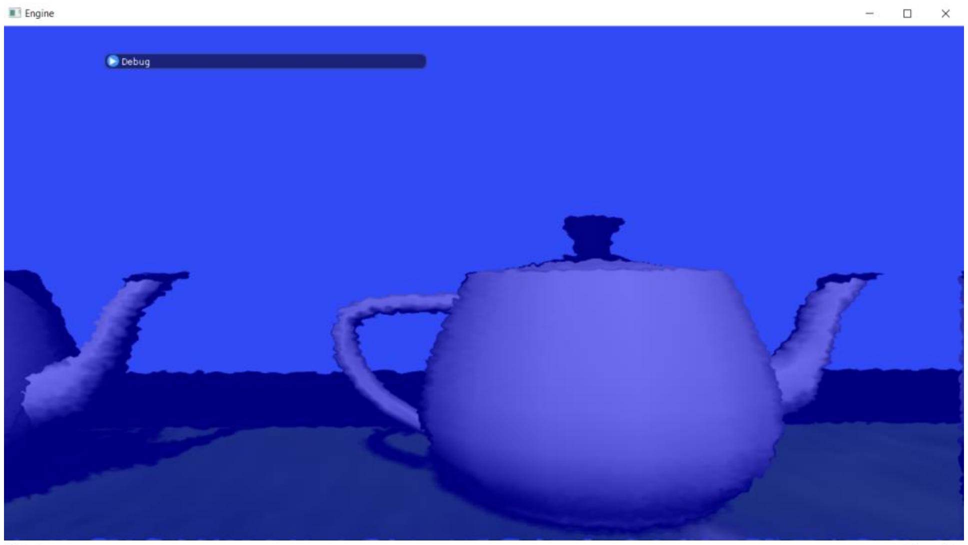
uv = (input.tex.x + distort * distortStrength, input.tex.y + distort * distortStrength))

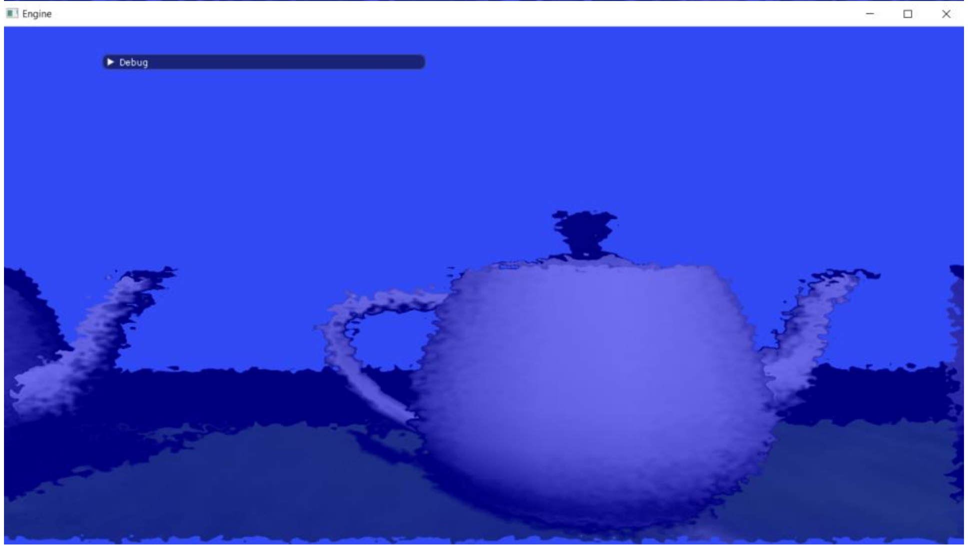
In the above calculation distort represents the value sampled from the noise texture and distort strength is a value passed into the shader which can be manipulated using the GUI controls to increase or decrease the offset in the uv coordinates which creates the effect of manipulating the amount of distortion applied to the scene. After applying this distortion, the pixel shader simply uses the lerp function to apply a blue tint before returning the colour value, this is shown in the code snippet below.

```
//apply blue tint
float4 finalcolour = lerp(colour, float4(0, 0, 1, 1), 0.5);
finalcolour.a = 1.0f;
return finalcolour;
```

