Report

# What are shaders

“A shader is a set of instructions to the GPU which are executed all at once for every pixel on the screen” (What Are Shaders?, 2021)

This quote is a good starting point to describe what a shader is. A shader is a recipe or a to-do list. It has an exact set of instructions. And when the recipe is complete, the shader will output a colour. This colour is then sent to the monitor to make a single pixel on the screen change colour.

A shader will take various inputs and then use those inputs to decide what the output colour is; at its most basic, a shader can be represented by the image below:

Shader

Input colour Output colour

Shaders can be a lot more complicated than this picture shows; in fact, this is a gross understatement of what shaders are capable of.

Most common shaders will take in many, many inputs and decide what the output colour should be based on several factors.

Shaders can be used for something as simple as tinting a picture a particular colour or as complex as rendering a 3d scene to a 2d viewport.

All of the above is in reference to a particular type of shader – a pixel shader. But shaders can do much more than change colours based on several factors; A vertex shader can be used to manipulate the vertices of a model based on inputs given.

A shader is an algorithm that will manipulate an object or a texture based on the environment around it. This links us to our next topic

# Why do we use shaders in games?

“ shaders transform dull and dreary environments that may have lacked a cohesive look to staggeringly beautiful works of art” (Shaders in Game Design: Origin, Basic Design Types, and How to Create Your Own, 2021)

We use shaders to make an object in a scene react to the scene around it, and it can be as simple as a reflection, but a less obvious example would be normal mapping. With this technique, you can make a flat 2D plane appear slightly 3d by interacting with the scene's light source.  
Imagine a brick wall in a game. If the artist were to create grooves and minor imperfections in the bricks by adding more vertices, this model would be a lot harder to compute since more vertices should be taken into account by the GPU.  
By using the technique described above, you can keep the models very simple and easy to compute while still creating the illusion of depth by changing the way the model interacts with the light to create slight imperfections and a rough face on an otherwise perfectly smooth plane.

This is why we use shaders. Because they can make a complex model, without using too much computational power.

Another reason shaders are used is to create shadows in scenes. Can you imagine if an artist hand drew all the shadows on a model? It would be a lot of work. And as soon as the relative position of the light changes to the model, the shadows would be incorrect.  
By calculating the shadows in real-time, it means we can have shadows that are reactive.

Lets use Minecraft as an example:



On the left, you have the standard shaders – very basic shaders. On the right you have a complex shader. You can see that on the right it has a glare/bloom effect around the sun, and shadows that reflect the suns position.

The scene on the right looks a lot more complex and a lot nicer. But the scene is the same as the left.

But there is more to shaders than just shadows and lights.

Lets take a look at a different shader for Minecraft:



This image better describes what shaders are capable of. On the left, advanced shaders are turned on, and on the right, the default shaders are used.

There are many differences between the two, but the most notable is the water. With the advanced shaders turned on, the water has reflections. You can see a reflection of the mountain in the distance on the surface of the water, whereas on the right – with default shaders- the water looks kind of boring and doesn’t really resemble water at all.

# How do they work?

The most basic shaders will have two parts: a pixel shader and a vertex shader.

Firstly the vertex shader will run. It runs once for each vertex on a model. And it will take those vertices and translate them from 3d world space and relative space to 2d screen coordinates.

Afterwards, this shader's output is passed down the “pipeline”, which means that it is passed onto the next shader.

The next shader is the pixel shader, most pixel shaders will compare a pixel against the lights in the scene using the normal of the pixel (this just means the direction that the pixel is facing in the scene). If the pixel is facing a light, the colour of the pixel is slightly adjusted to account for the colour of the light bouncing off the surface of the pixel.

Then the pixel shader outputs the final pixel colour, where It is sent further down the pipeline until it is rendered on the screen/

# Misconceptions of shaders

“A shader is exactly what it sounds like - it ‘shades’ vertices and pixels different colors. ‘Shading’ isn't just ‘making it darker’ ” (Why Even Use Shaders?, 2021)

A misconception of shaders is that they add dark bits where shadows are. And that makes sense. When an artist is painting on a canvas, they will add shading to their painting by adding darker shades of a particular colour in areas that might not get as much light.

