Shaders Document

A User Document for:

CMP301

Programming with Shaders Semester 1.

Abertay University

Written by Liam MacLean 22/11/2017

Table of Contents:

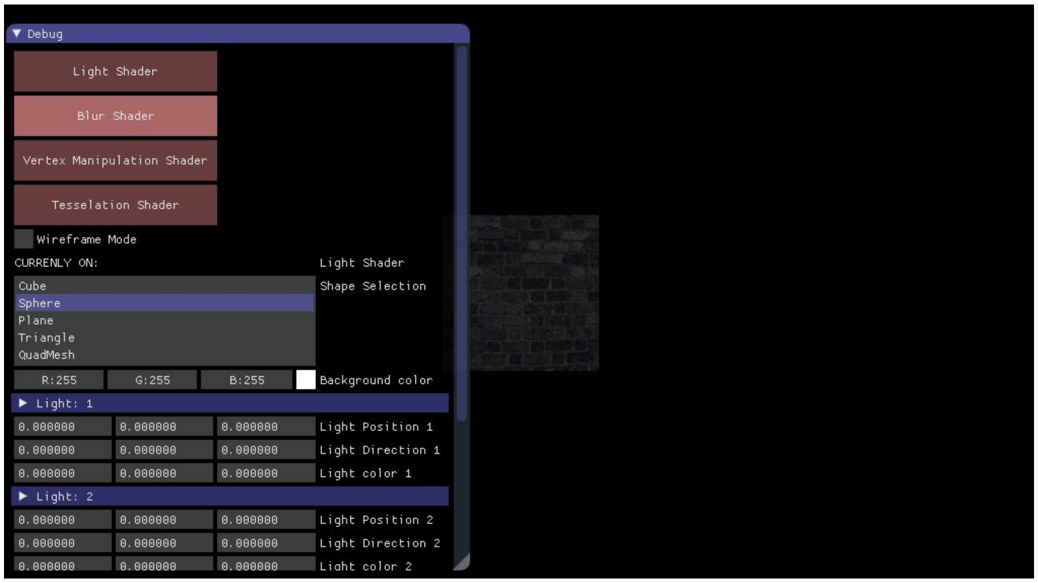
1 Overview

My proposal for this module, was to create a multi shader program, that shows off as many shader types as possible. In the program there are multiple buttons to change the different type of shaders being shown using the ImGui library. The different type of shaders being shown off are:

- **Directional Light Shader** - Multiple directional lights on a single object with editable values.
- **Spot light Shader -** Directional-conal spot light shader with a target position the cone is centralised on with editable values.
- **Vertex Manipulation Shader** - Manipulating an object using sine and cosine waves with a vertex shader.
- **Gaussian Blur Shader** - A blur shader using the gaussian blur method with several render to texture passes. Made from breakdown in CMP301 Lecture 06.
- **Tessellation Shader** - A shader that subdivides and creates new vertexes using a domain and hull shader.
- **Cel Shader -** A shader that lights an object in a particular style and outlines the object with a colored outline (Borderlands-esque)
- **Shadow Shader -** A shadow shader using depth testing and shadow mapping from lecture CMP301 lecture 10.

1.1 Controls

The controls for each of the shaders are consistent on every mode. The controls for the camera are identical to the ones in the example projects in the class. These are: WASD or Arrow keys in order to move, space to rotate and center the camera's eye at the mouse cursor, and E and Q to move the camera up and down the Y axis respectively.



Wireframe Mode

CURRENLY ON: Light Shader

Shape Selection: Sphere

R:255 G:255 B:255 Background color

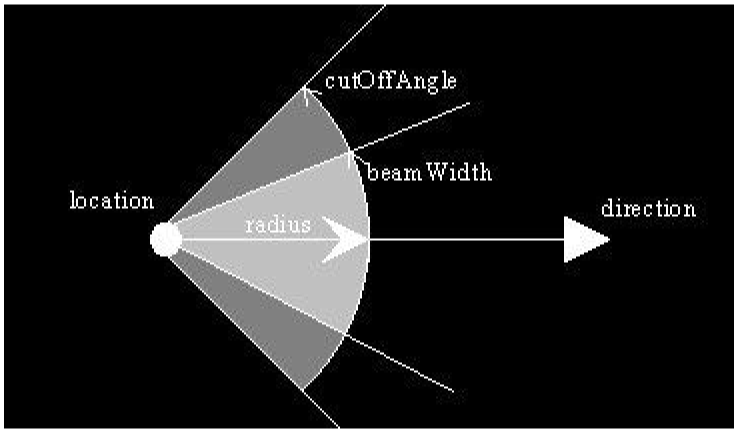
0.000000 0.000000 0.000000 Light Position 1
0.000000 0.000000 0.000000 Light Direction 1
0.000000 0.000000 0.000000 Light color 1

0.000000 0.000000 0.000000 Light Position 2
0.000000 0.000000 0.000000 Light Direction 2
0.000000 0.000000 0.000000 Light color 2

The UI for the ImGUI is relatively self explanatory. You have the option to choose between the different shaders, and multiple other options alongside that. Options will change dependant on what shader is open at the time. Wireframe mode can be toggled and background color can be changed from any shader.

