Shaders Report-p2529627

The first thing that I implemented into the base project was directional lighting. I did this using a single vertex and fragment shader for both the cubes and the plane. This was done by uploading values to the fragment shader for the object colour, light direction, light colour, ambient factor, and specular strength. The fragment shader would also take in the position of the cubes through the vertex shader as well as the normal.

The directional lighting was calculated in 3 parts, which were then added together to give the overall colour, the first component was calculated by multiplying the light colour by the object colour and the ambient factor.

The diffuse factor, which is the reflected light off objects such as the cubes in the scene were done using a similar method however, the diffuse factor is calculated by taking the dot product of the normal of the models and the negative light direction which then gets limited to 0. The final calculation for the diffuse component becomes light colour \* by object colour and diffuse factor.

The specular factor which is how the light is perceived by the eye(or camera in the scenes case). This is calculated by working out the vector of the reflected light for which I used the reflect() function taking in the light direction and the normal. The reflected vector is then dot \* by the view direction to calculate the specular factor. The specular factor is then limited to 0, and then raised to the power of the shine. The specular component of directional lighting becomes the light colour \* object colour \* specular factor \* specular strength.

To switch from the phong lighting model to Blinn-phong, I changed the way that the specular factor was calculated so that instead of using the reflect function, we would be taking a vector at the halfway point between the light direction and the view direction and working out the dot product of the halfway point and the normal.

The next implementation I made was the use of textures, the textures were done by assigning unsigned int values to variables which are then bound to textures which are loaded through stb\_image. From this point they are bound to the shader using openGL’s active texture and the pass the integers that the texture is assigned to into the shader. The lighting calculation then changes as the diffuse colour in the lighting was changed to sample the texture that we passed in using a uniform sampler2D with the texture. Having textured both the cube and the floor, I proceeded to add the specular map in the same way that I did with the diffuse map, but only altering the specular colour to match the x component of the sample taken from the specular map.

The first light caster implemented was the point light, to implement the point light I passed in struct values into the shader using setInt, these variables would then be put into a struct called pointlight in the fragment shader of both the floor and the cube shaders, I then implemented a function to calculate the result of the pointlights which took in a vec3 for normals, a vec3 for view direction and a vec3 for fragment position. The variables passed into the point light were as follows:

* Vec3 position
* Vec3 ambient colour
* Vec3 diffuse colour
* Vec3 specular colour
* Float constant constant
* Float linear Constant
* Float Quadratic (or exponential) constant

The point light function would first sample the diffuse and specular maps for the texture of the shader as vec3’s for use later when determining the colour. Following on from this, the distance is calculated by finding the distance between the position of the pointlight and the fragment position. Then the attenuation is calculated using the following formula

A picture containing diagram

Description automatically generated

[1]