Shaders do a lot more than make pixels darker, they can be used to make a model look different. For example, cell shading is a type of shader that makes a 3d model look cartoonish by changing how the lighting affects the model so that it looks as if model was drawn in a manga-ish style.

Dev Log

# Day 1

## Put a textured floor and teapot in a scene with per-pixel lighting

### One of the lights should pulsate on and off

I started from lab 19- the one with the better model importer.

Immediately removed the unnecessary 2% code for the lab (making the model walk) and changed the robot.x model for the teapot.x model.

Right near the bottom of scene.cpp , in the UpdateScene function, I made a small snippet of code that changed the lights strength over time. It used a multiplier that switched between being negative and positive: the light would start off at a brightness of 40 and decrease. once the light strength hit 0, the negative would flip to be positive and the light would now get brighter over time until it hit 40 and then the multiplier would flip again, making the brightness decrease again and restarting the loop.  
this could have been done better with a sin wave, this would negate the need for a multiplier that flips between being negative and positive.

### Whilst the other light should gradually change colour

below that, with pretty much identical code, the other light will switch between two different colours, I did this by using the same flipping multiplier technique, but this time it changes the colours ‘.x’ value. Meaning that I am gradually turning the light’s amount of red light up and down.  
it does all the way down to 0, and up to 1 before flipping the negative, again, this could probably have been done better/quicker using a sin wave.

This would have definitely looked a lot nicer if I had used the HSL to RGB calculations we did in the first semester and made the light cycle through the Hue value. But in the instance of saving time, I elected not to explore that option any further.

### Include ambient light attenuation in the lighting equations

As far as I am aware, the base Lab19 already included attenuation by dividing the light’s strength by the distance, but it was barely noticeable, so I multiplied the distance float to smooth this attenuation more and make it more noticeable

## Add a sphere model to the scene, but using different shaders

This took some time, a little trial and error, but eventually I managed to get another vertex shader working called Wiggle\_vs.

### The vertex shader must constantly move the vertices in some way to produce some kind of pulsating effect

Once I decided I needed to add another float to the PerFrameConstants, I realised that the padding variables are essentially unused float variables, so instead of creating a new float in the struct and buffer, I decided to use padding1 as a Time variable. This time variable would increase by frametime each frame, though in order to stop the numbers from spiraling out of control after a certain amount of time, after a certain large number is passed, the time is set back to 0. Once I was able to pass this number over to the GPU, I used it in the wiggle\_vs vertex shader.  
at first I was attempting a wiggle effect, so I did the calculation sin(padding1) and added that to the models position just before the world position is calculated. This made the model scale infinitely and back. Assume this is because some of the numbers were approaching 0 or becoming negative, adding a +1 to the end of that equation made the sphere effectively scale up and down, but the scaling was done entirely in the shader, not by altering the models size.

### The pixel shader must constantly scroll the texture coordinates and tints the sphere texture to a fixed colour

Using the padding1 variable I modified earlier, I simply found the code in the vertex shader that dealt with UVs and I added padding1 to the UVs on both the x and y coordinates. This created a diagonal scrolling effect with the UVs

I later realised that I completely misread the brief and noticed that the UV scrolling is required to be done in a pixel shader.

So I made a pixel shader for the wiggle effect and I moved the code from the vertex shader to the newly created pixel shader

## Add a textured cube to the scene, again using different shaders

I started by adding a cube to the scene and positioning it in an appropriate area.

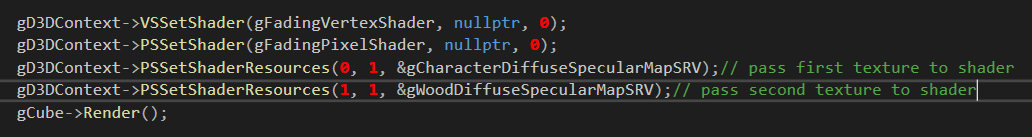
After that, I added two more shaders, Fading\_VS and Fading \_PS

### The texture on the cube should gradually fade from one texture to another and back again

I added another texture to the game. The Wood texture. I plan for my cube to fade between metal and wood.