2.0 Spot Light Shader

The spot light shader is a light that can be setup to point light in a specified direction within a conal radius of light. This shader has attenuation factors, a range factor and height attenuation.



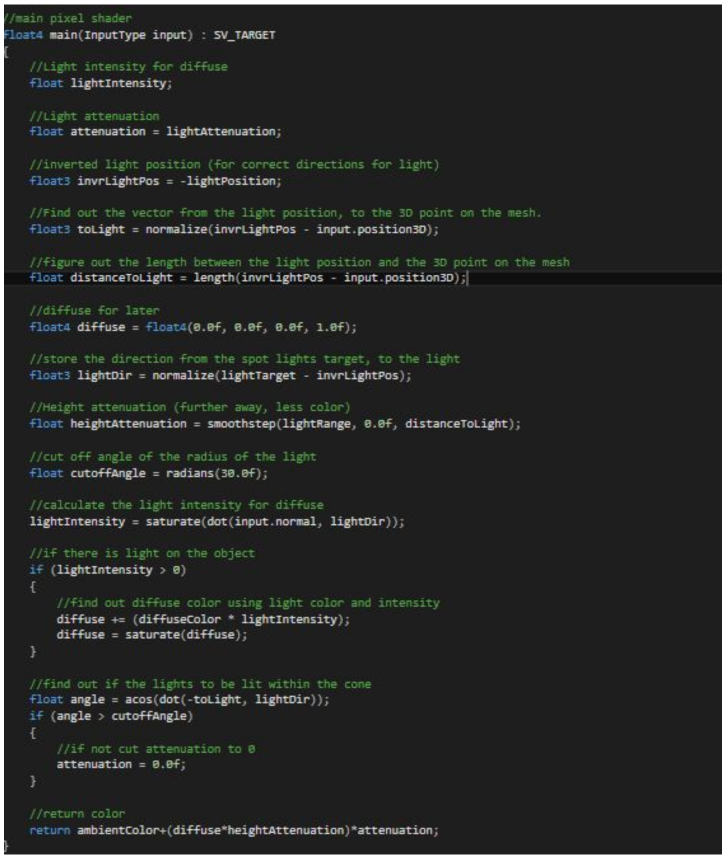
Resources:
Tom Dalling Blog on OpenGL Lighting[3]

Shaders used:
The shaders used for this example are a two sets of pixel and vertex shaders, and a geometry shader. The first set deal with the light and color of the object directly using a pixel shader and vertex shader. The second set of shaders use a geometry shader with a vertex and pixel shader. These shaders are for drawing the directional widget for the direction of the spotlight onto the screen in world space.

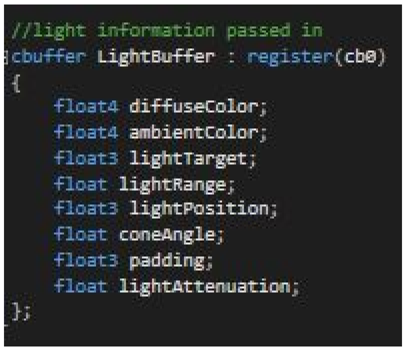
Objects/Tools used:
The example uses a BaseMesh with polymorphism to allow the ability to change the mesh at will. There is a PointMesh for the directional widget that is used with the geometry shader to plot the directional vector on the screen from the point light.

The example also uses a SphereMesh to simulate the point in space where the light exists. This light has it's own world matrix in order to manipulate its position in world space independently.

How it works:



```
//main pixel shader
float4 main(InputType input) : SV_TARGET
{
    //Light intensity for diffuse
    float lightIntensity;

    //Light attenuation
    float attenuation = lightAttenuation;

    //inverted light position (for correct directions for light)
    float3 invrLightPos = -lightPosition;

    //Find out the vector from the light position, to the 3D point on the mesh.
    float3 toLight = normalize(invrLightPos - input.position3D);

    //figure out the length between the light position and the 3D point on the mesh
    float distanceToLight = length(invrLightPos - input.position3D);

    //diffuse for later
    float4 diffuse = float4(0.0f, 0.0f, 0.0f, 1.0f);

    //store the direction from the spot lights target, to the light
    float3 lightDir = normalize(lightTarget - invrLightPos);

    //Height attenuation (further away, less color)
    float heightAttenuation = smoothstep(lightRange, 0.0f, distanceToLight);

    //cut off angle of the radius of the light
    float cutoffAngle = radians(30.0f);

    //calculate the light intensity for diffuse
    lightIntensity = saturate(dot(input.normal, lightDir));

    //if there is light on the object
    if (lightIntensity > 0)
    {
        //find out diffuse color using light color and intensity
        diffuse += (diffuseColor * lightIntensity);
        diffuse = saturate(diffuse);
    }

    //find out if the lights to be lit within the cone
    float angle = acos(dot(-toLight, lightDir));
    if (angle > cutoffAngle)
    {
        //if not cut attenuation to 0
        attenuation = 0.0f;
    }

    //return color
    return ambientColor+(diffuse*heightAttenuation)*attenuation;
}
```



```
//light information passed in
cbuffer LightBuffer : register(cb0)
{
    float4 diffuseColor;
    float4 ambientColor;
    float3 lightTarget;
    float lightRange;
    float3 lightPosition;
    float coneAngle;
    float3 padding;
    float lightAttenuation;
};
```