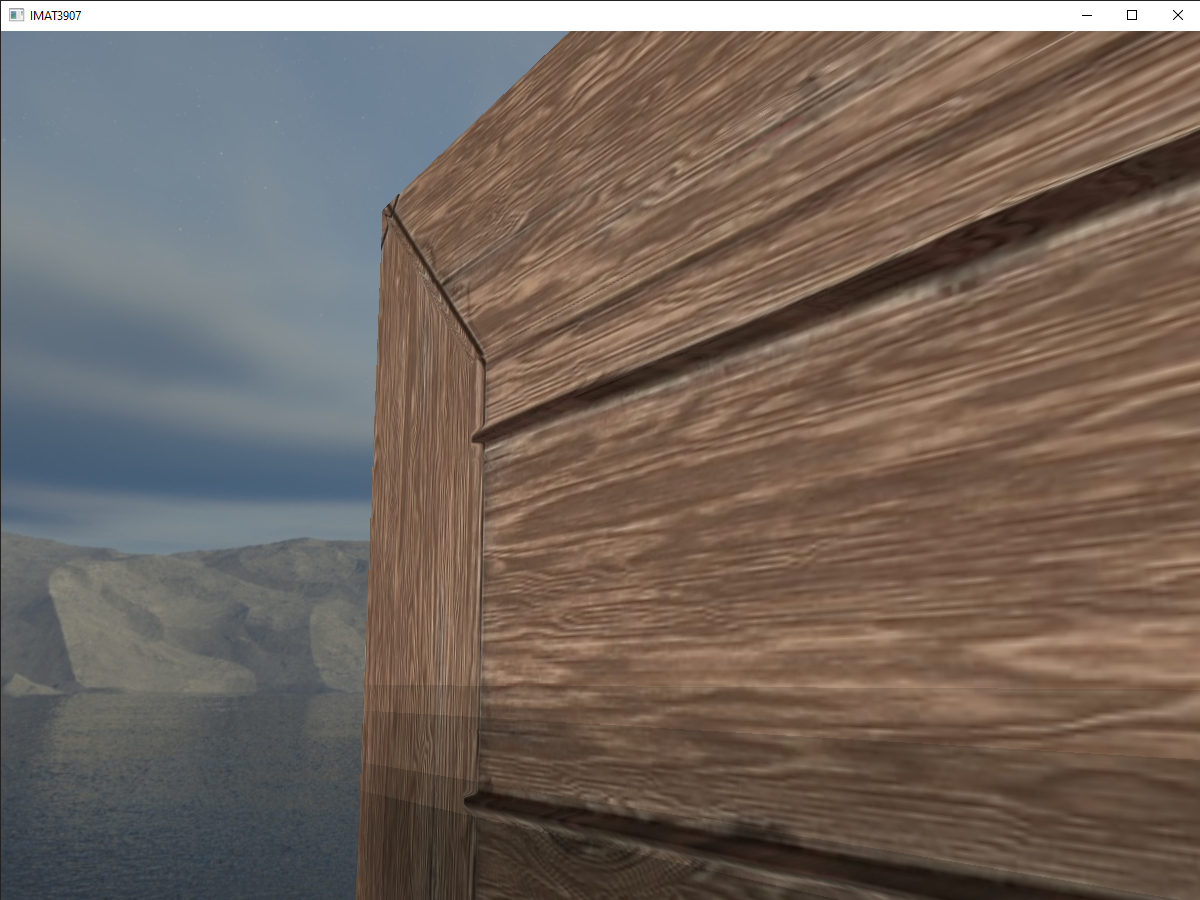
The implementation of point lights was like the way in which the directional light is calculated but factoring in attenuation by multiplying the resultant lighting components by the attenuation value calculated in the function.

After implementing point lights, the final light caster implemented is spotlights which are implemented in a mostly similar way apart from the fact that the values are clamped by an inner and outer radius from a certain point, the main difference however is the illumination factor which is calculated using the angles of the spotlight.

Following on from light casters, I implemented normal mapping, which required the introduction of new values into the array. I went about doing this by getting the vertices from the initial array, adding 2 vec3’s for the tangent and binormal, use the indices to assign the correct edges for the calculations for the tangent and binormal, to then calculate it and then return it to a vector which can then be called for and hence assigned to a vertex attribute array, so that it can be passed into the vertex shader, once the values have been passed into the vertex shader, they have been sorted into their own vec3’s which are put together and transposed into a TBN matrix which is used to convert the fragment position, light direction and view direction all into tangent space, those variables are then passed into the fragment shader, and used to calculate directional lighting in tangent space.

Having finished with normal mapping the next change made to my project was introducing parallax mapping, which is supposed to give the illusion that a 2d plane looks as though it is in 3d when it is a 2d texture that has had its uv co-ordinates moved to give that illusion. The initial implementation used a displacement map which gets sampled and returns the uv coordinates adjusted for the view direction according to the displacement map, the function for this takes in uv co-ordinates and the view direction as both are required to return a new uv coordinate in this way.

Which gave this result:



I improved upon this basic parallax mapping by implementing steep parallax mapping, which is an improvement upon parallax mapping due to the improved accuracy with which it gives new uv co-ordinates, this is because it uses a layer system to determine how much the uv co-ordinates need to move by and compare the layer depth to the red channel value returned by the texture to more accurately give uv coordinates based upon the cameras position.

