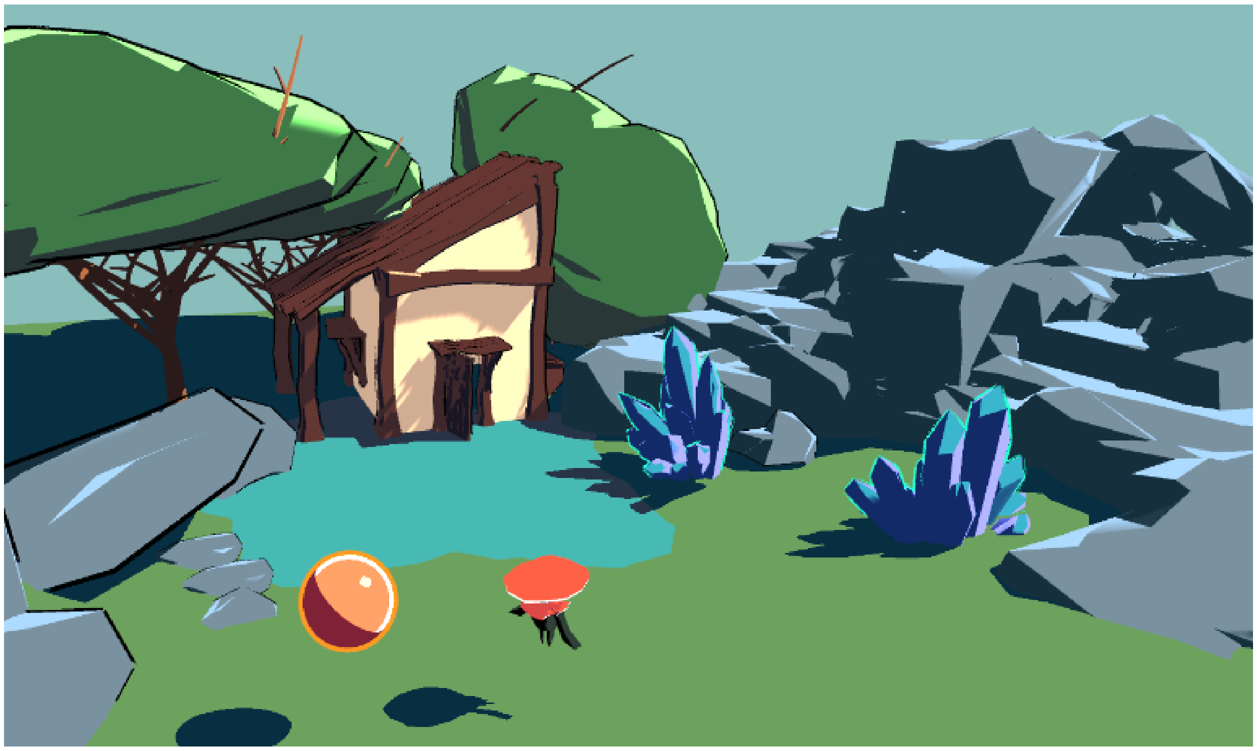
CS 112

Final Project
Unity Toon-Styled Shader

By Hung Bui



Introduction

For my Introduction to Computer Graphics final project, I've decided to expand on what we've learned about shading techniques, to make a customizable shader that can blend between smooth and toon-style shading. I expand upon Lambertian and Phong shading models, programming diffuse shading, specular highlights, rim lighting, and inverted hull outline, and utilizing built-in shader components to implement shadow mapping. This project is made in **Unity**, in Nvidia's **Cg Shader** language.

Background About Me

I have a deep interest in game development, and am particularly intrigued by game aesthetics, and graphics. Having made several projects in Unity using built-in material shaders, post processing, and lighting, I wanted to expand my knowledge on shading to allow myself a level of customization I had not before. I have a liking for cell-shaded, or toon-shaded, aesthetics as seen in most anime, and some games such as *The Legend of Zelda: Breath of the Wild.* So, that is what I decided to implement, in hopes of using this shader in a future project.