For the spotlight shader, I pass in multiple light values that alter the calculation for the spotlight. The main values that are different from the usual directional lights and point lights are **lightRange, lightAttenuation** and **lightTarget.**

In the calculations, we need to know the light position so that we can figure out if the shape is going to be lit or not. With the light position we get the directional vector from the light to the shape itself. We also want to store the length of the vector as well for calculating the Height attenuation to figure out how much light we should be getting based on the distance away from the light.

The core part to the spot light is the getting the light direction using the spotlights target. The spotlights target is what we are pointing our light towards. Light intensity for our diffuse target is calculated roughly the same way as a directional shader, but instead of calculating the distance of the light from the object, we want to use the lights Target as the end point instead of the object's position to check if our object has been lit. This is because even if our light is within range to calculate lighting, our light be facing the wrong way, so we don't want to calculate any lighting if we don't need to.

The second part to the spotlight is checking the spotlight's cone angle to check if the light is within the cone of light. Using this angle, we check if the angle between the look direction and distance vector between the light and the object is less than the cutoff angle. If it is less than the cutoff angle, that means that our light direction is lighting some of the object in that cone. If it is over the cutoff angle, that means we are not going to light the object as the lights target is facing the wrong way. To do this, we set our attenuation to 0.

The final color returned is calculated using all the attenuation factors from before, and the ambient color from the light passed in. The ambient color is always the lowest color, and adds in the diffuse light after attenuation factors such as height attenuation and linear attenuation are calculated.

There is also a set of directional widgets drawn with the geometry shader. These are vertexes plotted along using the directional vector of the spotlight in the scene. The individual vertexes are plotted manually along the directional vector, one at the beginning, middle and end of the vector.

3.0 Directional Light Shader

For the light shader, four lights are available for use in the scene. The lights are directional lights with editable directions and colours.

Resources:
Abertay University - CMP301 lecture 02 and 03 [1]

Shaders used:
The Shaders used in this scene are an additive lighting shader that combines the diffuse color of multiple lights and sets a base ambient color to pixels with no light. There is also a geometry shader for the widgets used to show the light direction from the light model.

Objects/tools used:
As far as objects used, a “BaseMesh” object is used so that the object can be changed easily through polymorphism, and editable in the ImGui editor.

How it works:
The light shader used is a simple vertex shader and a slightly more complicated pixel shader. In terms of the pixel shader, the shader has a single buffer. The parameters passed are ambient light, an array of diffuse colors, and an array of light directions. Ambient light is the lowest amount of light that makes the object always visible, this is editable but keeping a 20% ambient light yields ideal results, i.e: float3(0.2, 0.2, 0.2). The array of diffuse colors that are passed are the colors from the lights, and the array of light directions are also passed in from the lights.

The pixel shader checks the light intensity using the dot product of the inverse of the directional vector from the light, and the normal from the object. If the light intensity is less than 0, then no light has hit the object and the ambient color value is applied to the object. If the light intensity is higher than 0, that means the light has hit the object. After this we can take the diffuse color from all the lights that hit the object on that pixel, and combine the diffuse color and intensity with the color to be applied. We then clamp the value of the color between 0 and 1, in order to get a color less than white, and use this color to light the pixel.

4.0 Vertex Manipulation Shader

The vertex shader works with a simple pixel shader and a slightly more complicated vertex shader. The vertex shader manipulates the normal and position of the vertex over time. This is achieved using the formula of sin(position + time * speed).

Resources:
Abertay University - CMP301 lecture 05 [1]

Shaders used:
The shaders used in this example are a simple pixel shader that is the default shader and just returns a red color. There is also the vertex shader where most of the changes happen.

Objects/tools used:
As far as objects used, a “BaseMesh” object is used so that the object can be changed easily through polymorphism, and editable in the ImGui editor. *

**For best results, you can use a the PlaneMesh option in order to see the wave manipulation more “statically”. If you pass the sin wave through the sphere and cube, and many of the other options, the object moves erratically and it’s not as clear as the PlaneMesh to see the sin wave changes.*

How it works:
For the shader, the object is manipulated by a simple vertex shader that passes a sin wave through the object using the sin function.

In order to do this we need to pass in a time variable for the shader. This time variable is a float value incremented by the amount of time passed between two frames (delta time or in this case timer->getTime();). This gives us real time results that are always changing, and always passed to the buffer differently.

There are two options to tinker with when using this shader. You can change the speed of the sine wave, which changes the rate at which the sin wave is passed through the object. The height variable makes it so you can change the height of the sin wave through the GUI.

5.0 Gaussian Blur Shader

The Gaussian Blur shader is quite long and difficult to understand and keep track of. The Gaussian Blur uses the same mesh as the vertex manipulation shader and light shader, which uses the BaseMesh with mesh changes as an option.