I added a second diffuseSpecularMap to the shader like so: 

And made sure to pass a second texture to the shader when I render the cube:



Despite the assignment brief sugesting I use liner enterpolation

# Day 2

## Adding another cube to the screen with separate shaders that use normal mapping to make the flat faces appear more 3d

I thought this was going to be easy at first. I took a look at the lab work I did for normal mapping and I copied the shaders over. It works similarly to the above. It takes in a second Texture2D, this is not a normal texture like last time, but a normal map. Meaning that the r g b values actually represent x y z values in world space (relative to the face of the model – I think ). It uses these XYZ values to alter the worldNormal of each pixel.

After this point, the same lighting equations as before were used. But instead of using input.worldNormal, we use the worldNormal we just generated.

I had an issue with the Tangent vertex loading the uvs into the tangent. After spending a while looking at the DX11 API and how the cpu and gpu communicate, I spotted this if statement:



After this I realised my foolish mistake and I added the necessary parameter when loading in the mesh (I didn’t realise I had to put the second parameter in):

  
after this I made a note not to make this mistake again, and the code executed as expected

## Replacing the ground with a different texture, one with parallax mapping

This works the same as the normal mapping, but you include a float variable in the gPerFrameConstants named Parallax depth. This depth is used to offset the texture to make it look more 3d. note that it does not actually change the shape of the model at all, it just simply offsets the textures.

# Day 3/4

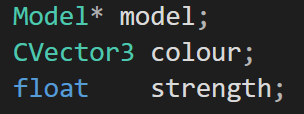
### Splitting the light off into its own Class and passing an array of lights through the PerFrameConstants instead of hardcoded values for each light

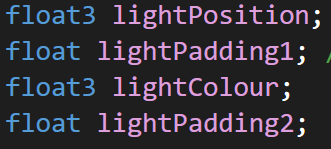
This was a very difficult challenge.

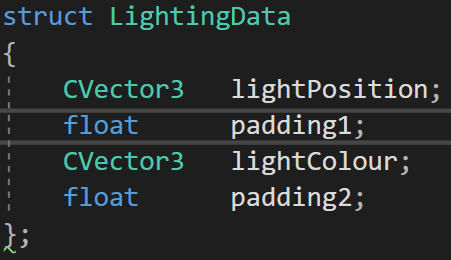
I started off by making an empty class called light, and copy pasted the code in the struct Light located in scene.cpp. I made all the variables public so they can be accessed easily – I’m not messing about with setters and getters just yet, I just want to see if this works.

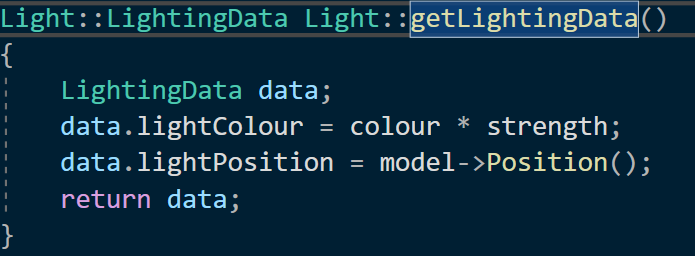
After I copied the code over, I realised that I can not pass the light class through to the GPU because it contained a pointer to the Model class; after a long, late night chat with Laurent he explained that the GPU can take a class as an input, as long as it is using PODs (plain old data types). This means that if I had a light class with many floats, I would be able to pass that over to the GPU. Unfortunately because the light contained a model, this foiled my plans to pass an array of lights to the GPU.

