CMP301 Graphics Programming report

Cameron Wiggan

Student number: 2001551

GitHub: Camwig

Overview of application

My graphics application contains three lit models with appropriate shadows, a height map, and manipulated plane mesh with correctly applied normals for lighting and shadows. It also features another plane mesh that linearly interpolates between two height maps to help give the illusion of water which also has calculated normals for lighting and the final mesh in the scene is a simple quad mesh that is dynamically tessellated based on the distance from the camera. The scene has the option to have a post processing depth of field effect applied to it along with options to edit the scenes lighting with graphical user interface elements.

The Initial proposal given for the project was to produce a scene similar to a tropical island. The project I have produced contains many elements of a tropical island. The terrain mesh which texture turns from sand to grass, the water mesh that resembles an ocean and its waves and a velociraptor with the boxing gloves model, all Staple tropical island set pieces.

There are four lighting sources within the scene. One is a spotlight that can have its diffuse colour, direction and position edited, a point light by the water mesh which can also have its diffuse colour edited through the Gui along with its position and the two directional lights which can also have their direction and diffuse colour edited individually also through the Gui. There is the option to enable postprocessing and wireframe mode (depending if the post processing is on or not).

Figure 1 is the scene at the beginning of the application.

Figure 1

Shader Techniques Implementation

For the creation of this scene, I have used a number of techniques associated with shaders and their implementation. Here is a guide to what was techniques were implemented throughout the scene, Following you will find explanations as to how and why they work.

Vertex Manipulation

Vertex Manipulation was used throughout the scene but was most explicitly used for the manipulation of the terrain and water meshes which are manipulated through the use of displacement maps which takes in a height map image which samples that texture from one of the red, green, and blue channels which will represent the height of that part of the image. That height value will then be multiplied by a specified amount which can be changed within the graphical user interface to raise it appropriately as the sampled value will be between zero and one as the RGB channels are clamped between these two values. Which displaces the mesh based of the height map values(Below are some of the height maps I used in the project, figure 2 being the image that was used in week 5 during the labs and the other two being from the internet both from Evo Magz (2018) 'Height map download' at: https://downloadappgratisan.blogspot.com/2017/04/height-map-download.html), was fairly simple in terms of techniques and logic that were learned from this module. This was achieved by passing the terrain shader the height map texture and the amount we wish to multiply the height value by, then passing the texture and the height multiplication value to both a pixel and vertex shader. However, calculating the normals of this newly transformed and displaced mesh had a greater level of complexity.

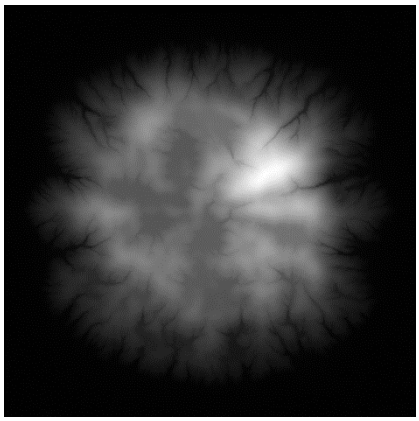

Figure 2

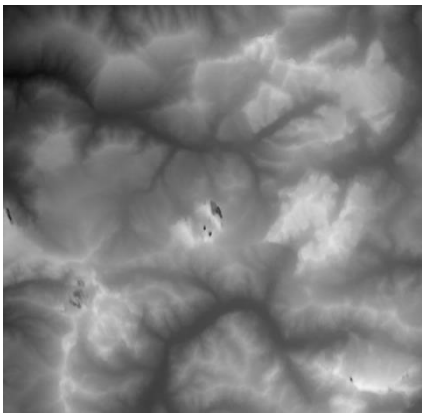

Figure 3

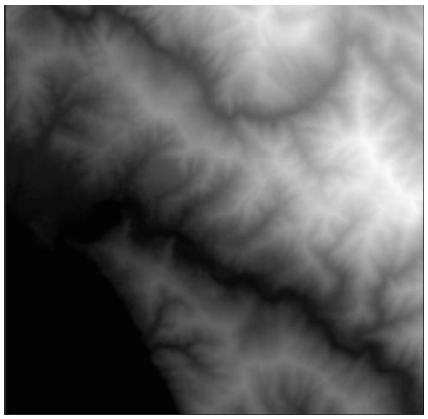

Figure 4

The most effective technique for calculating the normals for this displaced mesh is Normal Mapping (or sometimes referred to as bump mapping).Calculating the normals of the mesh after having the displacement map applied requires the calculation of the height of the cardinal points surrounding the point we are currently performing the calculation on. These points are offset by a certain amount based on the size of the texture. The surrounding points heights are then used to get the tangents direction from the current normal of the current point.(Figure 5 to the right. demonstrates this in a diagram) We then cross multiply the tangents together and divide by the number of tangents for the resultant