Resources:
Abertay University - CMP301 Lecture 6 [1]

Shaders used:
The shaders used are a combination of multiple vertex and pixel shaders. A texture shader is used in order to render the scenes texture to an ortho mesh to display the post processing effect made by the blur, a simple texture shader is used for the object that is blurred in the scene, and a horizontal and vertical blur vertex and pixel shader are used to blur the whole scene.

Objects and tools used:
A render to texture class is used in order to render the whole viewable scene into a texture. There are multiple of these for the different stages of blur. One for the horizontal blur, one for the vertical blur, one for the downscaling of the texture and one for the upscaling of the texture. An OrthoMesh uses the texture made from the RenderTexture class and displays the post process' effect on the screen.

How it works:
The blur shader is split into six distinct stages. These stages are the RenderToTexture stage, the DownScale stage, the HorizontalBlur stage, the VerticalBlur stage, the UpScale stage and the Render stage.

The Render To Texture stage renders the scene into a texture without any lighting calculations at full texture size. This saved for manipulation for later on in the code.

The scene is then scaled down to half the dimensions in order to save time calculating the blur. This means the GPU has less time calculating the blur, that we are later going to scale up. This works by rendering the scene to a texture again, but rendering an OrthoMesh with the previous texture applied to it.

The half scaled texture is then sent to the horizontal and vertical blur stages. This is where the bulk of the shader calculations are carried out. The pixel shader takes neighbouring pixels for these two shaders and blurs them together in a direction, with a weighting value. The weighting is simply there so that pixels that are closer to the pixel we are doing calculations with are weighed more against the ones that are further away, giving a blur effect. The more neighbours, the more intense the blur will become. For this example we are using 9 samples for both the horizontal and vertical blur stages.

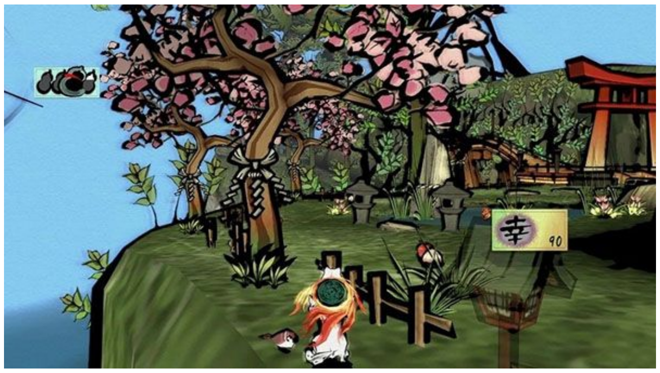
After the calculations are completed and the colours are passed back from the horizontal blur shader, the texture is then passed to the vertical blur shader where the vertical pixels are blurred.

After this stage, we upscale the resulting texture created from the blur effect back to our texture size from the first pass. With this texture, we can draw to the screen. The scene is then rendered normally, with an OrthoMesh drawn over the top using the OrthoView matrix and the Ortho projection matrix.

6.0 Cel Shader or “Toon Shader”

This shader is used to make models resemble a style like the Borderlands style. The shader makes models look relatively cartoon like, so games that come from a strict 2D art style such as games like Street Fighter or Naruto fighting games may also use cel shading for their models and lighting. This shader also makes use of different culling methods with the implementation.

One of the more obvious styles of cel shading is the game okami, that uses borders in an “anime” style that makes it very light hearted. And appeals to the japanese style with thick outlines.



Okami - 2006

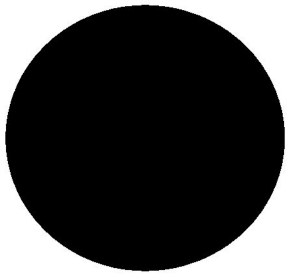
Resources:
Rbwhitaker - Creating a toon shader (no date) [2]

Shaders used:
The shaders used in this example are four sets of shaders. Two light shading, called the celshader_ps and celshader_vs, pixel shader and vector shader respectively. The other two are for the border, which are the silhouette_ps and silhouette_vs shaders.

Objects and tools used:
Because of the way that the silhouette in the cel shading works, the best results are visible when using a mesh with a high amount of normals. This is due to the fact that the vertex position and “size” of the silhouette for the object is based on the normal of the vertex. This makes low poly meshes such as the cube mesh not work. This is one of the limitations of the method that was chosen for the cel shader.*

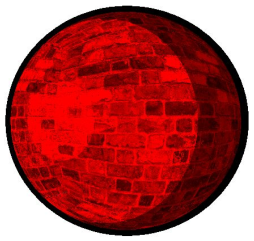
Another tool used is the introduction of cull shading for this shader. The shader uses frontface culling for the silhouette, and backface culling for the original sphere.