Not to worry, I came up with an alternative.  
currently my light class contains 3 things:

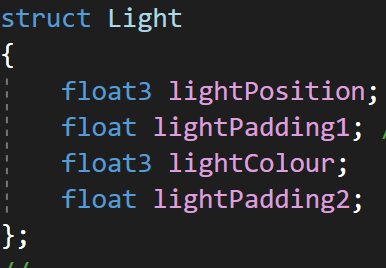
  
but the GPU is expecting code that looks like this:

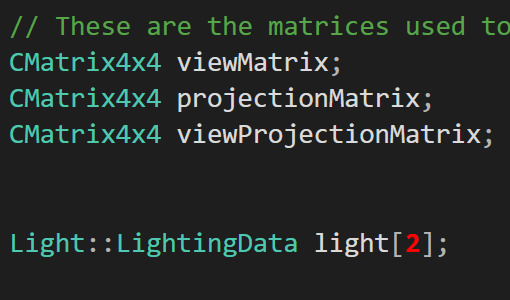


So I made a Structure in my Light Class. To contain the data to send over to the GPU which matches what the GOU will expect:  
  
I then created a function that will take the attributes of the light, and return them as the struct shown above:

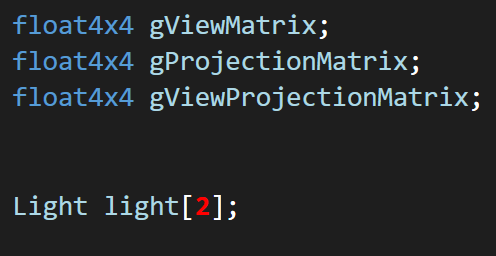


After this I created a matching struct for the GPU-side where cVector3 becomes float3:



Now all I have to do is modify the PerFrameConstants to use the light structures I have created.   
  
this is the CPU- side.  


And this is the GPU-side:



Note that they use a hardcoded number.  
I plan to limit the number of lights the player can have in the scene.  
in unity, they limit it to 4, but this number can be changed in the settings, I will later change this to a round 10.

After this, I just had to modify my shaders to use the new structure array instead of hardcoded values in PerModelConstants

It was fairly simple to modify these shaders, all I needed to do was remove the duplicate code and instead of using light1Colour I now use light[i].LightColour



After making a few changes to scene.cpp to accommodate the movement of light from a local struct to a class, the program was ready to run.  
and to my surprise it ran first time. With the only notable discrepancy that the lights were all white – the lights could not show colour at all.  
  
after many hours of debugging I found this to be the culprit:



I’m embarrassed it took me so ling to notice this. But the lights were being truncated to just their red value, meaning all the lights were monochromatic – just different brightnesses of the same white colour.

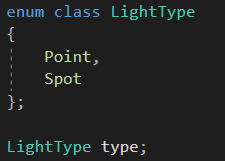
After changing these to float3 the scene went back to what it was before.  
even though the scene looks identical to how it did at the end of day 2, IHave learned a lot about dx11 and cbuffers (I spent a lot of my time debugging looking at documentation), and learned especially a lot about how the CPU and GPU communicate.

Now that I have started encapsulating things into classes, I will finish up with the lights by adding shadows and different kinds of light. And then will finish encapsulating things into classes until the code is much cleaner and resembles TL-Engine a little more

# Day 5

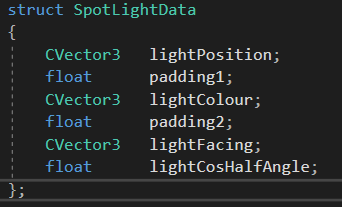
## Expanding upon my light class to use spot lights

I wanted my lights to be fairly flexible. So I started by making an enum LightType:



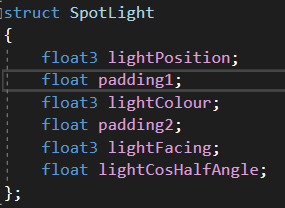
So if you wish to change your light from a point light to a spot light, all you need to do is change the .type property of the light to Light::LightType::Spot and a different set of calculations are used in the shaders.

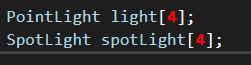
Spot lights require more data to be passed to the shaders than point lights so I had to create a new lightData struct to be passed through the cbuffer.:



As of right now, I am not attempting to get the shadows on the point lights working just yet, I will be happy with just a cone light with no shadows.