The implementation of steep parallax mapping gave the result

A picture containing wooden, wood

Description automatically generated

Following on from the steep parallax mapping, I introduced linear interpolation between the co-ordinates and returned the uv co-ordinates of the interpolated value.

Steep parallax with occlusion mapping.

A picture containing wooden, wood

Description automatically generated

With ambient occlusion mapping done, I decided to move framebuffers. The first implemented framebuffer was the colour fbo, which uses a colour attachment , the colour attachment is an empty texture which is bound to the framebuffer the colour fbo also contains a render buffer object which used as render targets. This FBO is primarily used to apply colouring effects as the scene drawn is rendered to a quad which is drawn using a shader specifically for post processing, within the post processing fragment shader by changing the fragment colour output of the shader. A point of note in the generation of the texture is that the channel of the texture is GL\_RGBA16F rather than GL\_RGB, this is because GL\_RGB16AF allows for 16bit rbg rather than the default 8, meaning that it gives us high dynamic range(HDR).

Another framebuffer that was implemented was the depth buffer, which is somewhat similar to the colour fbo, however where I’d put gl\_RGBA16F, I now put GL\_DEPTH\_COMPONENT as required when creating a depth framebuffer object, a point of note with the depth FBO is that the colour attachment is explicitly set to none so that the framebuffer runs as openGL requires some knowledge of the state of the colour attachment. The vertex shader for this is the same as the colour attachment and the fragment shader takes a texture of the depth map created by the scene rendered in the FBO.

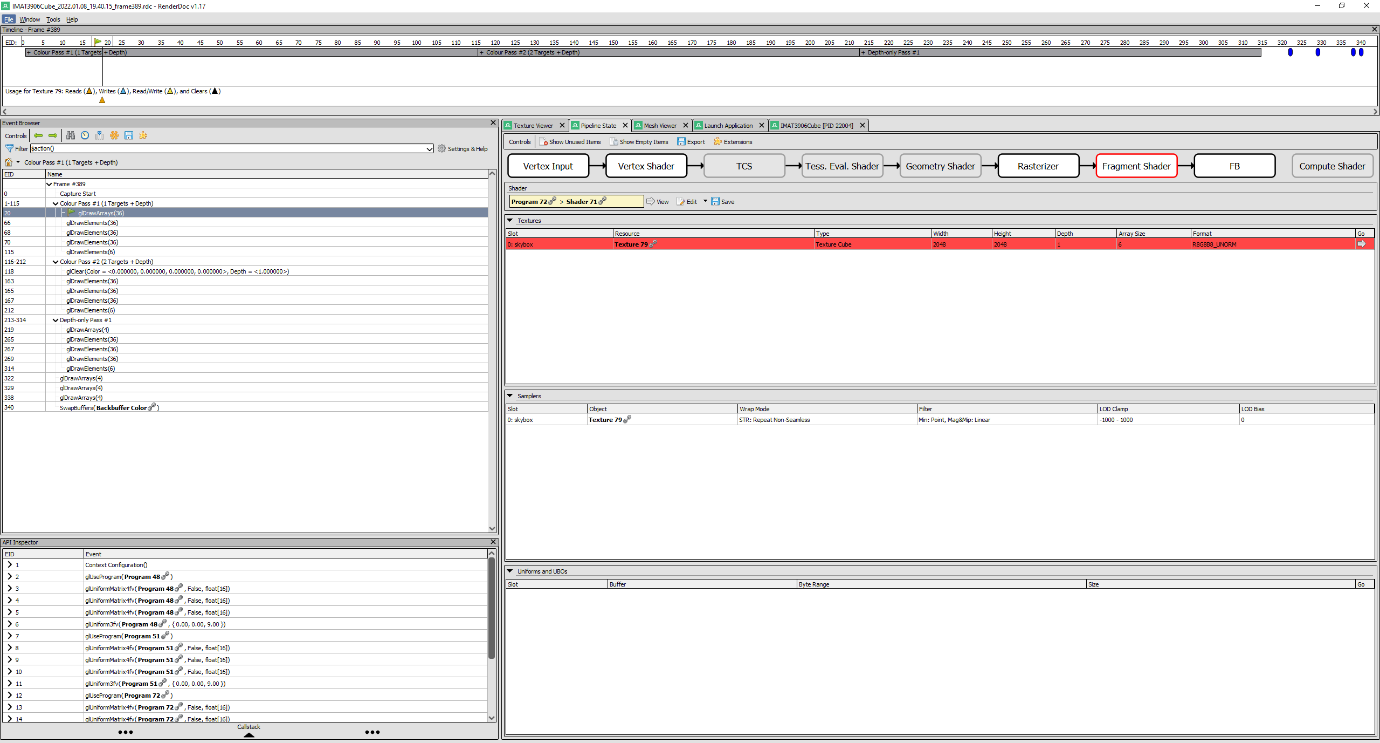
The next framebuffer is that of the colour and depth, this is notable for using multi target rendering as it consists of 2 colour attachments which is beneficial as it allows me to render multiple instances of the scene and apply different effects to each, this will be useful when implementing blurring and bloom in later framebuffers.