**Because of this, you are restricted to some of the more high poly meshes for this particular shader.*



First Pass

Second Pass

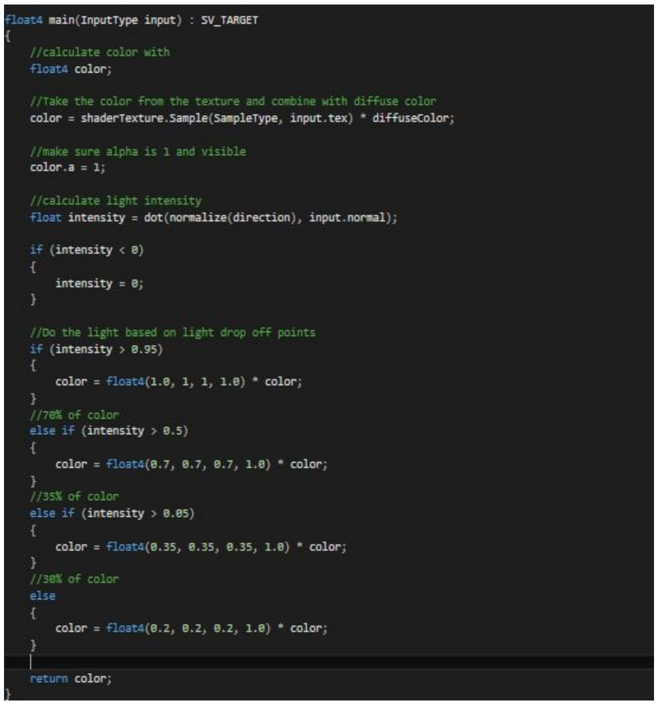


Final Render
(Red color amplified for clarity)

How it works:
For this shader, we have to render the object twice for this method, the first for the silhouette and the second for the original untampered object with the cel shading applied.

The shader in total is split into two parts or passes.

The first pass is the black outline or silhouette of the object that we are rendering. The first pass uses the silhouette vertex shader on the silhouette mesh object. What this shader does is it takes the vertex position of the object and add the normal direction of the vertex, and multiplies it by an offset float factor. What this does is expand the object's size a little bit to bring out an outline for the cel shaded object.

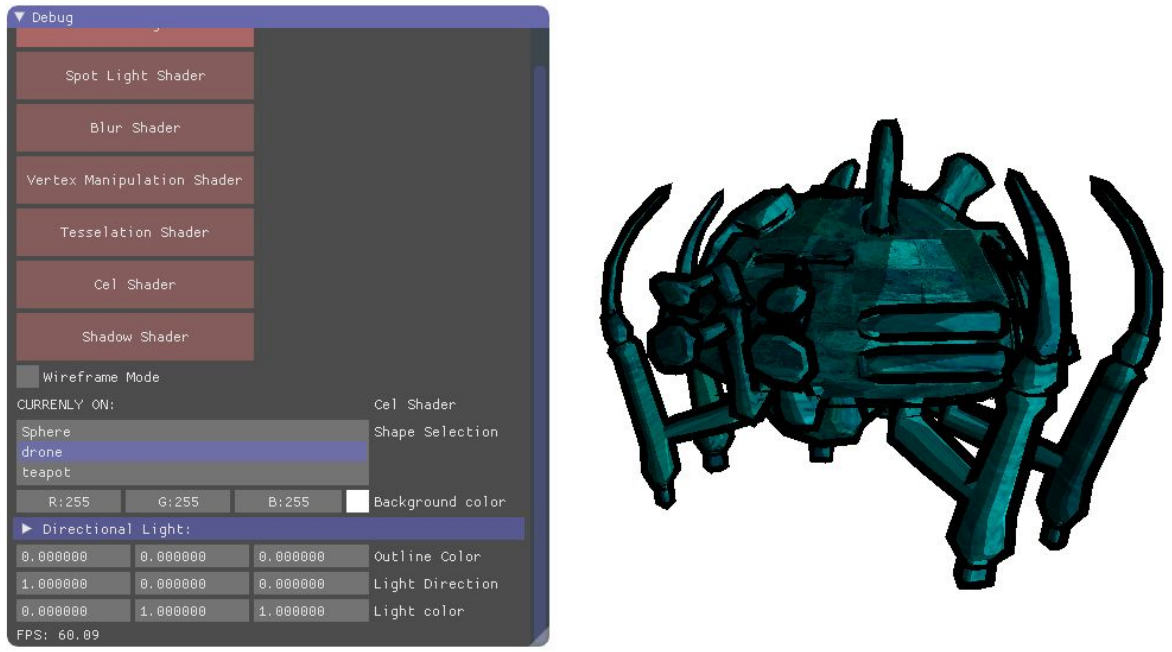


```
float4 main(InputType input) : SV_TARGET
{
    //calculate color with
    float4 color;

    //Take the color from the texture and combine with diffuse color
    color = shaderTexture.Sample(SampleType, input.tex) * diffuseColor;

    //make sure alpha is 1 and visible
    color.a = 1;

    //calculate light intensity
    float intensity = dot(normalize(direction), input.normal);

    if (intensity < 0)
    {
        intensity = 0;
    }

    //Do the light based on light drop off points
    if (intensity > 0.95)
    {
        color = float4(1.0, 1, 1, 1.0) * color;
    }
    //70% of color
    else if (intensity > 0.5)
    {
        color = float4(0.7, 0.7, 0.7, 1.0) * color;
    }
    //35% of color
    else if (intensity > 0.05)
    {
        color = float4(0.35, 0.35, 0.35, 1.0) * color;
    }
    //30% of color
    else
    {
        color = float4(0.2, 0.2, 0.2, 1.0) * color;
    }

    return color;
}
```

With this first pass, we do front face culling on the object. This is so that we are not rendering any faces that we don't require. All we really require are the back faces of the object for the outline that we desire.

