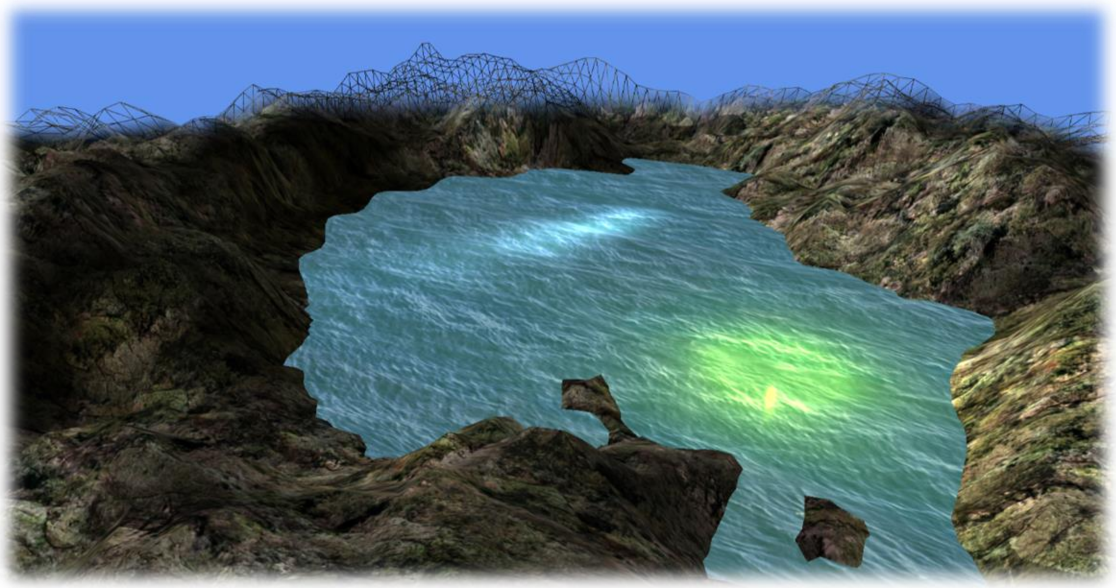
UNIVERSITY of ABERTAY DUNDEE



Graphics Programming with Shaders

Coursework Report

Gavin George

1501029

Table of Contents

Introduction

What follows is a comprehensive description covering the entirety of the coursework submission that this report accompanies. It is intended to familiarise the reader with the content, the development process, the successes and failures and the author's personal experience of the project from start to finish. The author will endeavour to achieve several objectives with this report. Primarily, the report will aim to convey the author's technical understanding of the submitted coursework and relevant knowledge on the subject of Graphics Programming. Furthermore, the report will put key emphasis on areas that show where the author has addressed the coursework requirements, as stated in the brief. Included in this report is detail of the research, techniques and limitations of the project as well as an in depth critical reflection upon the application.

Scene Overview

1.1 Scene Objects

The first main object in the scene is the hillscape which demonstrates procedural terrain generation. The process uses techniques including tessellation, static heightmap displacement and appropriate lighting and shadowing.

The second main object in the scene is the water plane which also makes use of tessellation and has its vertices manipulated. The object demonstrates water simulation and in addition to the tessellation and vertex manipulation it also features specular lighting.

Also present in the scene is a moving sphere to demonstrate shadows in the scene from multiple directions.

1.2 Relation to Brief

Each object fulfils a different set of requirements stated in the brief. The key graphical techniques that have been the priority to implement are vertex manipulation, post processing and lighting and shadows. Each piece of geometry had to be correctly lit and textured after manipulation and it was required to demonstrate pixel manipulation on a pixel by pixel basis. To improve upon this the application would need to include non-trivial tessellation, with correct lighting, and geometry manipulation or generation using the Geometry Shader.

The application displays two types of vertex manipulation, present in the terrain and the water. The terrain uses displacement mapping which consists of sampling a heightmap and applying a transformation to each vertex at the position relative to the sample point. The water is manipulated using a sine algorithm to simulate waves. Both techniques require the normals to be recalculated after manipulation which will be addressed later in the report.

The post processing technique that the application demonstrates is Bloom. The bloom shader is a combination of bright pass filtering and gaussian blurring to focus bright spots and manipulate the relevant pixels. The brief states that the post processing effect must make use of render to texture, which is answered with the use of render textures to isolate bright areas on the screen, down-sample them and display them in the final render with the post processing effect applied.