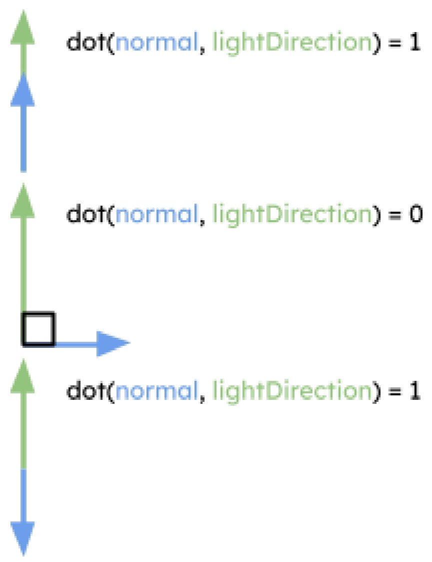
Creating the Shader

Flat Shading

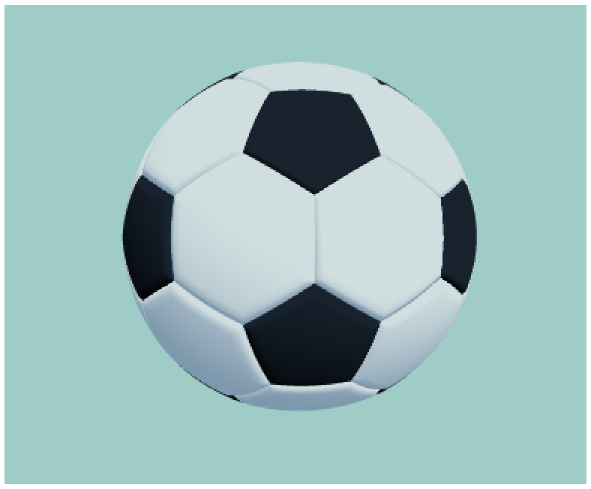
Very straightforward, material is given a color, in which the fragment shader simply returns that color.

Diffuse Shading

To get diffuse shadows to appear, I used the Lambertian model of shading, which takes the dot product between the normal vector and multiplies it by the lighting direction. This returned a value between -1 and 1, depending on the angle between the normal and light vector. When these two vectors are the same(the light pointing directly against the surface), the dot product is 1. When perpendicular to each other(the light parallel to the surface), the dot product is 0. Any angle larger than 90(the surface not facing the light) results in a negative number.

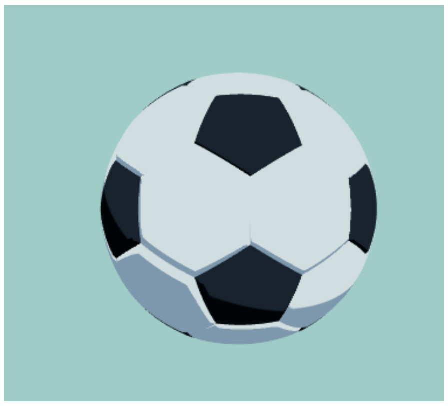

dot(normal, lightDirection) = 1
dot(normal, lightDirection) = 0
dot(normal, lightDirection) = 1

I clamp negative values to zero, and multiply this by the color, which darkens the shadows too a black. I created a new color input for ambient light, which I added to the color. The result is as follows:

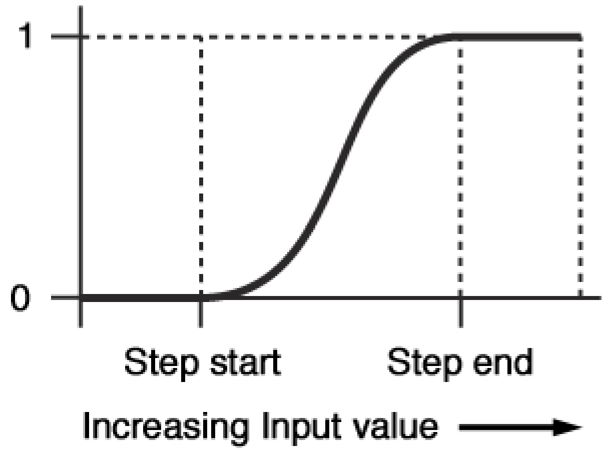


Smoothstep vs Discrete

To get a cartoony effect, the color values between 0 and 1 had to be discretized to either 0 or 1, depending on a threshold. This threshold I made to be originally 0(clamping values below 0 to be 0, and above 0 to be 1). The result was as follows:

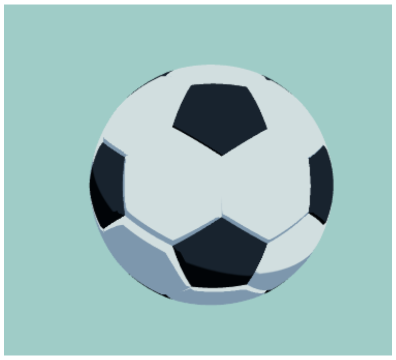
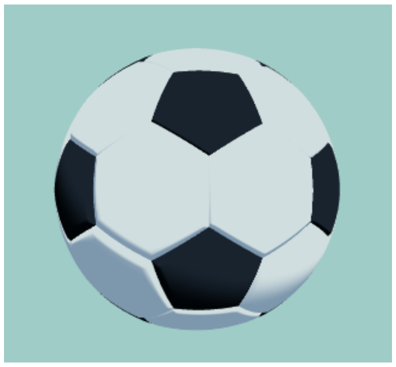
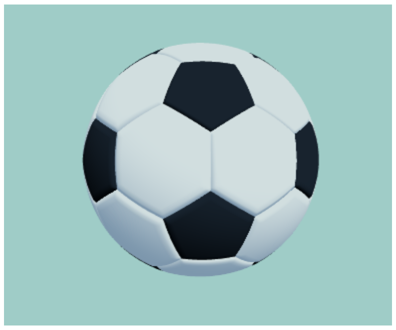
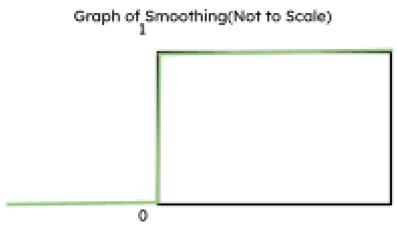
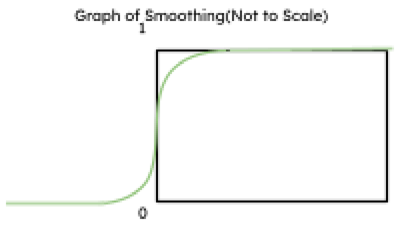
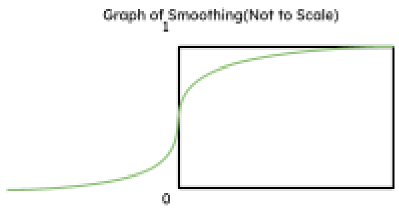


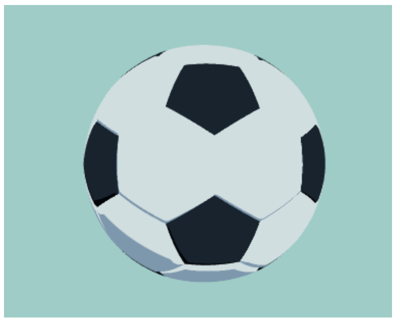
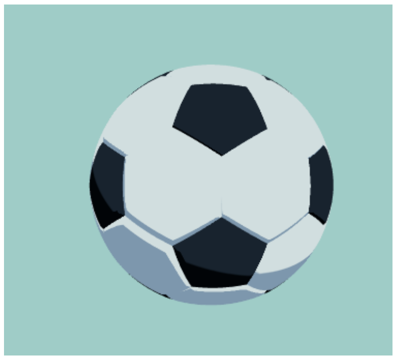
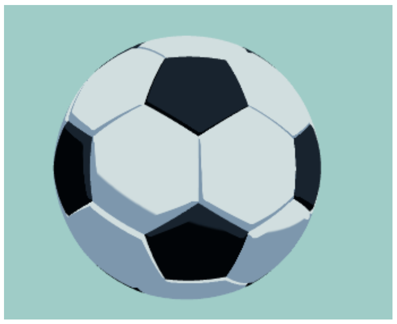
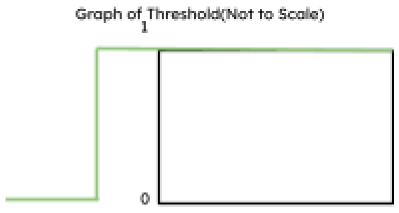
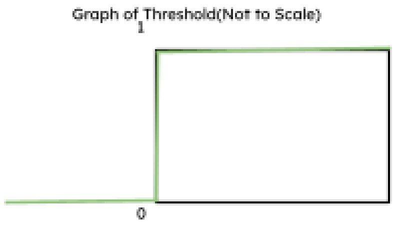
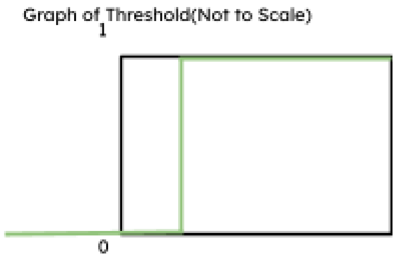
This looked pretty nice and cartoony. However, I also wanted to have a way to blend between smooth diffusing and cartoony. So, I looked into cgShader's smooth step function , which takes a step start and step end value, which sets values outside of these bounds to 0 or 1, but smoothens what's inside of these boundaries with an S curve.



https://planetside.co.uk/wiki/images/b/b5/Smoothstep.gif

I created a parameter to determine the amount of smoothing, with the center of the S curve located where the threshold is. I also created a parameter to control where the threshold lies.