<- The second pass or shader is the cel shading. This is a lighting method that is a little bit like the usual default lighting shader, however it has a faster drop off. Instead of colors fading in a gradual way, and having less of a massive contrast in

color, colors are changed in light intensity intervals. For instance if an object has a light intensity between 1 and 0.75, then the color of the object is multiplied by 1 in every RGB value. The next light intensity check, would be at around 0.75, and if the value is between 0.75 and 0.5 then the color would be multiplied by a factor of 0.7. This would keep going until we have vector like shading which resembles the vector art fill color style of
cartoon characters.



CURRENLY ON: Cel Shader

Shape Selection: drone

R:255	G:255	B:255	Background color
0.000000	0.000000	0.000000	Outline Color
1.000000	0.000000	0.000000	Light Direction
0.000000	1.000000	1.000000	Light color

FPS: 60.09

7.0 Tessellation Shader

This shader uses a basic Hull and Domain shader in order to subdivide the object into more vertices and make the model more detailed.

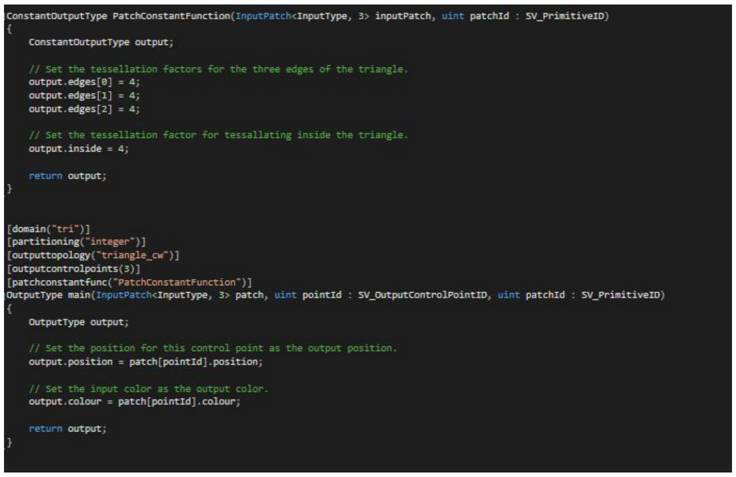
Resources:
CMP301 lecture 09 [1]

Shaders used:
The shaders used for the tessellation shader are a domain shader, hull shader, pixel and vertex shader.

Objects/Tools Used:
For the tessellation mesh, an altered sphere and triangle mesh are the only meshes set up to be able to be tessellated for this example.

How it works:
The tessellation mesh has four shaders used in it's process.

The first shader is a vertex shader. This shader is used to manipulate the control points of the tessellated mesh. If desired manipulation can be done at this stage before tessellation to move groups of vertices after tessellation around. This vertex shader is left as default, passing on the position of the vertices onto the later stage. This also stores the color for the vertexIn the pixel shader we are just returning the color of the vertex that we got in the vertex shader. The pixels for each face are the same as the vertex on each face.



```
ConstantOutputType PatchConstantFunction(InputPatch<InputType, 3> inputPatch, uint patchId : SV_PrimitiveID)
{
    ConstantOutputType output;

    // Set the tessellation factors for the three edges of the triangle.
    output.edges[0] = 4;
    output.edges[1] = 4;
    output.edges[2] = 4;

    // Set the tessellation factor for tessallating inside the triangle.
    output.inside = 4;

    return output;
}


[domain("tri")]
[partitioning("integer")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(3)]
[patchconstantfunc("PatchConstantFunction")]
OutputType main(InputPatch<InputType, 3> patch, uint pointId : SV_OutputControlPointID, uint patchId : SV_PrimitiveID)
{
    OutputType output;

    // Set the position for this control point as the output position.
    output.position = patch[pointId].position;

    // Set the input color as the output color.
    output.colour = patch[pointId].colour;

    return output;
}
```

Next stage is the hull shader stage where the tessellation factor is set for the mesh. We want to set the tesellation factors for all the edges of the triangle. In the case above, we set them all to 4. To go alongside this, we want to set the output topology and all the heading functions that go with it. In the case above, we use triangle_cw to render the new faces in triangle_cw rendering format, so the vertices for each face are rendered counter clockwise and the indices match.

Finally in the domain shader, we use our newly made patch data from the hull shader, which has our tessellation factor and patch control points to set the new position for the new vertices made. The way it does this is by taking our triangle patch from the hull shader, and places the newly made triangle into position using UV space coordinates.

8.0 Shadow Shader

The shadow shader uses the view and projection matrix for the lights to do depth testing. With this depth testing we can display a shadow on the screen using a shadow map across multiple objects.