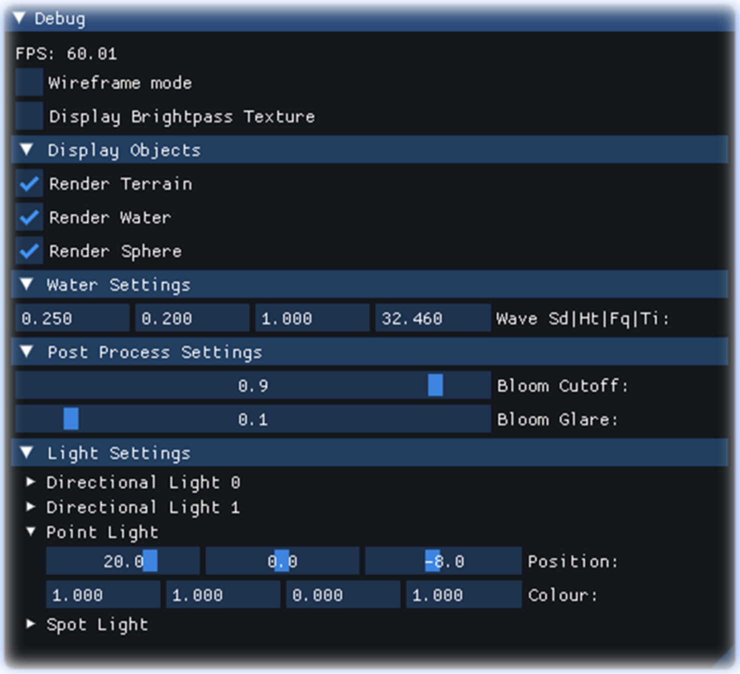
In the application all objects that interact with light have correctly transformed normals and each is involved in the depth pass to calculate shadows. The use of correct diffuse and specular algorithms, and directional, point and spot light types, was also a requirement in the brief and has been addressed in the application.

Tessellation is a prominent part of the application. Both the terrain and the water feature dynamic, distance-based tessellation which is made complex by the usage of vertex manipulation on each object. Both the water and the terrain have tiled textures and are correctly lit and in addition to this the terrain uses a blend of textures to improve the overall appearance.

1.3 Controls & UI

The application has several functions to improve its demonstrative capacity. Present in the UI is controls for choosing which objects to render which can be accessed in the Display Objects drop-down menu, with checkboxes for each object. Below that is a Water Settings menu for controlling the speed, height and frequency of the waves, respectively. The next menu holds controls for altering the sensitivity and intensity of the bloom post process. Finally, there is a drop-down menu with further branches for each light and positional, directional or colour settings for each. The first checkbox in the ImGui menu is for switching to wireframe mode and below that is a checkbox for viewing the bright-pass render texture.

To experience an in-depth demonstration, it is recommended that the user tries altering the Post Processing settings, as well as the light colours and positions. To see the dynamic tessellation in process, switch to wireframe mode and move the camera up and away from the scene.



```
▼ Debug
FPS: 60.01
☐ Wireframe mode
☐ Display Brightpass Texture
▼ Display Objects
✓ Render Terrain
✓ Render Water
✓ Render Sphere
▼ Water Settings
0.250    0.200    1.000    32.460    Wave Sd|Ht|Fq|Ti:
▼ Post Process Settings
                0.9                  Bloom Cutoff:
                0.1                  Bloom Glare:
▼ Light Settings
  ▸ Directional Light 0
  ▸ Directional Light 1
  ▾ Point Light
      20.0      0.0      -8.0        Position:
    1.000    1.000    0.000    1.000    Colour:
  ▸ Spot Light
```