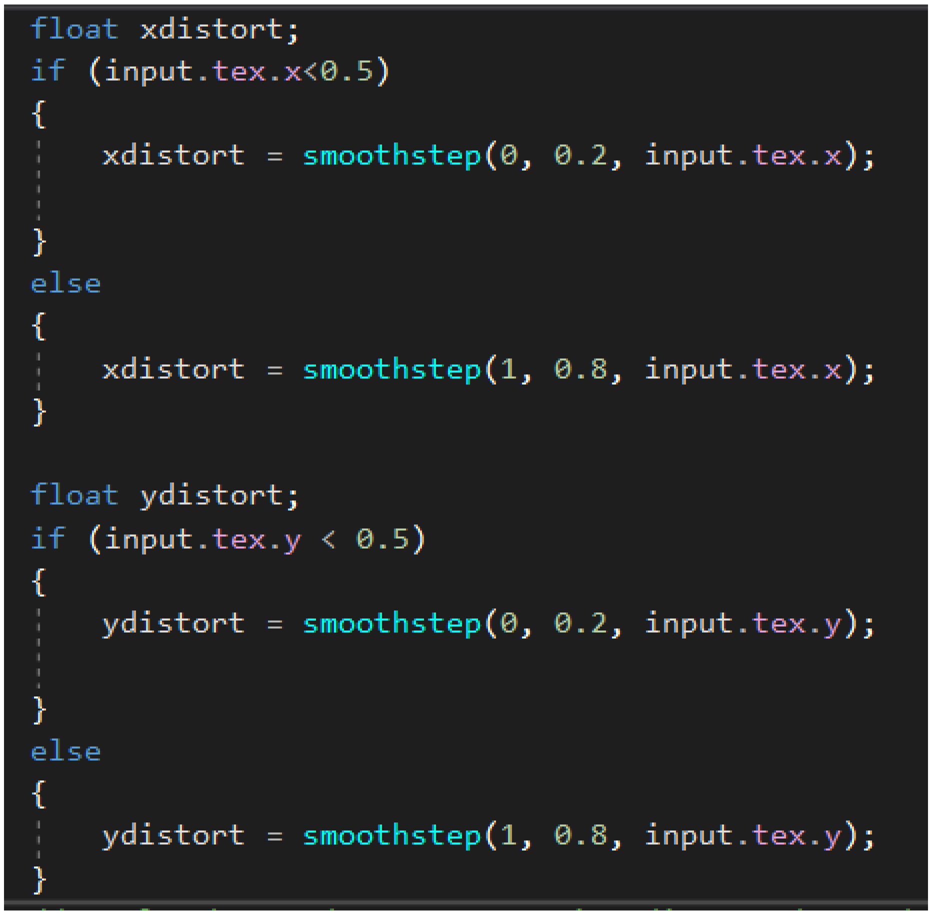
To demonstrate this post processing effect, I have included screenshots below, the first image shows a teapot model without the post processing effect applied, the second image shows the same teapot but with the post processing active and the distortion value set to 2 and the final image show the effect with the distortion value set to 5.







As can be seen from the final image above, when the distortion is set to a high value, pixels near the edge of the screen will sometimes be offset too much and therefore attempt to use uv coordinates outside the range of the texture, this can result in the edges not looking very smooth however to combat this I introduced a distortion strength variable based on the texture coordinates of the targeted pixel, the calculation for this variable is shown below.



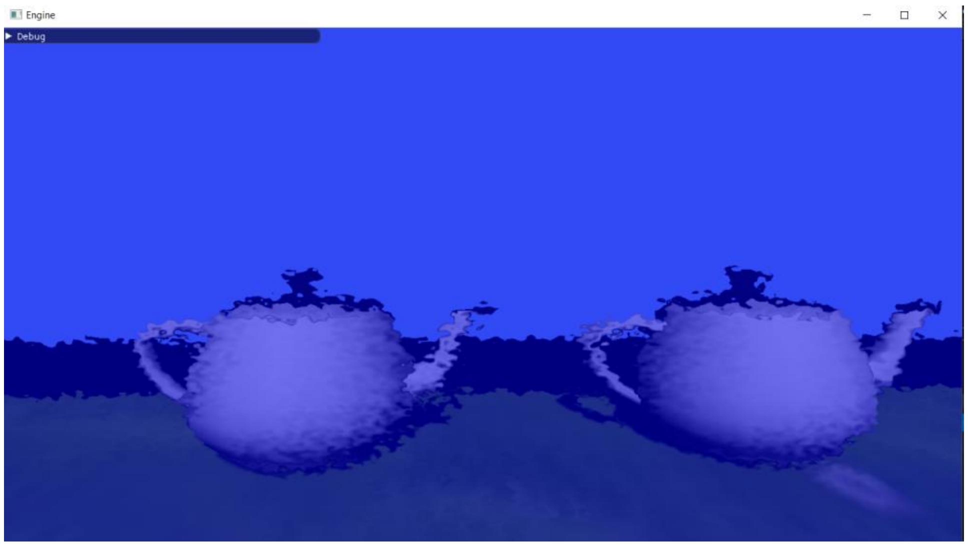
```
float xdistort;
if (input.tex.x<0.5)
{
    xdistort = smoothstep(0, 0.2, input.tex.x);

}
else
{
    xdistort = smoothstep(1, 0.8, input.tex.x);
}

float ydistort;
if (input.tex.y < 0.5)
{
    ydistort = smoothstep(0, 0.2, input.tex.y);

}
else
{
    ydistort = smoothstep(1, 0.8, input.tex.y);
}
```