Resources:
CMP301 lecture 10 [1]

Shaders used:
The shaders we use in this example are pixel and vertex depth shaders and pixel and vertex shadow shaders. The depth shaders deal with getting depth values back from the objects in the scene and turning them into a texture through render to texture, and the shadow shaders take this texture and generate shadows.

Objects/Tools Used:
In this example we use the BaseMesh with 3 options; the teapot, the drone and the sphere. We use a plane mesh for the floor as well as a translated matrix for the floor so that it can be moved underneath the BaseMesh object.

How it works:
The way the shader works is split into two stages.

The first stage is the RenderToTexture stage (shadow mapping stage). This stage uses the depth vertex and pixel shader to create a shadow map texture.

The way it does this, is through the pixel shader, it takes the Z pixel depth and divides it by the homogeneous W position in the matrix. What this does is it gives us a value that is between 0 and 1 that we can then use to color the screen white or black. White is for objects far away, black is for pixels that are close. This stage is all done within a RenderTexture to store the texture for later use.

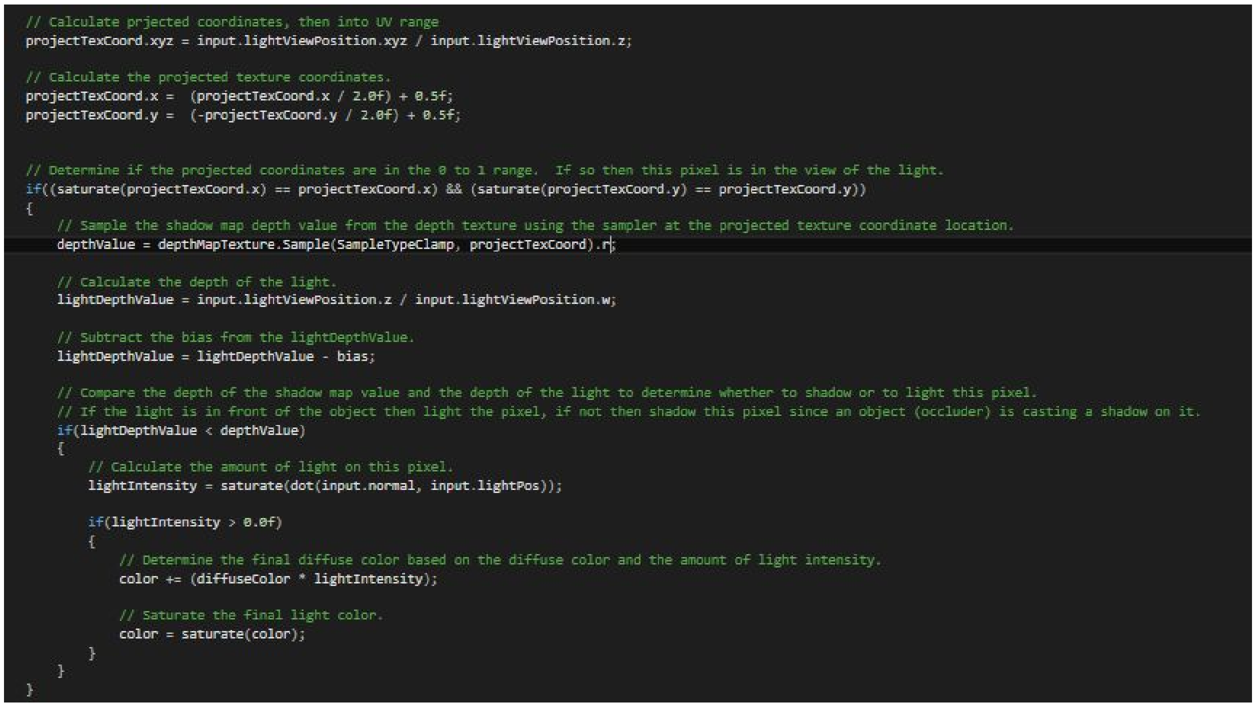
What's important about this stage is the matrices used. In this stage we want to use the viewMatrix from the light and not the camera. This is because want to see what's far away from the light instead of the camera when we're calculating lights. For this, we use the matrices generated from the light class, in particular the lights viewMatrix and ProjectionMatrix.

After this stage, we should have a shadow map texture that will allow us to tell where light should and shouldn't appear on pixels of objects. We can then use this texture to render the shadow.

Our next stage renders the scene normally but through the shadow shader. To summarize the shadow shader, it takes in parameters from the light source namely the diffuse color, ambient and position and calculates the light given to the pixel using them.

As well as the light parameters, we also take the light view matrix and the light projection to calculate light viewing position in the vertex shader stage. We will require this later on so we pass this onto the pixel shader. We also check the position of the light for the pixel shader using the world matrix from the object we want to light.