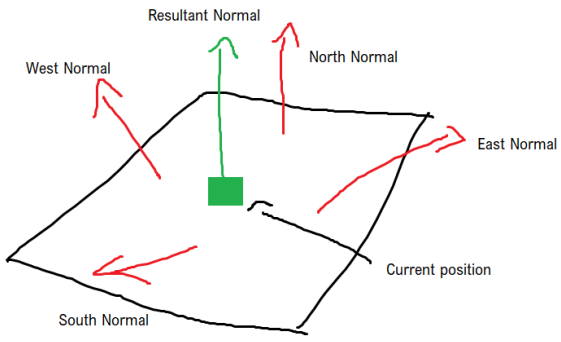

Figure 5

normal of the current point. This is the process of normal mapping. These calculations were all done within pixel/coordinate space within the pixel shader, The reason I didn't go pixel by pixel was due to the fact that the pixels might not always line up exactly with the texture and each texture put into the function is more than likely to be of different dimensions so if we move in pixel space according to the texture we are far more likely to get accurate and better looking results.

1. Get texture dimensions and calculate uv offset
    a. Uv offset is number of points you wish to move in texture space divided by the texture dimensions.
    b. Calculate the world offset which is the uv offset increased scale of its current state of zero to one to the larger scale of the texture.
2. Get the heights of the four cardinal directions
    a. For each of the cardinal directions calculate their current height
        i. To calculate the height, sample the height map texture x (or whatever rgb value that is used to denote height in the image you are sampling) value at the uv plus or minus the offset depending in which direction we wish to calculate.
        ii. Multiply the height by a value from the main application
3. Calculate the tangents of the four cardinal directions
    a. Normalise a new float3 value setting either the y or z value to the negative or positive world offset depending on the direction of the tangent we are calculating. Then set the y value to that of the current direction's height subtracted by the current height of the position we are wanting to calculate the normal of.
4. Cross multiply the tangents for the resultant normal.
    a. Add together the cross multiplied normals of the Intercardinal directions and then divide by four (Number of items) to get the average which will be our new normal

Figure 6

Above is the pseudocode for the normal mapping method (shown in Figure 6) which walks through the process step by step and explains how it was implemented within my project. Below are the results with and without the normal mapping (Figure 8 being the before with the un calculated normals and figure 7 being the results of the normal mapping). As you can clearly see there is quite a difference between the mesh with the non-calculated normals image and the one with the calculated normals. The reason also that the original normals are so flat is because they are setup to be normals for a flat plane mesh.

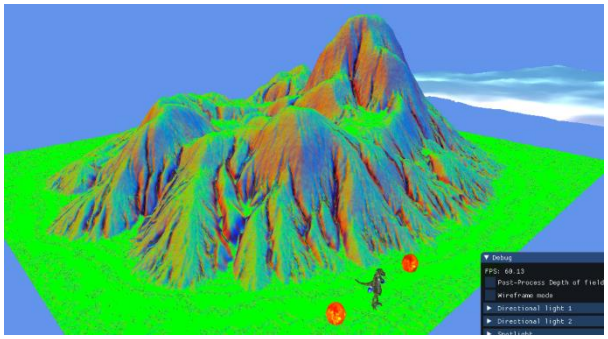

Figure 7

Figure 8

The functions to calculate the height, normals, lighting calculation and specular and attenuation calculations, along with the shadow depth checks and calculation for both the terrain mesh, the water mesh and models were held within a hlsli file which acted as a header file for shaders to retrieve common functions between them and greatly improve efficiency.

This is not the only vertex manipulation demonstrated within the scene. There is also a plane mesh that is setup to move between two different height maps this is an attempt to simulate a large body of water and its many waves. This works in much the same way as the terrain mesh does in that it samples height maps and then multiplies that value by a random amount for each point on the texture The value is between 30 and 1) to give it a more natural organic look as otherwise it would be much more obvious that all it is quite simply moving between two height maps The difference is that instead of sampling a singular texture it also samples two separate height maps and then linearly interpolates between them with the t value being a simple variable that will move back and forth between one and zero throughout the run time. The water meshes normals are also calculated using the normal mapping technique following the same steps as previously stated, however when we get the height of the cardinal directions. We can then calculate the height of these points on both of the two height maps to create the water like movement, we can then quite simply cross multiply the two texture heights for each cardinal directional height we have, including the current height of the point in the texture we are attempting to calculate the normal of. Then we divide by a value of two(Number of heights) to produce the average height of these two values. Which we will then be used to calculate the tangents of the incardinal directions. The results of which can be seen in the diagram above with a point light casting light onto the mesh(Figure 9).