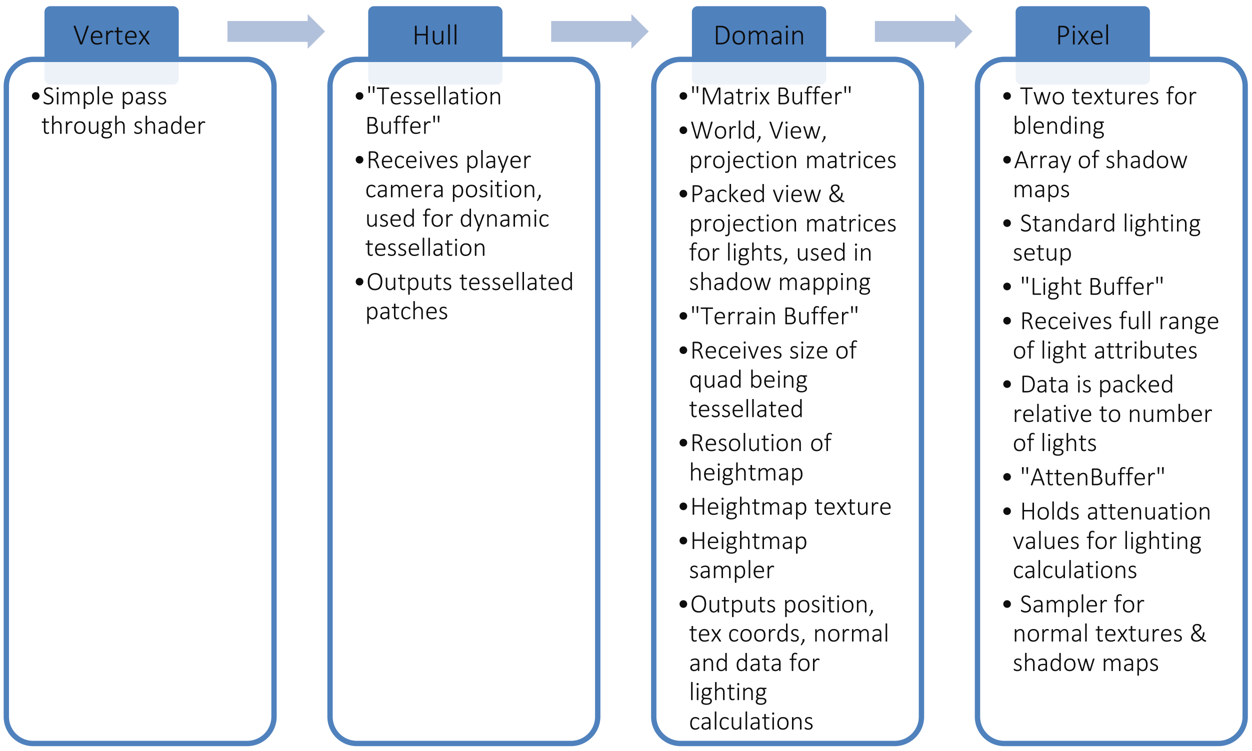
Algorithms & Data Structures

2.1 Data Structure & Transmission

The shader stages that are employed in this application are vertex, hull, domain and pixel shaders. Each shader serves a different function and each receives different data. To store this data the shaders use a number of constant buffers which are set up depending on their requirements.

2.1.1 Terrain

The techniques that the terrain demonstrates requires data to be sent through the vertex, hull, domain and pixel stages. Below is a diagram referencing each stage and the various inputs/outputs of the stage:



Vertex
- Simple pass through shader

Hull
- "Tessellation Buffer"
- Receives player camera position, used for dynamic tessellation
- Outputs tessellated patches

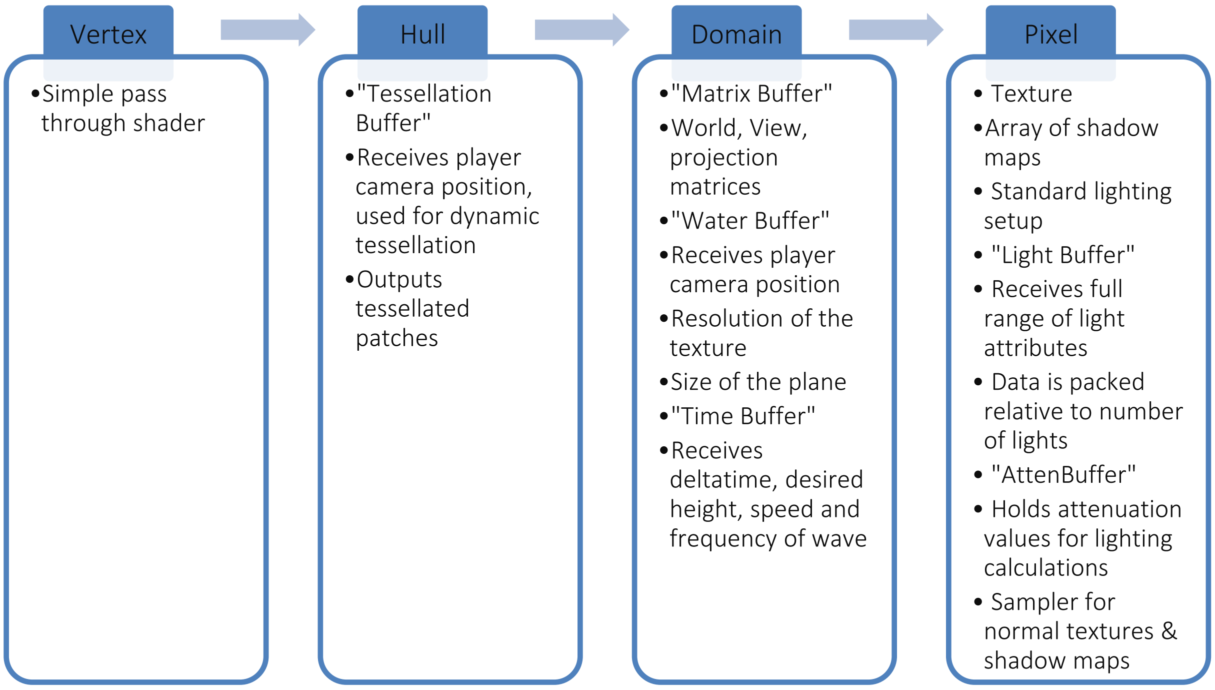
Domain
- "Matrix Buffer"
- World, View, projection matrices
- Packed view & projection matrices for lights, used in shadow mapping
- "Terrain Buffer"
- Receives size of quad being tessellated
- Resolution of heightmap
- Heightmap texture
- Heightmap sampler
- Outputs position, tex coords, normal and data for lighting calculations

