Advanced Graphics – BRANDON STAFFERTON

Contents

[Literature Review 3](#_Toc121584092)

[Camera 3](#_Toc121584093)

[Controls 3](#_Toc121584094)

[Light 3](#_Toc121584095)

[Light Movement 3](#_Toc121584096)

[Directional Light 3](#_Toc121584097)

[Point Light 3](#_Toc121584098)

[Spot Light 3](#_Toc121584099)

[Normal Mapping 3](#_Toc121584100)

[Tangent Space Lighting 3](#_Toc121584101)

[Normal Map 3](#_Toc121584102)

[Parallax Mapping 3](#_Toc121584103)

[Simple Parallax 3](#_Toc121584104)

[Steep Parallax 3](#_Toc121584105)

[Relief Parallax 3](#_Toc121584106)

[Parallax Occlusion 3](#_Toc121584107)

[Parallax Self-Shadowing 3](#_Toc121584108)

[Special Effects Pipeline 3](#_Toc121584109)

[Render To Texture 3](#_Toc121584110)

[Screen Space Tint Effect 3](#_Toc121584111)

[Advanced Techniques 3](#_Toc121584112)

[Gaussian Blur 3](#_Toc121584113)

[Bloom 3](#_Toc121584114)

[Deferred Rendering 3](#_Toc121584115)

[Shadow Mapping 4](#_Toc121584116)

[Bibliography 6](#_Toc121584117)

# Literature Review

## Camera

The camera class I have made is a class I made last year that I have reused and adapted to fit the purpose of this assignment. I have added an orthographic view as well as using a perspective view which will be used later for shadow mapping. I have also setup controls to use WASDEQ to move the camera around and move up and down. I have optimised and modified the calculations for the movement by adding a rotation that is calculated using cosine and sine and is multiplied by a speed variable to create a free moving camera. This is shown in figure 1 and 1.1.

Text

Description automatically generated

Figure 1: Camera calculations, the same is done for up and down, left and right.



Figure 1.1: Updating and setting the potion after the position has been calculated.

## Light

### Light Movement

Light movement is another feature I have added by using ImGui to change the direction and position of the light, this allows you to see the parallax and normal mapping in effect such as seeing the specular move or seeing the normal mapping vanishing out of the light.

### Directional Light

I have also setup a directional light where the position and direction can also be moved using ImGui. The directional light has been setup to use the light’s direction compared to a point light where the vector of vertex to eye is being used. Directional light will mainly be used for shadow mapping as a point light is omni directional and needs more time to set up than I have for this assignment. Also, a cube map of each light’s direction will have to be setup which takes more time than I have.

### Tangent Space Lighting

Tangent space is a space that is local to the surface of a triangle, normals are always pointing upwards in the z direction, I have implemented a way to change my light to tangent space to orientate the normals in the light’s direction.

The way I have implemented tangent space lighting is by making the scene objects have their own vertices and indices making them not shared anymore, and then calculating the tangents and binormals o the cube. Once the calculations of the tangents and binormals are done, they are passed through the pixel and vertex shader by creating a new layout that includes tangents and binormals shown in figure 2. Next, the vertex shader would put the object to world space and then calculated a position for the light by taking away the world position from the light position to create a light vector and I do the same for the camera’s eye position. After I have the object in world space, I then create a TBN matrix which multiplies the tangents, binormals and normals by the object as shown in figure X. With this TBN matrix, I multiply the light and eye vector by the TBN matrix to put the light in to tangent space as shown in figure 2.1. Now I have a variable that is in tangent space, I can send this through the lighting calculations with the normal map to have normal mapping working with tangent space lighting as shown in Figure 2.2.

Text

Description automatically generated

Figure 2: The new layout to accept tangents and binormals.

Text

Description automatically generated

Figure 2.1: TBN Matrix for the tangents, binormals and normals.

Graphical user interface, text

Description automatically generated

Figure 2.2: Multiplying the vector by the TBN matrix.

## Normal Mapping

### Normal Map

Now I have a variable that is in tangent space, I can send this through the lighting calculations with the normal map to have normal mapping working with tangent space lighting as shown in Figure 3 and the normals are now orientating in the light’s direction as well as passing a normal texture through the pixel shader. The result of this is shown in figure 3.1.

Text

Description automatically generated

Figure 3: Passing the eye and light vector in tangent space to the lighting calculations.

A picture containing brick, building material, outdoor

Description automatically generated

Figure 3.1: The result of tangent space lighting with normal mapping.

## Parallax Mapping

### Simple Parallax

Parallax mapping is an enhanced technique of the bump mapping and is done by adding a height map on to the cube with a normal map. You should create a height scale so you can scale down or up height on the height map that is being loaded onto the object.