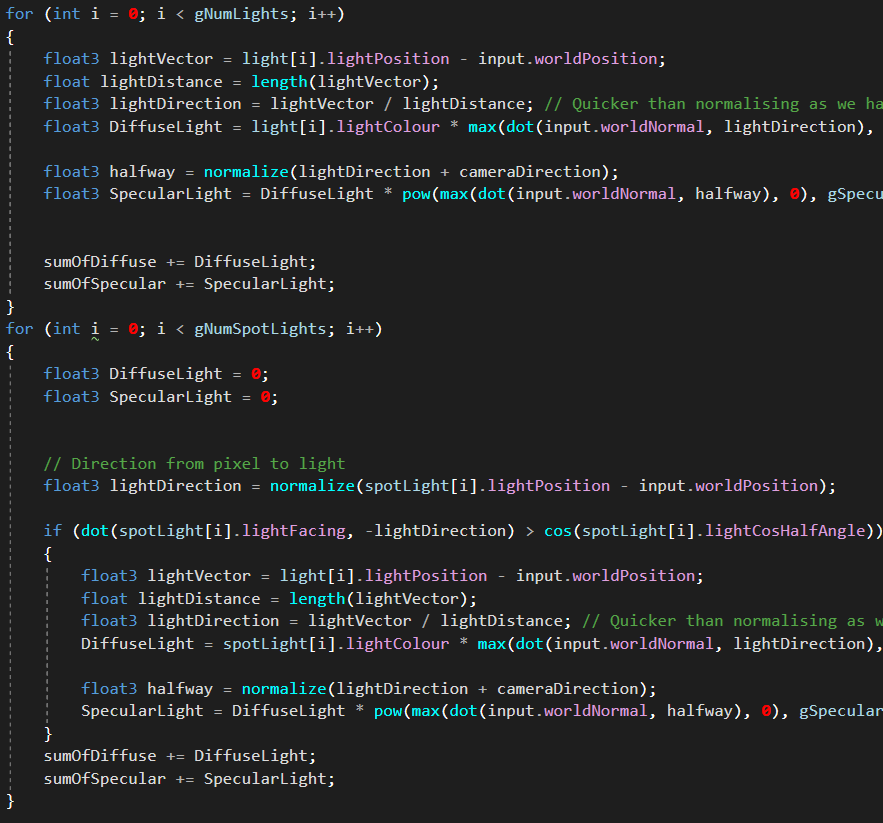
Next, I needed to make a struct in Common.hlsli to mirror this struct:





Dy default there will be maximum 4 lights of each type. One day I plan to make this dynamic so it will automatically adjust to the amount of lights you have placed, but for right now, I am hardcoding it.

The shaders will loop through all the point lights and do an algorithm for point lighting, then afterwards, it will loop through all the spot lights and do a different algorithm for spot lighting:



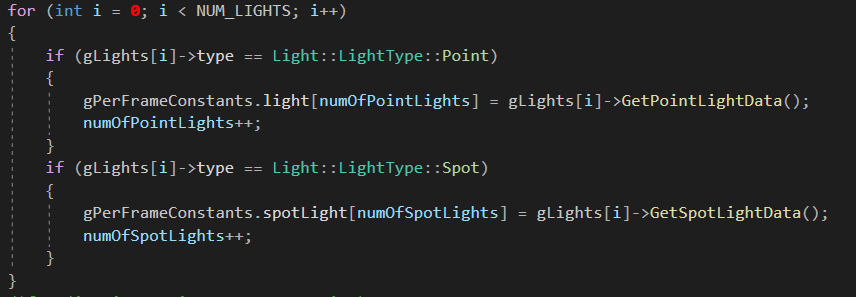
Spot lights

Point lights

It is done this way to minimise any IF statements in the shaders.

Instead, all the lights are split into different arrays on the CPU-side – this is why we have separate arrays for point and spot lights (so that the lights are already sorted on the GPU-side and no if statements are required.

This is the code that splits the different lights into the different arrays:



Now, if I want to change a light type, it is as simple as changing the .type modifier of the light and the rest is done for you by the program

## Encapsulating texture code into its own class

First I made the texture class, and ran the project to see if it compiled- might seem pointless at first but it is actually not. I am checking to see if the class name Texture has any ambiguity with existing code.