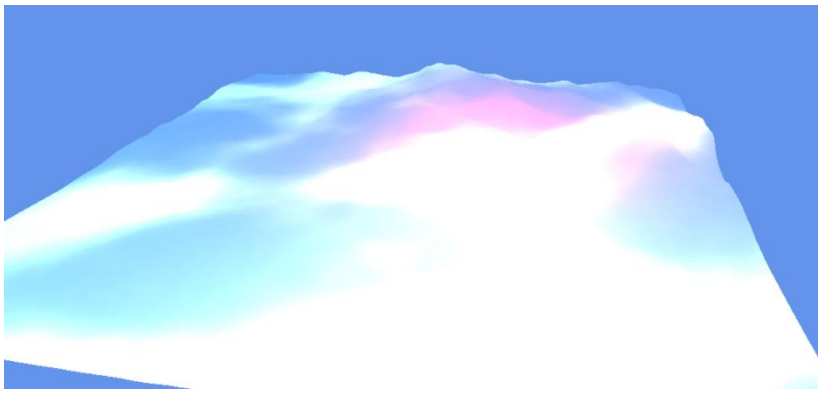

Figure 9

Which brings us to the point in the project where we light the scene. At this juncture, as stated in an earlier part of this report there are four lights within the scene, two directional lights that cast shadows on the terrain mesh and the models, a spot light over the terrain mesh and a point light over the water mesh. Setting up the lighting I found was relatively easy, however when it came to handling multiple lights at the same time within a shader or passing a shader multiple lights the solution that worked best was to instead pass in an array of all the lights within the scene all at once and then set directional, positional and diffuse values etc to their appropriate equivalent from the passed in light array. Another issue found was the idea of having to be able to pass some form of identification in with the lights when they were passed into the pixel shader. Therefore I incorporated a technique that is very common within the game development space which was to use the almost completely , otherwise unused position w value to denote what type of light this light object is and then consequently run the appropriate code according to those findings.

I also incorporated both specular values and attenuation into the scene. For the specular lighting(This helps to simulate local illumination on a point) I used the blinn-phong technique which uses a half vector(The half vector is calculated by normalizing the view vector added to the light direction)which is then used to calculate the intensity by using the dot product for both the normal and the half vector and then multiplying that resultant value to the power of whatever we

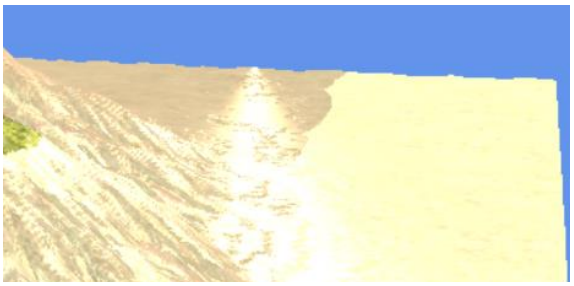

Figure 10

have set the specular power to. This is shown in the image to the right (Figure 10).

The attenuation was used to simulate how light intensity physically weakens as it moves further from the position of the light. Which is calculated by dividing one by the constant factor added to linear factor multiplied by the distance, then added to the quadratic distance multiplied by the squared distance. For these values the light vector length acted as the distance with the constant factor being 0.5, the linear factor being 0.125 and then the quadratic factor being zero which seemed to give the lighting a relatively realistic drop off.

Post-processing

Within the project post processing is demonstrated through the use of a depth of field shader. The effect is achieved by very simply Lerping between a blurred version of the scene and an non-blurred version of the same scene based on the distance of current pixel and a point on the screen which is all in depth. This all happens within the composite shader. However, to get the depth map of the scene a very similar technique to how the shadow maps are calculated was used. Each mesh within the scene had its depth calculated by being passed into the depth shader which quite simply divided the pixel' z depth value by the homogenous w depth value. However, for both the water and terrain mesh since unedited because they are both simple plane meshes they each needed their own depth shader that would multiply the depth value by the height value in the same way each mesh so that an accurate calculation of depth values could be made.