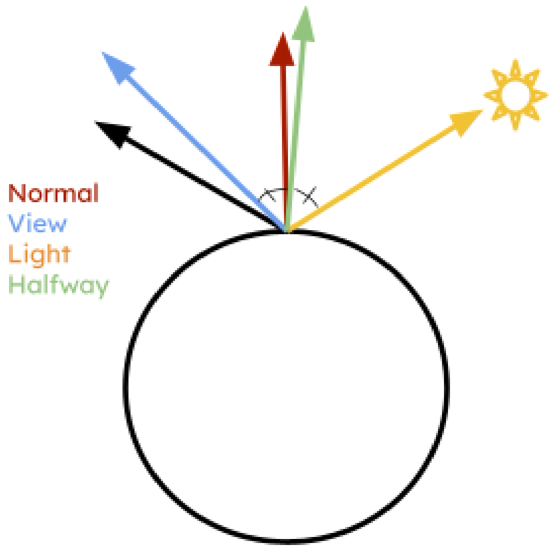
Smoothing examples		
No smoothing	Some smoothing	Fully Smoothed
		
		

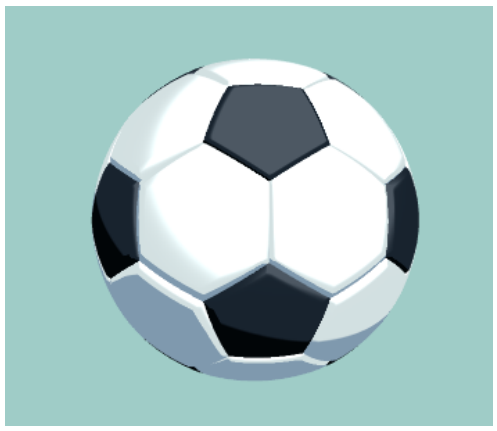
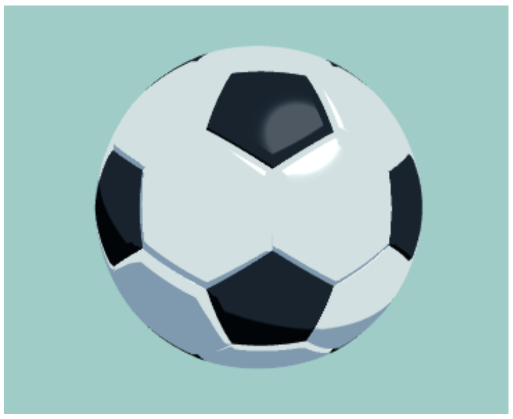
Varying Threshold examples		
Threshold: -0.25	Threshold: 0 (Normal)	Threshold: +0.25
		
		

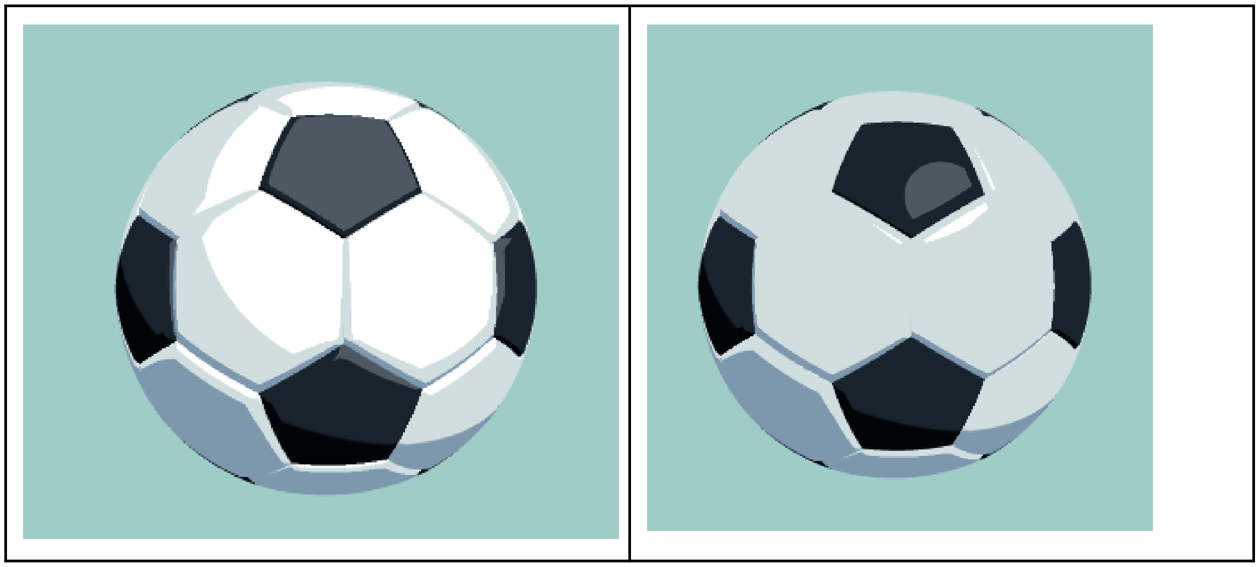
I use this method of adjusting threshold and smoothness values throughout the project, with specular highlights, and rim lighting.