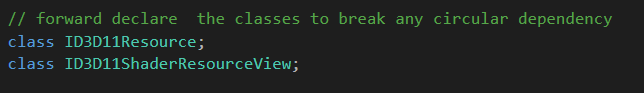
If there was another class or struct named Texture, this would reveal the issue before I start moving lots of code around.

I added the TextureMap and the TextureMapSRV as properties of the texture class:



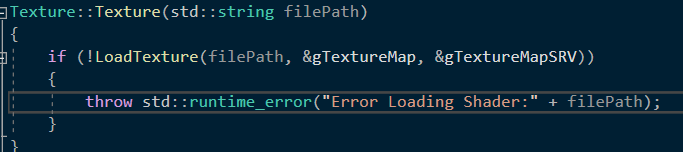
To my understanding, the texture map is where the texture is kept and the shader resource view is like a pointer to the texture map so that the shaders can access the texture.

In order for this to work, I needed to forward declare the appropriate classes:



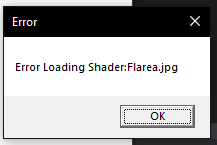
If I were to using a #include instead of this forward declaration it would create a circular dependency, with multiple scripts all including each other effectively causing an infinite loop.

I created a simple constructor to take in the file name of the texture and then load that texture:



If there are any errors, they will be caught in the scene.cpp file:   
while I was at this, I realised from some debugging on a previous day that if there was an error with loading a texture, the program would just tell you there was an error with shaders and leave you to figure out which file is missing / spelt wrong.  
I made a slight edit so that now if there is an error tryig to load a texture, the program will tell you exactly which texture it failed trying to load.

Example of loading a texture with a typo below:

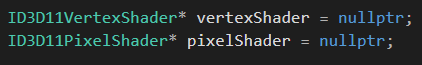


## Encapsulating shaders to their own class

Despite there being a Shader.cpp and shader.h, there is no Shader class. Meaning if we want to create a new shader, we will have to write a lot of code and create many new variables.

After adding a few shaders, the files begin to become excessively large. I was in the process of adding a new shader that used blending, and I was frustrated by the amount of code I had to duplicate. So I decided I would encapsulate all that code into a class for easier use.

I started by creating the class in the shader.h file and I commented all the duplicate code, out.  
I wanted each shader to have a pixel shader and a vertex shader, and you will pass in a file name like “Wiggle” and it will append “\_ps” and ”\_vs” to the end of that to load both the pixel shader and the vertex shader, which will be stored as member variables:



But, there are a few scenarios where you may only want to load a singular pixel shader, without the accompanying vertex shader. So I added a constructor with a few parameters:



By default the two Booleans are set to true so there is no need to enter the additional parameters unless you need to change them.  
if the Boolean is set to false, then the default shader will be loaded.  
for example:



My alpha blending shader will override the default pixel shader, but it does not need a standalone vertex shader. In this instance, the default vertex shader is loaded into the VertexShader member variable.

Now when I set the shaders ready for a model, I can access these member variables:



The above is a snippet of code where the normal mapping shader is being set ready for the “bumped cube” in the scene.

In future I may want to be able to attach a shader and texture to a model. And when the model is rendered it will automatically set the shaders and textures before rendering the model

## Blending

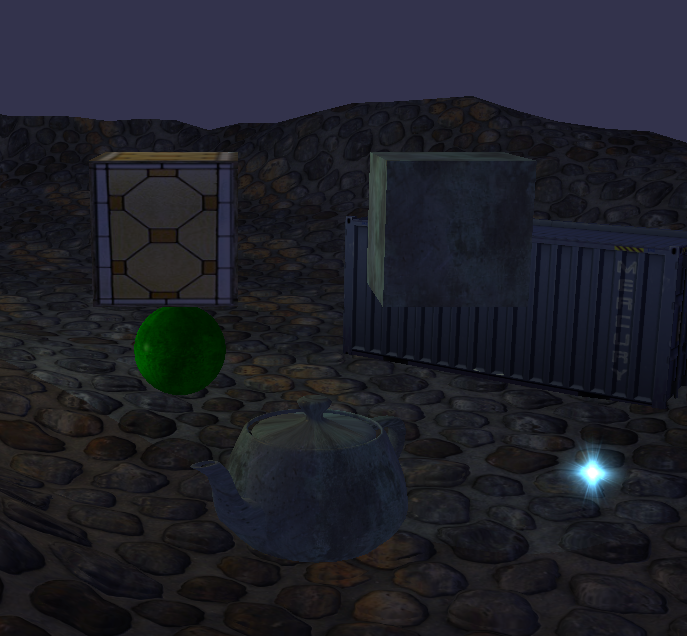
Now that I have made it easy for myself to create new shaders. I creates a blending shader.  
innated of using alpha for shininess, in this shader, it is used to make an object transparent.  
all I needed to do was change the last line of code in my normal default lighting shader