However, before the composite shader is called and executed the scene must first be rendered and then appropriately blurred in order to linearly interpolate between the two versions. This is then rendered to a full screen ortho mesh as is typical with post processing effects. The order that these passes take place are as follows, first the depth pass where the depth map is calculated along with any shadow map that may be needed for the scene, then the first pass is run where the scene is rendered properly and then put into a render texture, next, that render texture is blurred along the horizontal axis in the horizontal blur pass and then put into a texture, that horizontal blur texture is then blurred in the vertical blur and passed along the vertical axis creating a full screen blur which is once again put into vertical blur texture, Finally, the fully blurred texture and the original scene render texture are both passed into the composite shader along with the depth map which will then linearly interpolate between the two textures dependent on the depth value this process will depth of field effect.

The standard scene without the post processing only has two passes the depth shader followed up by a base pass to render the meshes to the scene.

For the blurring in the scene a gaussian blur was used. This was used for both the horizontal and vertical blur functions. What a gaussian blur does is that it takes in the neighbouring colour values and then based on the distance from the centre pixel they are combined with weightings to produce the final colour. For the horizontal blur this happens in the cardinal left and right directions and for the vertical it happens in the cardinal up and down directions. How these pixels are weighted is shown on the diagram to the right (Figure 11).

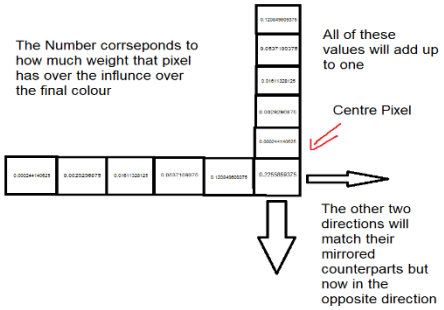

Figure 11

The main logic of this depth of field takes place within the composite pixel shader. It calculates the focal length by sampling the depth texture at a specific point on the screen (Typically this is sampled at the centre of the screen, however I found I personally got better results if I instead sampled at coordinates 0.5,1.0). The next step in the process is to get the distance to the focal plane which is as simple as sampling the depth map texture, these two values will be used to calculate the distance between the camera and its distance from the depth map. It then samples both the unblurred texture and the blurred texture. We then calculate the value that will act as the t value in the upcoming linear interpolation function, this value is the absolute r value of the focal length subtracted by the absolute r value of the distance to the focal plane. Then it simply uses that value as the t value to lerp between the sample blurred and unblurred texture and returns the result.

1. Get the focal length by sampling the depth texture at a specific point on the screen
    a. Sample the texture at 0.5,1.0 as that seemed to produce better results
2. Get the distance to the focal plane by sampling the depth texture
3. Sample both the blurred and unblurred textures
4. Calculate the value that will be used as the t value in the lerp function
    a. Subtract the absolute distance to focal plane r value from the absolute r value of the focal length. We use the absolute value of to avoid using negative numbers.
    b. Multiply the t value by a larger value (I used three hundred)
        i. This is due to how we have to use the projection matrix to have the depth value line up with the camera however it returns much more muted values so for the t value to be big enough to see a large difference we multiply it by a larger value.
    c. Clamp the new t value. As the lerp needs a t value to be clamped between zero and one for it to work correctly.
        i. Checks if the t value is greater than one and if so set it to one
        ii. Checks if the t value is less than zero and if it is set it to zero
5. Lerp between the blurred and unblurred texture according to the t value we calculated
6. Return the lerp value

Figure 12

The pseudocode for this is in the above diagram (Figure 12).

All of this comes together to produce a depth of field effect that blurs the pixels further away from the camera. You can see the results below (Figure 13 and 14).

Figure 13

Figure 14

The shadowmaps used for the creation of the shadows were created in much the same way the depth map was, except the shadow maps were calculated from the light view projection matrix and the light view matrix. Which are then passed to the appropriate shader to create the shadows based of the light that cast them.

Tessellation

Along with the rest of the meshes and models within the scene there is brown coloured quad mesh that is dynamically tessellated based on the distance from it to the position of the camera and the closer the camera is to this mesh the more tessellated it will become. Tessellation is the process of subdividing geometry to improve memory optimisation or to add more detail to a lower poly model or mesh.

Dynamic tessellation is tessellating a mesh based of the distance of that mesh to the camera so the further away a mesh is the lower the tessellation will be and the closer the camera is the higher the tessellation will be. The main inspiration for my implementation comes from the book 'Introduction to 3D game programming with directx11' By Frank D Luna in chapter 13.5.

The way the dynamic tessellation works is that when it gets to the hull shader for the tessellation it will determine the tessellation factors based on its distance from the camera eye this is done by finding the centre of the meshes patch in world space and then finding the distance between that and the current camera position, We will then set a value to act as our near and far values, the near will act as the distance at which we begin to dynamically tesselate the mesh and the far will be the furthest from the mesh the camera eye can be before it will no longer dynamically tesselate it. We then use the formula the near value minus the distance value over the near value minus the far value. This will provide a value that is clamped between zero and one and then all that depending on the distance value we either increase the tessellation value the closer we are to the mesh and decrease it the further away we are from the mesh. This is also demonstrated in pseudocode in the below (Figure 15).