The code above uses the smoothstep function to obtain an x and y distortion value between 0 and 1 based on how close the texture coordinates are to the edge of the screen. As the texture coordinate increases from 0 to 0.2 the appropriate distortion factor will increase from 0 to 1, similarly as the coordinate decreases from 1 to 0.8 the distortion factor will increase from 0 to 1. When these above values are factored into the distortion calculation the edges appear much smoother as shown in the screenshot below where the distort value is set to 5.



Reflection

Overall, I am quite satisfied with my scene and the techniques used within it. However, there are many improvements and different techniques which could be implemented. I was happy with my implementation of lights as I managed to successfully implement three unique types of lights that can all work together whilst being manipulated using the GUI controls. It was very beneficial for me to learn how to implement a spotlight as this can be used in games to represent a player's flashlight, this can be done by simply setting the direction of the spotlight to the direction the player is looking at.

I was also happy with the implementation of soft shadows as the number of samples used could be adjusted using the GUI controls, this allows the user to see the effect of adjusting the number of samples in real time. I could have improved the scenes shadows by adding point light shadows, this can be done by doing 6 depth passes, one for each direction of the point light. This depth data could have then been used in a cube map which can be used to create the point light shadows.

I was satisfied with the vertex manipulation I used on the water as the use of two samples using different direction and speed values resulted in very dynamic waves. However I could have used a different technique known as Gerstner waves. This technique uses a series of equations to simulate realistic waves.

I thought the implementation of the post processing effect went well as it created the effect I was aiming for. In future I could implement this effect into a game by applying the post processing when the player is underwater, I could also increase the amount of distortion as the player swims deeper underwater, this could be implemented using the same method I discussed earlier to increase the distortion using GUI controls.

References

Harold Serrano *A brief explanation of Game Engine math*
available at: https://www.haroldserrano.com/blog/a-brief-explanation-of-game-engine-math

[Accessed 4/01/2022]

Microsoft *Attenuation and Spotlight Factor (Direct3D 9)*

https://docs.microsoft.com/en-us/windows/win32/direct3d9/attenuation-and-spotlight-factor

[Accessed 4/01/2022]

Etay Meiri *Percentage Closer Filtering*

https://ogldev.org/www/tutorial42/tutorial42.html

[Accessed 4/01/2022]

Vw1522191vw *Toon grass texture 2k tileable*

https://www.freelancer.com.au/contest/Toon-grass-texture-k-tileable-1437066-byentry-24206887?w=f&ngsw-bypass=

[Accessed 12/12/2021]

Tuulijumala *Seamless beach sand texture*

https://www.dreamstime.com/stock-illustration-seamless-beach-sand-texture-wavy-yellow-vector-illustration-image97502392

[Accessed 12/12/2021]

the-night-bird *Petrified 01*

https://www.deviantart.com/the-night-bird/art/Petrified-01-269650946

[Accessed 21/12/2021]

Rusm *Seamless fresh snow background stock photo* https://www.istockphoto.com/photo/seamless-fresh-snow-background-gm185071242-19374577

[Accessed 21/12/2021]

Wilhelm Burger *Synthetic gradient noise*

https://www.researchgate.net/figure/Gradient-noise-images-generated-with-different-integer-hash-functions-listed-in-Prog_fig5_333134067

[Accessed 14/11/2021]