The pixel shader for the shadow stage takes in two textures, one is the shadow map that was generated from the depth testing before, and the other texture is the texture that we want the mesh to have. The basis of the pixel shader for the shadow shader is to figure out if the pixel is to be lit or not, based on if the projected texture coordinates are between a range of 0 and 1. If the texture coordinates are between those values then we are definitely lighting this pixel.



```
// Calculate prjected coordinates, then into UV range
projectTexCoord.xyz = input.lightViewPosition.xyz / input.lightViewPosition.z;

// Calculate the projected texture coordinates.
projectTexCoord.x =  (projectTexCoord.x / 2.0f) + 0.5f;
projectTexCoord.y =  (-projectTexCoord.y / 2.0f) + 0.5f;


// Determine if the projected coordinates are in the 0 to 1 range.  If so then this pixel is in the view of the light.
if((saturate(projectTexCoord.x) == projectTexCoord.x) && (saturate(projectTexCoord.y) == projectTexCoord.y))
{
    // Sample the shadow map depth value from the depth texture using the sampler at the projected texture coordinate location.
    depthValue = depthMapTexture.Sample(SampleTypeClamp, projectTexCoord).r;

    // Calculate the depth of the light.
    lightDepthValue = input.lightViewPosition.z / input.lightViewPosition.w;

    // Subtract the bias from the lightDepthValue.
    lightDepthValue = lightDepthValue - bias;

    // Compare the depth of the shadow map value and the depth of the light to determine whether to shadow or to light this pixel.
    // If the light is in front of the object then light the pixel, if not then shadow this pixel since an object (occluder) is casting a shadow on it.
    if(lightDepthValue < depthValue)
    {
        // Calculate the amount of light on this pixel.
        lightIntensity = saturate(dot(input.normal, input.lightPos));

        if(lightIntensity > 0.0f)
        {
            // Determine the final diffuse color based on the diffuse color and the amount of light intensity.
            color += (diffuseColor * lightIntensity);

            // Saturate the final light color.
            color = saturate(color);
        }
    }
}
```

Next we want to sample the texture from the shadow map using the texture coordinates we got earlier on. We use this sample as a depth value. Since all the values of the RGB color of the pixel in the shadow map are set to the depth value, we only require the red value since they are all the same. We then use the light viewing position that we made in the vertex shader to calculate the depth of the light. If the light's depth value is more than the depth value from the shadow map then we know not the light up the object since the light is being occluded by another object. If however the light is in front of the shape and the lights depth value is smaller than the object, we want to light up the object since it's not being occluded by any other objects.

After this large test is done, we handle the diffuse color from the light as we normally would in the directional light shader and add it onto the final color we want to return. We then add in the color from the other texture we passed in (the texture we want our object to have).

9.0 Critical Reflection

Originally, I planned to do more post processing effects in the hopes to show realistic light calculation. The post processing effect I wanted to do was volumetric lighting or god rays in order to see the light being casted out from the lights screen coordinates to the object being lit. There are still files in the project to do with the volumetric lighting, however a solution wasn't found in time to complete the shader.

One huge pitfall was getting the program to run while using the XMMATRIXMULTIPLY function. Using this function caused several phantom errors that don't actually exist and caused several crashes when starting up before the project would finally work. The work around I found for this was to add `#define _XM_NO_INTRINSICS_` At the top of the header file for the app1 file.

There is quite a lot of shaders I felt I could have put a lot more work into. The tessellation mesh should have had a tessellation factor that you could pass in and manipulate. The vertex manipulation shader could have had more to it (different sin wave directions, passing through x and y). The spotlight could have used a speculation factor and a little more work on attenuation.

The Cel shader border could have been implemented using a different strategy in order to be more efficient. Currently the shader works by drawing something twice. Despite front faces being culled from the border of the object, the scene still required 2 versions of the mesh. It would have been good if there was a possible implementation done through the vertex shader to discard the inefficient double render technique. Despite being flawed in some aspects for efficiency, I am extremely pleased with how the results look for the cel shader.

The shadow shader could have had a lot more work put into it to deviate from the lab. Unfortunately due to time constraints, I wanted to get something outwith the lab done rather than the stuff from the lab expanded on. This led to the creation of the Cel and Spotlight shaders. Ultimately, I felt like I had to put shadows in the application in order to show I was able to do the task successfully, however I don't feel like I gained much from implementing straight from the lecture.

I would have liked to do a lot more resources online outside the lab work. Although there is the cel shader and spotlight done outside lab time, I would have liked to do bloom or deferred lighting as an example.

The App1 cpp is extremely cluttered and frustrating to navigate. If I were to go back I would create a new scene entirely for each shader but I felt like it would have been better use for my time to create the shaders instead.

I would have liked to do more with the geometry shader, but largely used itfor debug tools for the spot light.

10.0 References:

[1] - Abertay University & Paul Robertson - CMP301 lectures available at: www.blackboard.abertay.ac.uk

[2] - RB Whitaker - Creating a Toon Shader available at: http://rbwhitaker.wikidot.com/toon-shader

[3] - Tom Dalling, Modern OpenGL 08 – Even More Lighting: Directional Lights, Spotlights, & Multiple Lights (2011)
Available at :
https://www.tomdalling.com/blog/modern-opengl/08-even-more-lighting-directional-lights-spotlights-multiple-lights/