1. Find the centre of the patch in the world space
2. Find the distance between that centre world position and the camera position
3. Define both the near and far values
4. Calculate the distance in relation to the near and far values
    a. Calculate the distance value that will define how the mesh is tessellated from the formula far minus the distance(Centre patch world position from the camera position) divided by the near value subtracted by the far value.
    b. This gets in terms of zero to one the distance value in terms of the far to the near value.
5. Set the tessellation value based on this distance value
    a. Simple if statement to set the tessellation value based on the distance so if it is above or below a certain value it will update the tessellation value.
6. Set the edges and the inside patches to this tessellation value

Figure 15

The results of this is shown below in (Figure 16 and 17). As you can see as the camera gets further away from the quad mesh the less it is tessellated and vice versa for the closer the camera gets.

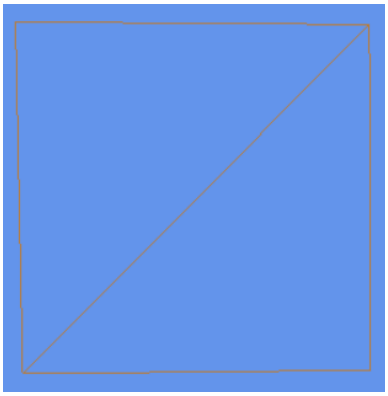

Figure 16

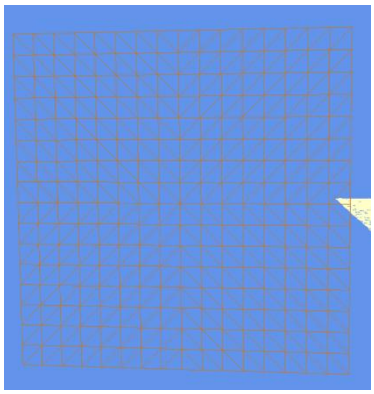

Figure 17

Reflection

This project presented plenty of technical challenges for me as a programmer. I found many things to sink my teeth into, some of which were challenging yet enjoyable and others which I found were incredibly frustrating. For example, the normal mapping was a difficult task to complete and understand but was satisfying in its completion and I felt that the maths as well as the logic came together with ease (In terms of the depth of this project). However, something that is conceptually simple such as the shadows from the terrain mesh took more time and required more effort than I initially anticipated. For future projects I would definitely take forward the idea of not assuming that some parts of the project will be easier or less time consuming than others, don't get me wrong I will continue to carefully plan out the tasks for my work and I will make sure I am certain to have enough time for the tasks that appear initially to be simpler . I have learnt that it may be better to complete them first and then move on as often these tasks tend to be more complicated than it may originally appear.

There are some things in this project I perhaps could have added to or improved on the main one being the tessellation. Which I would like to have been able to add complexity to. such as dynamically tessellating a displacement mesh which I am fairly confident now that I could do with a reasonable amount of time; I am aware that it would be preferable but not necessarily worth a higher mark if I did tesselate a displaced mesh however I wanted to Include it anyway to demonstrate the dynamic tessellation.

References

Height maps in figure 3 and 4, Evo mags (2018)'Height map download' at:
https://downloadappgratisan.blogspot.com/2017/04/height-map-download.html

Frank D. Luna(2012) Introduction to 3D Game Programming with DirectX 11.

Jason Zink, Matt Pettineo and Jack Hoxley (2011) Practical Rendering and Computation with Direct3D 11

Wood texture, SpringySpringo (2020) OpenGameArt.org at :
https://opengameart.org/content/realistic-seamless-wood-texture

Fire Gem model, Models124717 (Game of Origin released in 2002) The Models Resource at :
https://www.models-resource.com/gamecube/starfoxadventures/model/24723/

Alex(Velociraptor), Vinrax (Game of Origin released in 1995) The Models Resource at : https://www.models-resource.com/playstation/tekken2/model/53334/

Learn OpenGL(no date) ‘Depth testing’ at : https://www.scribbr.co.uk/referencing/harvard-website-reference/

Braynzar Soft Tutorials(2015) ‘Normal Mapping(Bump Mapping)’ at : https://www.braynzarsoft.net/viewtutorial/q16390-23-normal-mapping-bump-mapping

Water texture, YCbCr (2022) nGameArt.org at: https://opengameart.org/content/animated-water-texture-128px-0000png