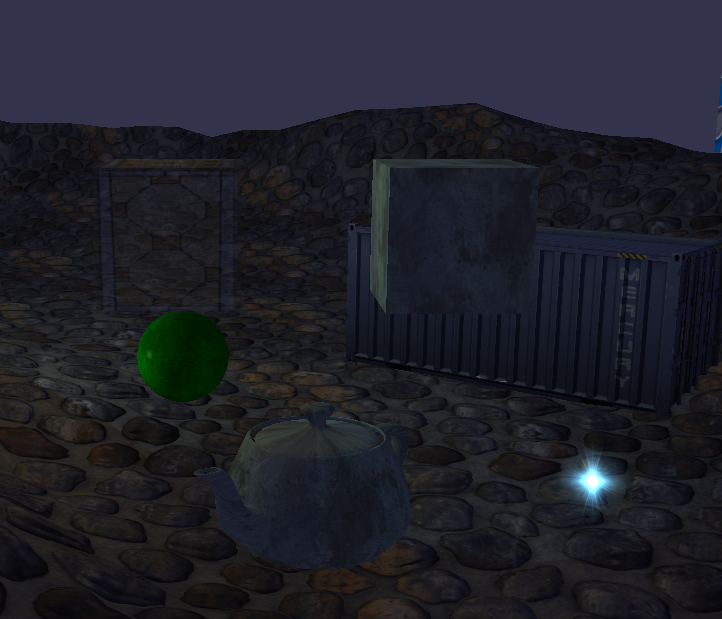


Instead of always returning an alpha of 1. I return the alpha read in from the image.

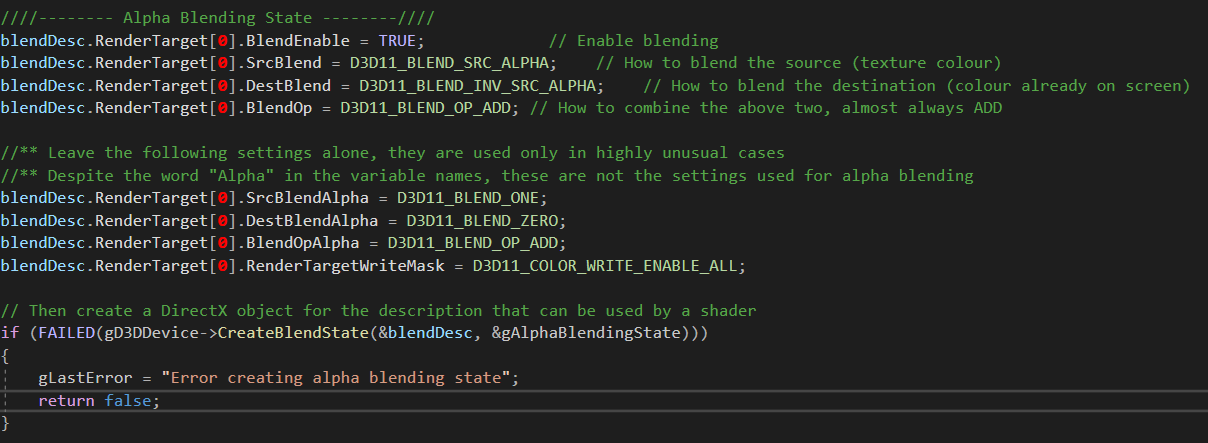
Here is a before and after picture where alpha is taken into account:

Before:

  
After:



In order to do this, I had to write my own blending state to tell the GPU how to blend the two colours.