Specular Highlights

To get specular highlights to appear, the view Vector needs to be taken into account. I use the Phong model, in which I obtain the dot product of the halfway vector by normalizing the view vector + light vector. I then raise this to the power of another parameter, the specular intensity, squared. This allows the user to adjust the size of the highlight. The resulting number, 0 - 1 is multiplied by the specular color, which is then added onto the diffuse color.



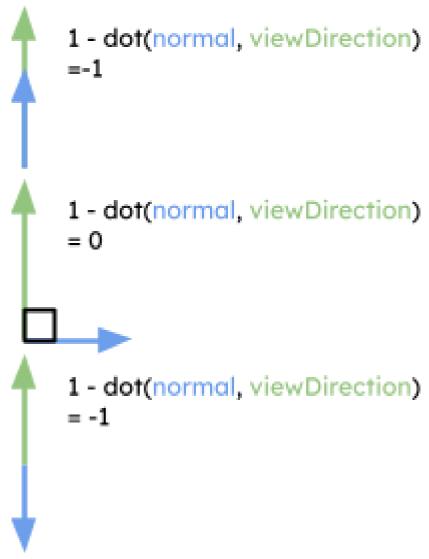
Examples of varying Specular Intensity	
Low specular intensity	High specular intensity
	



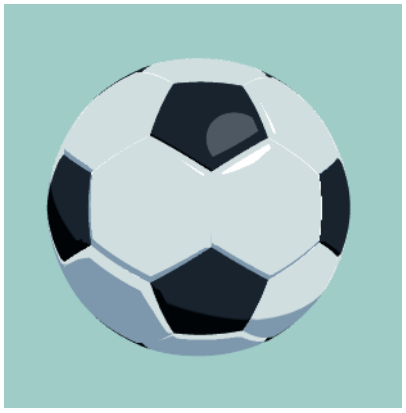
Rim Lighting

I implemented rim lighting, as it sort of tricks viewers into thinking that surrounding ambient light is being gently reflected off of the object. The goal is to lighten areas where the surface normal points perpendicularly to the view direction. This often illuminates the edges of the shape.

How I like to think about it is: it is similar to the diffuse shading, but inverted, using different vectors. In diffuse shading, as the angle of surface and lighting direction deviate, the lighting decreases; for rimlight, as the angle of surface and viewing direction deviate, the lighting increases. Therefore, 1 minus the dot product of the view vector and normal vector gives us this value.


1 - dot(normal, viewDirection)
=-1
1 - dot(normal, viewDirection)
= 0
1 - dot(normal, viewDirection)
= -1

This is what happens when implemented:



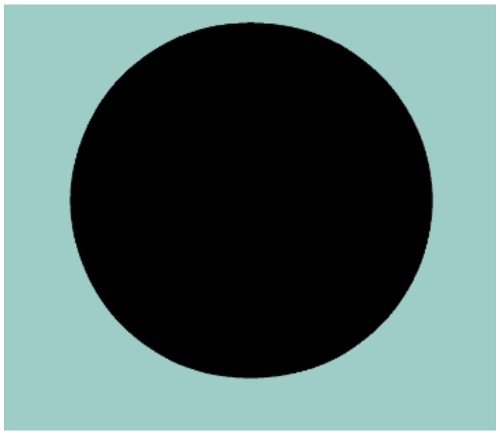
Notice that there is a VERY faint line along the rim, so there isn't really much rimlight. Think about it like this: if the rim light is composed of the "inverted" dot product of the view and normal, that means that the rim light only shows on surfaces facing away from the view vector(camera).

It is therefore necessary to increase this threshold for the smoothstep function. I created a parameter called “Rim amount”, which acts as that threshold. Increasing this value increases the amount of rim light that becomes visible to the camera.

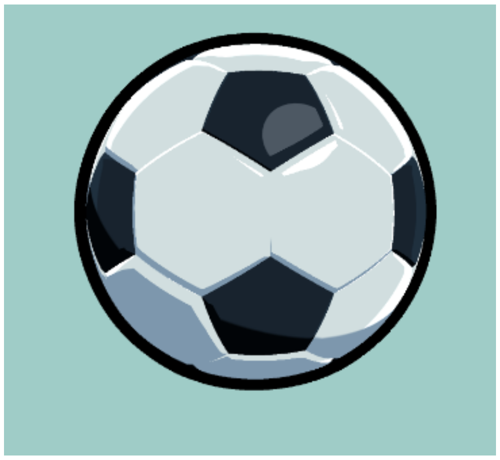
Like the diffuse and specular shading, the smoothness can also be changed via a parameter called “Rim Smoothness”.