The next implementation is of blurring , the FBO for blurring was set in the same way a colour FBO would be made. When the blurring framebuffer object is bound in the main loop, a Boolean value is passed in as true to the shader and the quad is drawn using the blur shader onto the 2nd colour attachment, the bool is then set to false and is drawn onto a new texture called blurredTex, the blur shader has an array of 22 values in a gaussian distribution which represent the weighting of the texels as they iterate through the for loop to determine the new uv of the texel, which is then returned.

This blurring feature is then put in conjunction with a change made in the fragment shaders as they now output bright values in the scene using layout(location), this is so that they can be passed into a new shader for bloom. Which samples the high colour image and blur map so that only the high coloured texels are blurred.

The final framebuffer made was for shadows for which I made an FBO like that of the depth FBO with the exception being that the texture size was limited to a square 1024,1024 due to the shadow map needing to be a square. Another thing that needed to be done for shadows was the uploading of a light-space matrix, this is some form of issue as a light space matrix requires a light view and light projection which is problematic due to the scene being perspective, and shadows requiring the specific view of the light source (making it need orthographic), this can be done using glm::ortho and the light view was done using glm::lookat. Light view also needed a light position, which I calculated by multiplying the light direction by -1(so that it was looking down at the origin) the shadow map vertex shader returns a position of the light-space matrix \*model\*vec4(position,1.0). The light space matrix also needed to be passed into the cube and floor shader to get the light space fragment position to work out the shadows, as the shadows are calculated in light space., this was attained by multiplying the light-space matrix by the world space position, this was passed into a function to calculate the shadows, which were passed into the directional light changing its result to ambient +(1-shadow)\*(diffuse + specular), this gave shadows, that were generally low quality however, so I implemented a similar technique to what was done for blurring but used an embedded for loop rather than only taking values form the vertical and horizontal texels, this heavily improved the quality of the shadows

The final thing that I implemented to my project was my skybox, which was done using a cubemap, which is loaded through a for loop what iterates through the faces, with this however, I ran into an issue in that the faces were loading, but they weren’t rendering. I ran the program in render doc to figure out what the issue was



Using renderDoc, I found that the issue was that I mistakenly put GL\_TEXTURE\_2D when setting the filters for the cube map rather than GL\_TEXTURE\_CUBE\_MAP, which once corrected, loaded my cubemap. The cubemap vertex shader takes in the position which it passes on as UV co-ordinates so that the fragment shader can draw the skybox. This drew the cubemap but still allowed the camera to move through it, to fix this I set the view uniform in the skybox shader to be the view matrix of the camera so that it moved with the camera. This concludes my IMAT 3906 report.

References

[1] <https://ogldev.org/www/tutorial20/tutorial20.html>-point light tutorial by openGL dev