I have implemented parallax mapping by loading in a height map and then sampling the texture in the shader. I put the camera in to tangent space the same way I did with lighting and then I take the cameras view direction’s x and y position and divide it by the cameras view direction’s z position and multiply it by the texture sample and multiply the height scale. This creates some basic layers and height in to the cube as shown in figure 3.2, however, there are some problems such as the depth being too much that it cuts part of the object out and this is what occlusion mapping fixes using layers.

A picture containing brick, building material, dessert

Description automatically generated

Figure 3.2: Simple parallax mapping

### Parallax Occlusion

Parallax occlusion is an improved technique of the simple parallax because layers are involved to start creating depth within the texture. Using these layers, we can use a height scale to determine how many layers the texture has.

The way I have implemented this is by using the previous simple parallax, however, I am creating a minimum and maximum layer amount and sampling a texture by checking if the currently layer height is less than the sample size of the height map. Since I set the current layer height to 0, I will add layers to the height map until I’ve hit the maximum number of layers. I then create a before and after height to get the height map before I added layers and after I have added the layers to get a weight, this weight is then added to the texture coordinates of the cube which determines how deep the height map layers will go, the result of this is shown in figure 3.3.

A picture containing container, box, brick

Description automatically generated

Figure 3.3: The result of using layers for parallax mapping, also known as occlusion mapping.

### Parallax Self-Shadowing

Parallax self-shadowing is also done is a similar way to the parallax mapping. This is done by sampling a current depth value of the height map as well as a current depth layer. With these variables, I then create a while loop going through the maximum number of layers and within the loop, I subtract the current layer depth from the number of layers. Finally, I check if the current layer depth is more than the depth value and if it is, we can sample and draw self-shadows to the height map texture, this is shown in figure 3.4. The result of this is shown in figure 3.5.

Text

Description automatically generated

Figure 3.4: Creating self-shadows using layer depth.

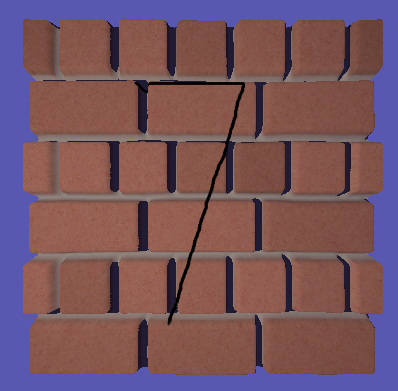


Figure 3.5: Self-shadowing result.

## Special Effects Pipeline

### Render To Texture

Render to texture is a technique I have done to create different screen space effects; I render the scene as normal, but I render it through a screen quad but as a texture which allows me to reuse it.

I have implemented this technique by doing two render passes and creating another render target view, shader resource and texture which is our render to target variables. Firstly, I set the render to texture target view and then render the scene and scene objects as I normally would for the first render pass. For the second render pass, I then change the render target view to the original target view, but I set the shader resource view to the render to target texture. Also, in this second pass I have made a screen quad that is made with a new vertex and indices layout. Using this layout, I have passed the position and texture coordinates of the quad into a new shader and this where I will be creating screen space effects. This results in a quad as the screen to which I can pass a texture through and use it multiple times.

### Screen Space Tint Effect

Now that I have passed a texture through a quad using the render to target shader resource, I can create functions to make a variety of different screen space effects. The basic one I have made here to test to see if it works is that I have tinted the texture to a red colour, this is shown in figure 4. I have done this by sampling the texture that has been passed through the quad, The result is shown in figure 4.1.

Text

Description automatically generated

Figure 4: Sampling and tinting the texture passed into the quad

A picture containing shape

Description automatically generated

Figure 4.1: Result of tinted texture, makes the cube red but the background is orange because it’s added the red colour tint to the bright green background.

## Advanced Techniques

### Gaussian Blur

Using the render to target technique to create screen space effects, the first advanced technique I have done is gaussian blur. Gaussian blur is a type of blur where you sample a small number of weights to assign to a vertical and horizontal blur, this is then sampled over an offset of an exact size of a single pixel of the texture, or known as a texel.

I have implemented this by creating a function that takes in the texture coordinates of the texture that is passed through the quad. I then create five different weights of type floats to use these to sample each pixel. I then create a texture offset by dividing one by the texture coordinates. Next, for the horizontal and vertical blur, I sample the texture again, however I add the texture offset to the texture coordinates and multiply it by the weights in both the x and y direction of the pixels as shown in figure 4.2. I then return the value of the blurred pixels out the shader and the result is shown in figure 4.3.

Text

Description automatically generated

Figure 4.2: Blurring in the horizontal and vertical direction of the pixel.

A picture containing background pattern

Description automatically generated

Figure 4.3: Result of blurring horizontal and vertical.