Pixel
- Two textures for blending
- Array of shadow maps
- Standard lighting setup
- "Light Buffer"
- Receives full range of light attributes
- Data is packed relative to number of lights
- "AttenBuffer"
- Holds attenuation values for lighting calculations
- Sampler for normal textures & shadow maps

The vertex shader consists of a simple pass through because it is at the domain shader stage that the vertex manipulation takes place after tessellation. The terrain buffer data in the domain shader is used to estimate adjacent vertex normals through a process called central differences, which will be explained in detail later. The same pixel shader is used for all objects however the terrain pixel shader takes an extra texture.

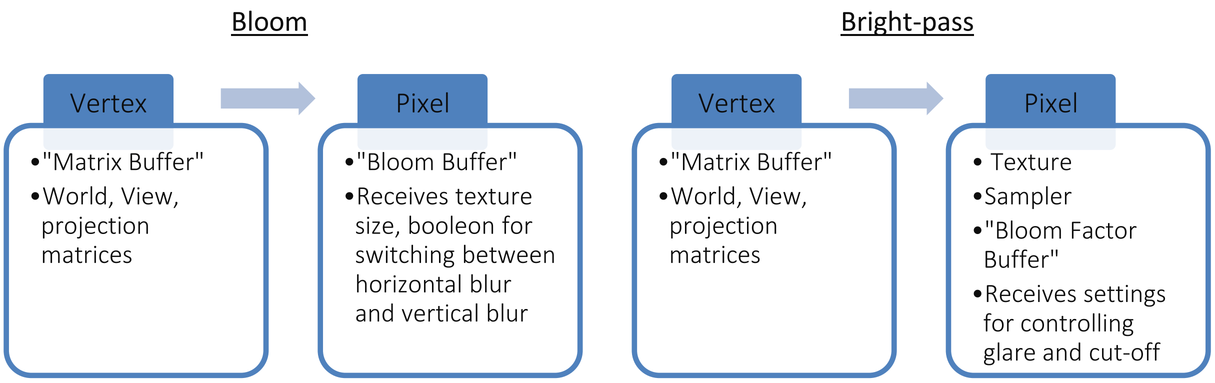
2.1.2 Water

The techniques that the water demonstrates requires data to be sent through the vertex, hull, domain and pixel stages. Inputs/outputs are as follows:


Vertex
•Simple pass through shader
Hull
•"Tessellation Buffer"
•Receives player camera position, used for dynamic tessellation
•Outputs tessellated patches
Domain
•"Matrix Buffer"
•World, View, projection matrices
•"Water Buffer"
•Receives player camera position
•Resolution of the texture
•Size of the plane
•"Time Buffer"
•Receives deltatime, desired height, speed and frequency of wave
Pixel
• Texture
•Array of shadow maps
• Standard lighting setup
• "Light Buffer"
• Receives full range of light attributes
• Data is packed relative to number of lights
• "AttenBuffer"
• Holds attenuation values for lighting calculations
• Sampler for normal textures & shadow maps