Outline Shading

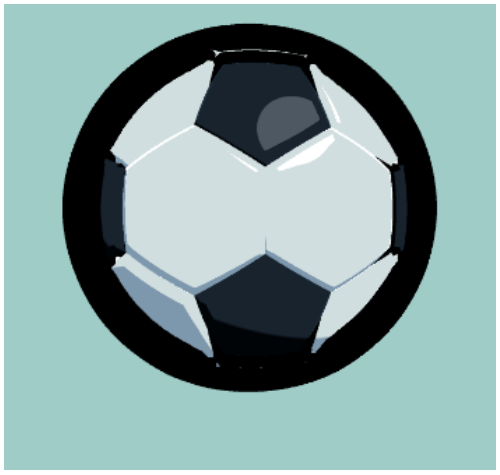
After completing diffuse, specular light, and rim light, I decided to move on to create an outline shader. This is different from the previously mentioned shaders, as it required that I edit both the fragment and vertex shaders(mostly the vertex shader however). My strategy was to make a copy of the object, inflate it in the vertex shader, shade it all black(or whatever outline color is wanted), and then lay it beneath the rendering pass.



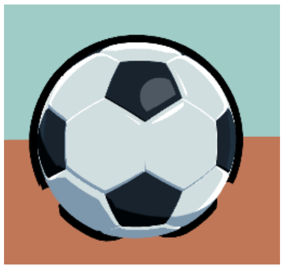
I “inflated” the vertices by increasing each vertex position by its normal times a scalar: the scalar being a new parameter named “outline size”. However, this clearly was not a proper border, as the actual colors were not in view.

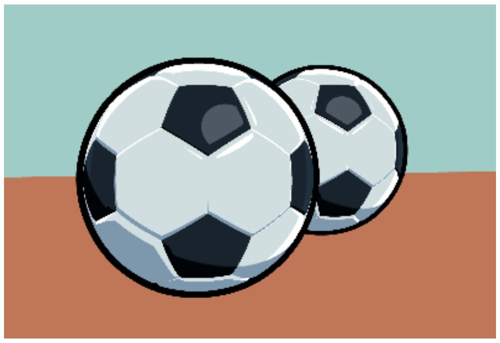


To fix this, I pushed each vertice’s z value back by a scalar(a parameter called “Outline Depth”) after the vertices were converted into camera space. This made sure that the outline was rendered underneath the diffuse, specular, and rimlight pass.

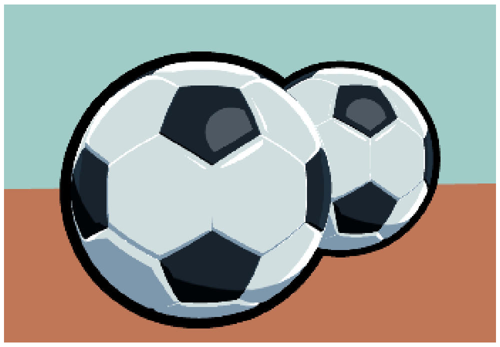


However, strange things would happen if the “Outline Depth” wasn’t large enough. So my solution was to make the outline depth default to a large number! But this varied from object to object, and caused outlines to not appear correctly sometimes.



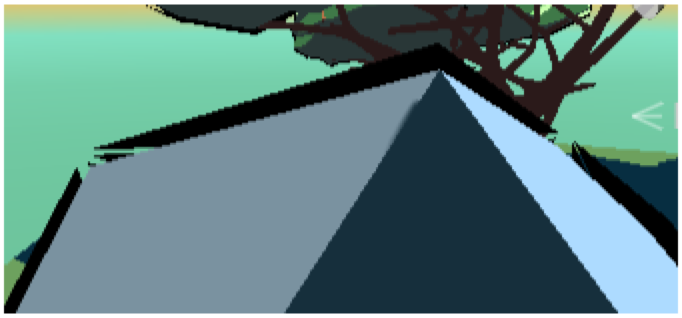


After doing a little research, I found out that unity shader passes gave me the option to cull front faces, and only render the back faces. This allowed me to use a small outline depth to still achieve the same results.



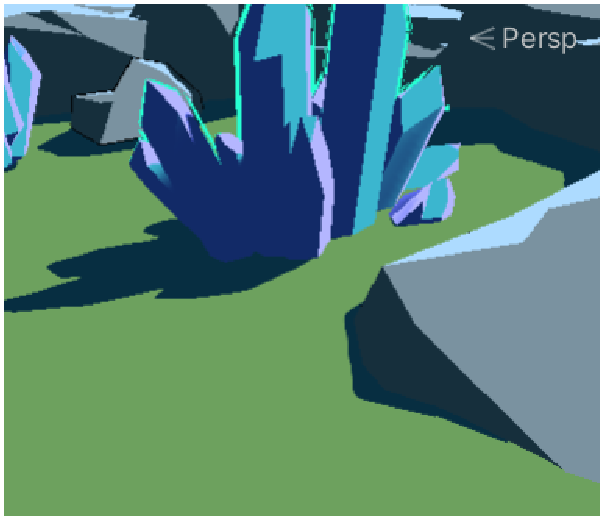
Can increasing Outline depth to merge borders

However, this outline shader is not perfect, as objects with sharper edges and lower vertice counts when expanded, tend to show a lot more seams. Furthermore, the outline size is dependent on the scale of the object, which makes border size inconsistent between objects of the same material with different scales. In the future, I plan to change the way vertices are extruded so they extrude along the average of vertices.

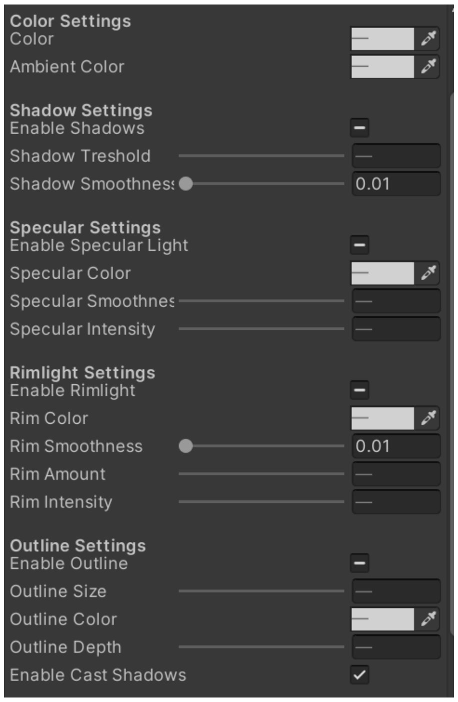


Cast Shadows

I decided I wanted shadows, so that I can render a whole scene with this shader without it looking awkward and flat. Though shadows were not the focus of this project, I implemented it, borrowing Unity’s "Legacy Shaders/VertexLit/SHADOWCASTER" shader to calculate the shadow attenuation(value 0-1 describing the shadow) for the fragment. I multiplied this value in with the diffuse calculations to do this.



An example of a cast shadow



Color Settings
Color
Ambient Color

Shadow Settings
Enable Shadows −
Shadow Treshold
Shadow Smoothness 0.01

Specular Settings
Enable Specular Light −
Specular Color
Specular Smoothness
Specular Intensity

Rimlight Settings
Enable Rimlight −
Rim Color
Rim Smoothness 0.01
Rim Amount
Rim Intensity

Outline Settings
Enable Outline −
Outline Size
Outline Color
Outline Depth
Enable Cast Shadows ✓

Customizable Interface

Challenges & What I learned

Challenges

- The outline shader was much more difficult than I had anticipated
- A lot of iteration was required for me to get the desired result
- Understanding the linear algebra behind shading

What I learned

- I have a way better understanding of how the phong and lambertian shading models actually work
- Much better understanding of object vs camera space
- Unity Shades & CgShader

Submission Details:

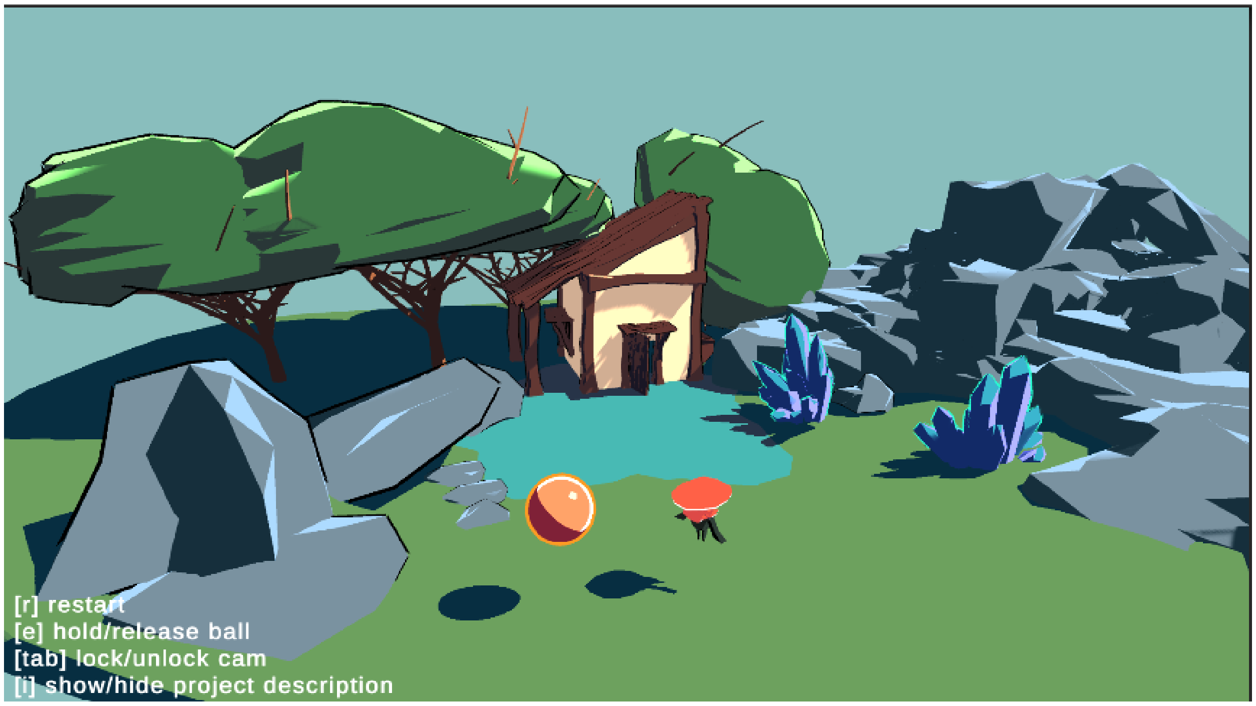
Github Repository of Entire Project: https://github.com/hungbuiwork/Toon-Shader

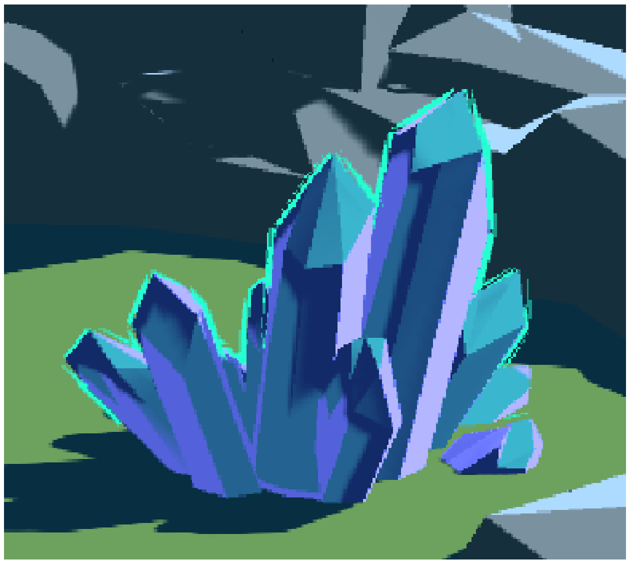
All Shader Code:

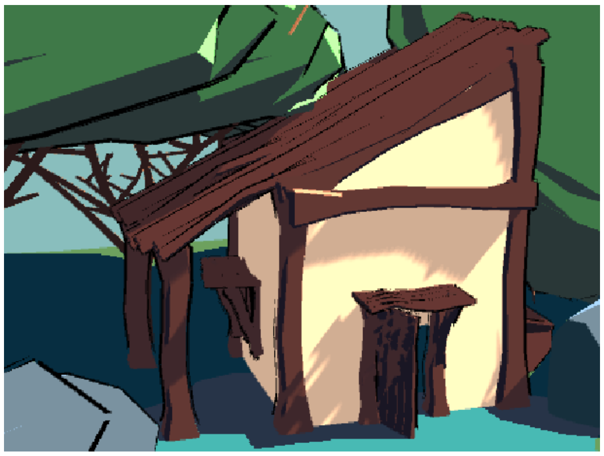
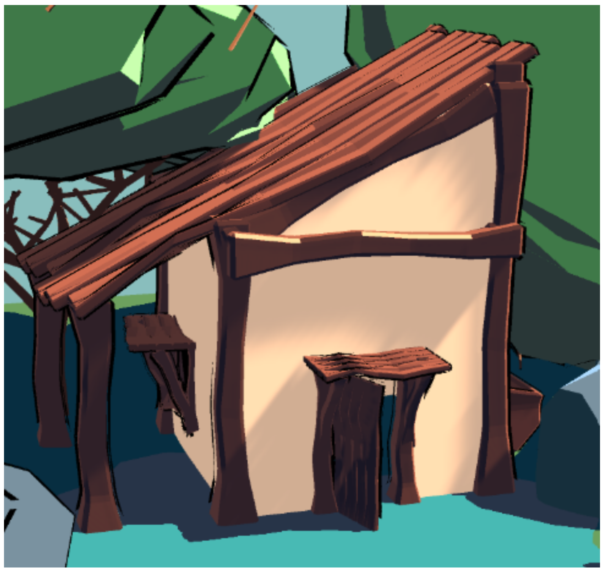
https://github.com/hungbuiwork/Toon-Shader/blob/main/Assets/ToonShader%20Materials/HBToon.shader

Screenshots of my Shader with different settings:

Sporo's Island, Fully Toonified


[r] restart
[e] hold/release ball
[tab] lock/unlock cam
[i] show/hide project description


Crystals, Smooth-ish diffuse

Diffuse-only, low smoothing, outline	Diffuse, Specular, Rimlight, high smoothing, outline
	

Soccer ball, high smoothing	Soccer Ball, No smoothing