### Bloom

Bloom is the second advanced technique I have implemented using the quad for screen space effects. Bloom uses gaussian blur and combines it with the rgb values of the scene and adding a brightness and gamma level to change how effective the bloom will be.

I have implemented bloom by adding the gaussian blur I previously made to a texture that has sampled the texture coordinates of the quad. I then use tone mapping which uses a brightness value that is multiplied by the scene colour and that value is then used to get the exponential brightness value. Lastly, I then create a gamma value for gamma correction for the bloom effect by doing the power of the one divided by the gamma value, this implementation is shown in figure 4.4 and the result is shown in figure 4.5.

Text

Description automatically generated

Figure 4.4: The bloom function which adds the scene colour and gaussian blur together and then passed through the full screen quad.

A picture containing chart

Description automatically generated

Figure 4.5: Bloom effect using gaussian blur. Brightness and gamma have been put up to higher values for clearer visualisation.

### Deferred Rendering

Deferred rendering is a screen space technique to remove forward rendering (learnopengl, LearnOpenGl, 2020). Using deferred rendering splits, the lighting calculations, and materials properties into two separate shaders. The material properties shader creates a GBuffer which contains the normals, diffuse, specular, position ambient and emissive textures and pixels. The lighting shader then uses the GBuffer, and lighting calculations combined to generate a rendering pass and results in the lighting only computing for the pixels that are visible and gives a better scalability to the number of light sources.

I have implemented this technique by creating six render target views, shader resources, textures and then a single lighting target view, shader resource and texture to pass through a screen quad s shown in figure 5. Now that six textures have been created, we can clear the render target views before doing the first render pass of the GBuffer. Once the render target views of the six textures have been cleared, we set the render target to the GBuffer textures, the first render pass then renders all the game objects by using the material shader which only does material calculations for normal mapping and height mapping. I then do a second render pass by setting the render target to the lighting view. This second render creates a blend state which combines the material shader and the lighting. Following the blend state, I then set the lighting vertex and pixel shader and set the shader resource to the GBuffer shader resource view and draw this to the quad which creates six textures as mentioned above is shown in figure 5.1. Lastly, I do a final pass that passes the lighting resource view to the quad by setting the render target to the scene and then setting the vertex and pixel shader for the quad. This results in the use of forward rendering being removed and now our pixels have been split up into six different textures so we can determine what pixels the light hits.

Text

Description automatically generated

Figure 5: Creating the six render target views, shader resources and textures.

Diagram

Description automatically generated

Figure 5.1: The result of creating a GBuffer and rendering all six target textures.

### Shadow Mapping

Shadow mapping is the use of a directional light that uses an orthographic projection and the use of a matrix that uses a lights position and direction to create a depth map. By using this depth map we then can create shadows based on the scenes objects positions (learnopengl, LearnOpenGl, 2014).

I have implemented this by extending my deferred rendering by creating another stencil view for shadows which will be used to create a depth texture; therefore, I made another shader resource for shadows as well. Shadow mapping will require four different passes, before the first I cleared all the render targets and set the depth stencil to the shadow stencil. The first pass is creating the shadow/depth texture. Firstly, I created a light matrix that stores the position and direction of the light and then create an orthographic projection and then multiply these together to get a projection matrix based on the light which is then passed through the directional light properties buffer as shown in figure 5.2. I then render all the game objects using this light matrix to create a depth texture.

Next, for the next three passes, I then did the same as my deferred rendering as previously shown by setting up the GBuffer, lighting pass and quad pass, however, within the lighting pass I change the vertex and pixel shader to use a different shader for shadows which is like the previous one but now includes shadow calculations and sets another shader resource to the shadows shader resource as shown in figure 5.3.

The shadow calculations are done in the shader by firstly creating a light space matrix which multiplies the position of the pixels in world space and the light matrix previously made. I then sample the depth texture and use this as a current depth to compare to the closest depth to determine which depth to be drawn on the objects as shown in figure 5.4. Finally, I multiple the shadow calculation by the final light calculations to get the results shown in figure 5.5.

Text

Description automatically generated

Figure 5.2: Calculating light matrix.

Text

Description automatically generated

Figure 5.3: Setting a depth texture.

Text

Description automatically generatedA cube on a black and white checkered surface

Description automatically generated with medium confidence

Figure 5.4: Shadow Calculations. Figure 5.5: Result.

# Bibliography

learnopengl. (2014). *LearnOpenGl*. Retrieved from LearnOpenGl - Shadow Mapping: https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping

learnopengl. (2020). *LearnOpenGl*. Retrieved from LearnOpenGl - Deferred Shading: https://learnopengl.com/Advanced-Lighting/Deferred-Shading