All I had to do was copy the code for additive blending and change SrcBlend and destBlend to reflect the formula that I need to calculate the correct output colour

## Adding a cell shaded troll

I started by adding a troll to the scene, and positioning the model in a place that can be easily seen.

The cell shading works by doing 2 things.

1. The object “outline” is drawn by drawing a the model slightly bigger and inside out.
   1. This is done by scaling up the model in the vertex shader and using front-face culling
2. The lighting gadient is made jagged. Instead of smooth lighting where the lighting seems to fade in and out, the lighting uses larger steps – the blend between bright light and dim light is not nearly as smooth – in fact there will be a visible line where the shading of the light steps up or down.
   1. In our case, this is done by a cellmap texture. The texture contains shades of grey which represents intensity of light. Instead of blending between two shades. The light will effectively be rounded up or down to the closest intensity that each shade of grey represents.

I also had to make a number of changed to the code to make this work with multiple lights, using arrays in the cbuffer – no hardcoded lights

The result is quite satisfying:



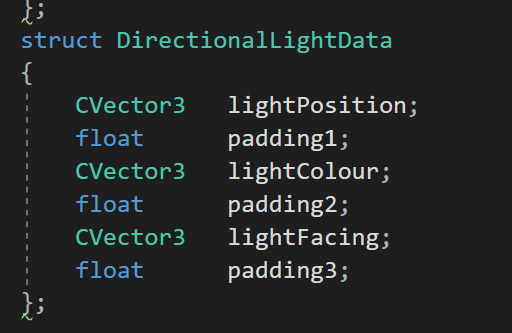
I tried doing this with the default troll texture, but unfortunately it didn’t quite work – a high quality image doesn’t suit the cartoonish style I was going for.

# Day 6

## Adding a directional light type

The directional light is very similar to the spot light, not only do we need to store the position, colour of the light, but we also need to store the direction we want the light to face.

So I created a new struct for data specific to directional lights:



This is the data that is passed over to the shader.

In each shader, the algorithm for directional lighting is now the same as the algorithm for point lights. With 2 changes.

There is no attenuation – therefore no need to calculate the distance to the light

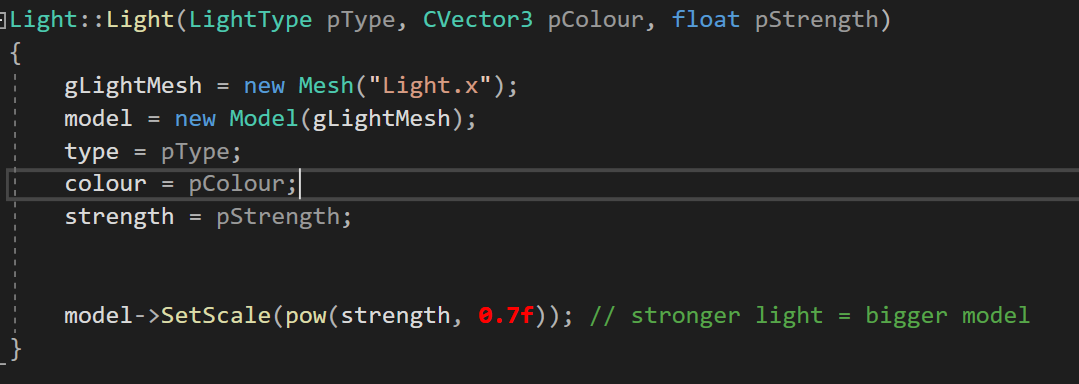
There is no need to calculate the direction from the pixel to the light. Instead we get the negative of the facing vector of the directional light.



Other than those changes, the shader code for a directional light is identical to the code for a point light

## Using a constructor to create lights

Previously I was creating an empty light, and then setting all of the properties.

This has now been changed to utilize a constructor to set the initial values of the light. 

This means that most of the messy code for lights has now been moved over to the light class and runs in the constructor if the light

This means that lights can be created and have initial values set, and the code isn’t to messy:

