Lab11
Rim Shader & Multi-Threading
Darrick Hilburn

Introduction

There are two things we will be looking at: rim shaders and multi-threading. Rim shaders are shaders that give lighting highlights around the outside edge of an object the shader is applied to and is only viewable on the front of the object, the front being where the light is hitting. Multi-threading is making the program operate on multiple threads, which can increase the program's efficieny when done correctly, and is required in game development to be able to fully utilize processing power. We will look at threading in a simplified manner for this lab.

Methods

We begin this lab by focusing on rim lighting.

This lab builds on the previous shader labs by including diffuse and specular lighting, along with Lambert lighting. With all of these implemented already, we can go on to calculate rim lighting. Two extra properties are needed for rim lighting: the rim color and rim power. Rim color is the color the outside rim will have, and rim power is how large the rim appears on the object. To calculate the rim lighting, we begin by calculating the rim. This is done by taking the dot of the view direction and normal direction. In this dot product, we normalize the view direction, which will keep values between zero and one. We then call the saturate function on the result to make sure the resulting value is constrained between zero and one. We then subtract this value from one, which inverts the lighting appearance, causing the center of the object to appear darker and the outside edge(s) of the object to appear brighter. With the object's rim calculated, we can calculate the rim lighting by taking the dot product of the normal direction and light direction, saturating this value to get a value between zero and one, then multiplying the result by the calculated rim to the power of the rim power property to get the rim lighting. We can then color the rim by multiplying the calculated lighting value by attenuation, the Unity supplied light color property, and our rim color property.

Next, we will look at multi-threading. For this lab, we write a program that uses two threads to show the concepts behind threading. Before any scripting is done, we must make sure that we are using the System.Threading namespace, which contains all Threading functions. The script has two class-level variables: a string and a bool. The string holds data the threads read from and write to, and the bool is used for stopping the threads. Two functions are written that write to the debug log which thread they are, modify the thread output string, then output the thread output string. A wait is also placed in the functions to simulate operations occuring in the thread. These functions are written using while loops to show how they process. A stop threads function is also written that changes the class-level bool variable to stop the threads from processing. With the functions written, in the Start function, we create the ThreadStart function calls to the appropriate threads, create the threads, then start the threads.

After observing the threads, we put locks in the threads inside the while loops around the functionality to observe what occurs. This causes the first thread to print repeatedly until the stop threads flag is true, then the second thread prints only once.

Conclusion

I have learned two major things in this lab: how to create rim lighting and how to multi-thread. More practice is required to better understand how these function, but this lab is a good foundation on their concepts. The Rim Lighting can be further explored to create toon shading effects for certain, and multi-threading must be further explored due to how modern computer architecture is designed since multi-threading gives access to the full processor power. The concept of locked and unlocked threading also makes sense, how a thread will run fully when it is locked down and threads may run in an erratic order when not locked.

Post-Lab

1. Display Thread 2 only displays once because while we were running Thread1, we had locked the program and told it while the stop threads boolean was false, it should process. When the stop threads function is invoked 10 seconds in, the boolean value changes to true, which stops the Thread1 functionality, and should stop the Thread2 functionality. However, because Thread2 is called to start before the Invoke function is called, the thread will run once before stopping because it's running on old data saying stop threads is false. The flag is still considered false to Thread2 because Thread1 locked all threads, making the stop flag value be held until Thread1 finished processing.