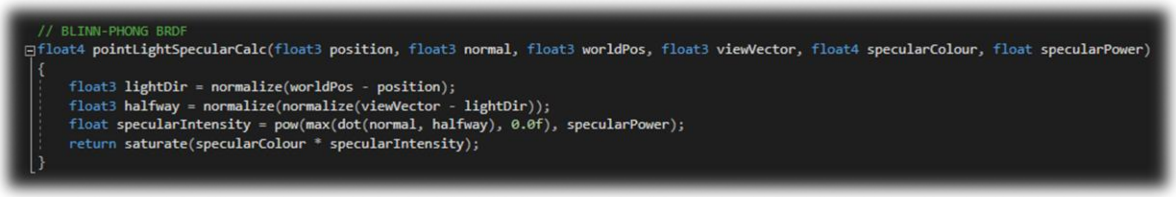
2.1.4 Bloom & Bright-pass

The techniques that the bloom demonstrates requires data to be sent through the vertex and pixel stages. Inputs/outputs are as follows:


Bloom
Vertex
•"Matrix Buffer"
•World, View, projection matrices
Pixel
•"Bloom Buffer"
•Receives texture size, booleon for switching between horizontal blur and vertical blur
Bright-pass
Vertex
•"Matrix Buffer"
•World, View, projection matrices
Pixel
• Texture
•Sampler
•"Bloom Factor Buffer"
•Receives settings for controlling glare and cut-off

2.2 Algorithms & Calculations

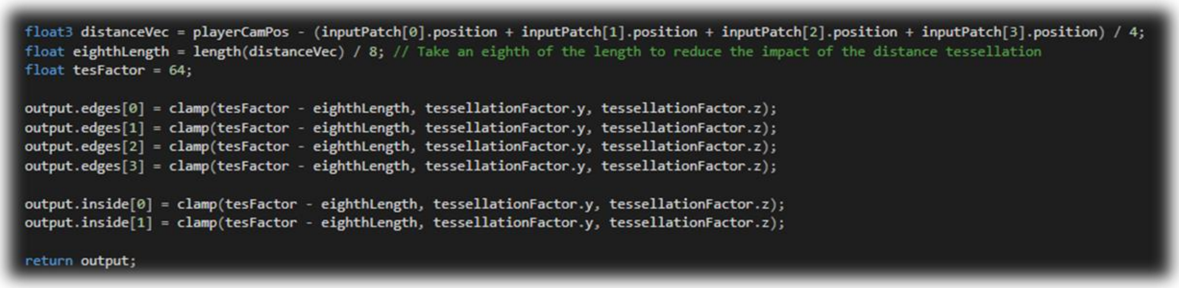
In the lighting shader to calculate specular highlights the following algorithm was utilized, known as Blinn-Phong:



```
// BLINN-PHONG BRDF
float4 pointLightSpecularCalc(float3 position, float3 normal, float3 worldPos, float3 viewVector, float4 specularColour, float specularPower)
{
    float3 lightDir = normalize(worldPos - position);
    float3 halfway = normalize(normalize(viewVector - lightDir));
    float specularIntensity = pow(max(dot(normal, halfway), 0.0f), specularPower);
    return saturate(specularColour * specularIntensity);
}
```

This algorithm takes the view vector from the player camera and the direction vector from a light source to calculate a halfway vector which is in turn used to calculate the theta angle between itself and the normal, using dot product. This gives the angle of reflection for the specular highlight.

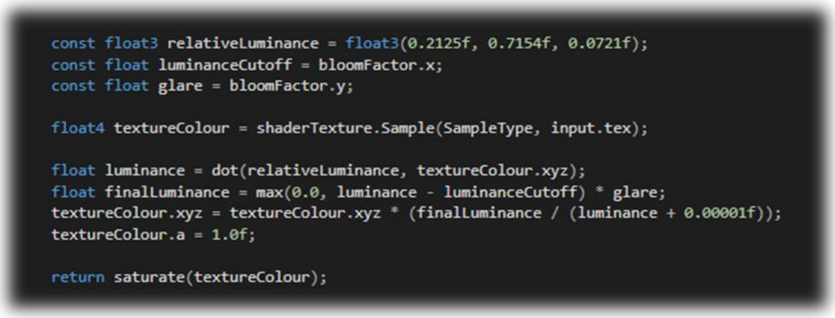
The application features dynamic tessellation which is achieved using this algorithm:



```
float3 distanceVec = playerCamPos - (inputPatch[0].position + inputPatch[1].position + inputPatch[2].position + inputPatch[3].position) / 4;
float eighthLength = length(distanceVec) / 8; // Take an eighth of the length to reduce the impact of the distance tessellation
float tesFactor = 64;

output.edges[0] = clamp(tesFactor - eighthLength, tessellationFactor.y, tessellationFactor.z);
output.edges[1] = clamp(tesFactor - eighthLength, tessellationFactor.y, tessellationFactor.z);
output.edges[2] = clamp(tesFactor - eighthLength, tessellationFactor.y, tessellationFactor.z);
output.edges[3] = clamp(tesFactor - eighthLength, tessellationFactor.y, tessellationFactor.z);

output.inside[0] = clamp(tesFactor - eighthLength, tessellationFactor.y, tessellationFactor.z);
output.inside[1] = clamp(tesFactor - eighthLength, tessellationFactor.y, tessellationFactor.z);

return output;
```

By passing the player camera position in a distance vector can be calculated between the camera and the centre of the input patch. To utilize this vector the magnitude is calculated and when applying the tessellation factor, it is subtracted from the max tessellation factor, 64. This assures that the detail decreases when moving away and increases when moving toward. To reduce the impact of the tessellation the magnitude is reduced by a factor of 8 and this helps disguise the changing geometry. Finally, to keep the tessellation factor within acceptable bounds it is clamped between a min and max value.

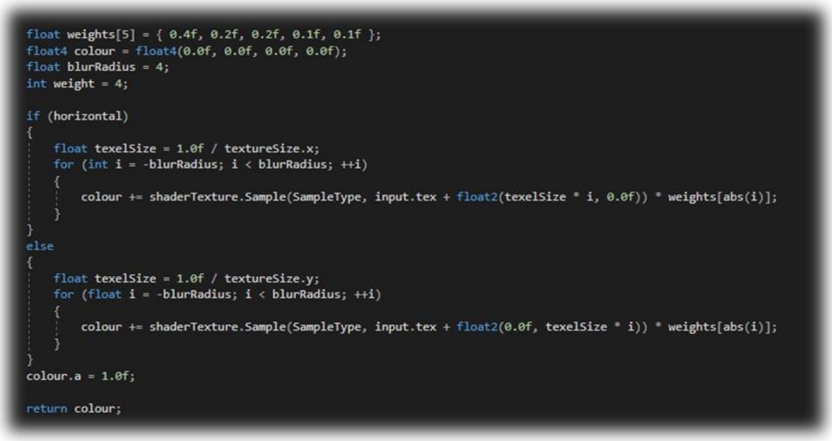
The first step of the bloom post process is isolating the brighter parts of the scene, for which this algorithm was used:



```
const float3 relativeLuminance = float3(0.2125f, 0.7154f, 0.0721f);
const float luminanceCutoff = bloomFactor.x;
const float glare = bloomFactor.y;

float4 textureColour = shaderTexture.Sample(SampleType, input.tex);

float luminance = dot(relativeLuminance, textureColour.xyz);
float finalLuminance = max(0.0, luminance - luminanceCutoff) * glare;
textureColour.xyz = textureColour.xyz * (finalLuminance / (luminance + 0.00001f));
textureColour.a = 1.0f;

return saturate(textureColour);
```

The relative luminance is used as an indicator of the relative brightness of a point in colour space. Along with this is a cut off value for luminance, which can increase the sensitivity of the algorithm, and the glare, which controls the intensity of the detected bright spots.

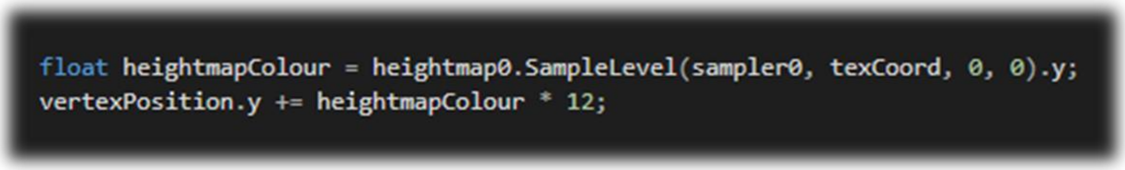
The second step of the bloom is to blur the resulting bright pass texture, using this gaussian blur algorithm:



```
float weights[5] = { 0.4f, 0.2f, 0.2f, 0.1f, 0.1f };
float4 colour = float4(0.0f, 0.0f, 0.0f, 0.0f);
float blurRadius = 4;
int weight = 4;

if (horizontal)
{
    float texelSize = 1.0f / textureSize.x;
    for (int i = -blurRadius; i < blurRadius; ++i)
    {
        colour += shaderTexture.Sample(SampleType, input.tex + float2(texelSize * i, 0.0f)) * weights[abs(i)];
    }
}
else
{
    float texelSize = 1.0f / textureSize.y;
    for (float i = -blurRadius; i < blurRadius; ++i)
    {
        colour += shaderTexture.Sample(SampleType, input.tex + float2(0.0f, texelSize * i)) * weights[abs(i)];
    }
}
colour.a = 1.0f;

return colour;
```

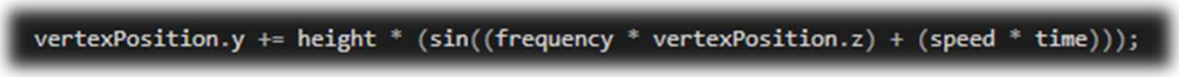
The shader accommodates both the horizontal pass and the vertical pass. Using the passed in texture size to calculate texture space, the pixels are blurred according to the specified weight.

The terrain has its vertices manipulated by sampling the heightmap at the related texture coordinate and, based on the colour value at that sample point, transforming the vertex position.



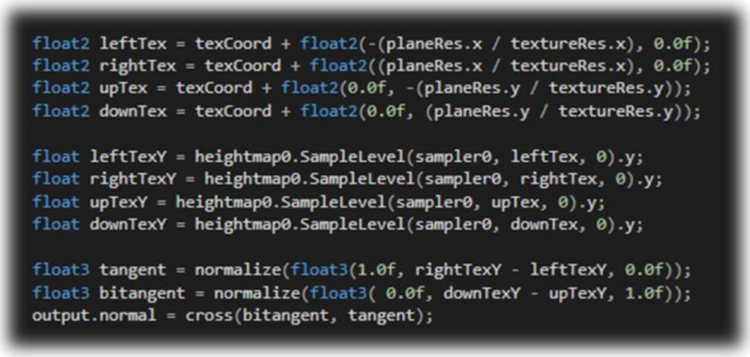
```
float heightmapColour = heightmap0.SampleLevel(sampler0, texCoord, 0, 0).y;
vertexPosition.y += heightmapColour * 12;
```

Similarly, the water vertices positions are manipulated during the domain shader, using this algorithm:



```
vertexPosition.y += height * (sin((frequency * vertexPosition.z) + (speed * time)));
```

The application uses *central differences*[1] to calculate the normals on the terrain after they have been manipulated:



```
float2 leftTex = texCoord + float2(-(planeRes.x / textureRes.x), 0.0f);
float2 rightTex = texCoord + float2((planeRes.x / textureRes.x), 0.0f);
float2 upTex = texCoord + float2(0.0f, -(planeRes.y / textureRes.y));
float2 downTex = texCoord + float2(0.0f, (planeRes.y / textureRes.y));

float leftTexY = heightmap0.SampleLevel(sampler0, leftTex, 0).y;
float rightTexY = heightmap0.SampleLevel(sampler0, rightTex, 0).y;
float upTexY = heightmap0.SampleLevel(sampler0, upTex, 0).y;
float downTexY = heightmap0.SampleLevel(sampler0, downTex, 0).y;

float3 tangent = normalize(float3(1.0f, rightTexY - leftTexY, 0.0f));
float3 bitangent = normalize(float3( 0.0f, downTexY - upTexY, 1.0f));
output.normal = cross(bitangent, tangent);
```

The first part of the algorithm calculates texture coordinates for use in sampling the heightmap, using the input patch UV coordinates as a centre point. Texture space is measured by dividing the plane resolution by the height map texture resolution. Samples are taken at these points to estimate the Y values. The tangent and bitangent are then calculated relative to the heightmap and the normal is the cross product of the two.

Critical Reflection

3.1 Shortcomings

After reviewing my coursework proposal, I have some areas where I have come up short and I would like to address these here. The first and most prominent shortcoming is the lack of any Geometry shader demonstration. I originally planned to have both billboarding and particles in my application but as the project went on I believe this fell to the wayside in favour of other, more urgent aspects of the brief that needed attention. In retrospect I think that I should've made sure to complete the lab before moving on or continuing with my coursework because during this time I could've asked more questions and gained a better understanding of the geometry shader and upon completion of the lab I would have had something that I could present in my application to demonstrate its use. However, if I had thought of it as a requirement rather than an improvement for my coursework I would've prioritized it and made sure it was included. I would do it differently if doing it again, in that regard.

Another proposed feature that I worked on for my submission but never completed was the implementation of Cook-Torrance BRDF. I spent a fair amount of time both researching and coding it and I think I was almost there with it apart from a lighting artifact bug, but in the end I had to let it go because it was using up too much time and I didn't want other areas of my coursework to suffer because of it. I settled for Blinn-Phong and left the Cook-Torrance code in but unused. In terms of other features that I left out, I originally wanted more than one post processing effect but as the deadline drew in I had to narrow the scope to the minimum number of effects.

Lastly, I only had shadows implemented for multiple directional lights. This was something that I failed to consider until it was too late and I unfortunately didn't get to try so point lights and spotlights don't cast shadows.

3.2 Areas for Improvement

In terms of the applications overall graphical quality there is one main area of improvement that effects most of the scene. The tessellated quad that is manipulated into terrain and water starts as a quad with vertices that are 50 units apart and this means that the tessellation is less effective as it can only tessellate to a factor of 64. While this does provide enough vertices to demonstrate several techniques it causes the tessellation to be obvious and in certain areas shadows can misbehave.

For my post process I use a total of 8 render textures and that is not including the scene texture and shadow map textures. Render textures come with their own overheads and while it doesn't have any visible effect in my small application, it will lower performance and that could be noticeable in a large application.

3.3 Possible Solutions

A solution to the low-resolution tessellation would simply be to tessellate a higher resolution quad in the first place and while I was aware of this I wasn't able to implement it in time.

Optimising the usage of render textures by reusing textures that are idle would be an effective solution, as it would reduce the overall number of render textures in the application.

3.4 Extending the Application

My basis on how to extend the application is built upon things that I discovered when researching other topics and things that interested me that I believe would be a valid continuation of the application.

Instead of a static height map the application could be improved to procedurally generate a height map that would be used for vertex manipulation. I have seen several examples of this during the course of the project using noise functions to generate their height map. I often saw the Perlin noise function being used for this and originally, I aimed to use a Perlin noise algorithm for my water simulation instead of the sine wave algorithm. The idea of fully procedurally generated terrain interests me and would definitely be how I take this application further.

A way to make the application more usable would be to implement a material system which would allow me to simply apply a material to a mesh which provides it with all the necessary information about its textures and how to react with light. This would be a cleaner way of giving object attributes rather than the current method of specifying everything each time for each object.

3.5 What I have Learned

I knew from the beginning that this was going to be an extremely interesting and challenging module and I knew that I would have a lot to learn from it. Initially I wasn't sure if I was going to perform adequately in this module because a large amount of graphical programming lies in being able to visualise graphics as both the thing that ends up on the screen and, crucially, what it is before it gets there, which is essentially just numbers, and that is something that I have struggled with. That being said, I feel like I have greatly improved my ability to do so since the start of this module and it has affected how I learn in multiple areas, primarily, but not only, in graphics programming.

Another key aspect of this module that was expected of us was to take a concept that we came up with through research and synthesise that concept into our application. Being given this task meant being able to act independently, but more importantly, it meant needing to be able to justify my choices. The process has changed the way I make decisions with regards to a project, insofar as, where I used to base my decisions for a project on whether I believed I could do it, now, if something I've found interests me, I won't be discouraged to try it if its beyond my level.

Finally, this module has taught me vast amounts about the programmable pipeline and I feel as if I understand it's aspects in depth. Compared to my level of knowledge in the subject at the start of the module I can say with confidence that I have learned, and understand, what I have been taught this year.

References

4.1 Techniques

[1] Luna, F. (2012). Introduction to 3D game programming with DirectX 11. Dulles: Mercury Learning and Information, pp.614, 615.

Kalgirou, C. (2006). *How to do good bloom for HDR rendering | Thoughts Serializer*. [online] Harkal.sylphis3d.com. Available at: http://harkal.sylphis3d.com/2006/05/20/how-to-do-good-bloom-for-hdr-rendering/ [Accessed 1 Nov. 2018].

4.2 Research

Stack Overflow. (n.d.). *Stack Overflow - Where Developers Learn, Share, & Build Careers*. [online] Available at: https://stackoverflow.com/ [Accessed 5 Nov. 2018].

GameDev.net - your game development community. (n.d.). *GameDev.net - Your Game Development Community - GameDev.net*. [online] Available at: https://www.gamedev.net/ [Accessed 5 Nov. 2018].

4.3 Resources

Textures.com. (n.d.). *Textures.com*. [online] Available at: https://www.textures.com/ [Accessed 10 Nov. 2018].