**Lab Report**

**Lab 11 - Shader and Multithreading**

**Craig Broskow**

**November 18, 2015**

Introduction:

This lab was different from our previous labs in that we explored two topics. First, we examined how to add rim lighting to our shaders in order to highlight the outlines of objects. Rim lighting is similar to the opposite of Lambert lighting: the rim's light is brighter as we get closer to the edge of the object.

In the second part of this lab, we started to explore some of the peculiarities of multi-threaded programming. Since computer processors determine the execution of low-level instructions, threads can execute in sequences that are difficult or impossible to predict because execution order may be dependent on factors outside the control of the programmer.

Methods:

Shader coding for this lab continued with the same approach we used for our last shader lab: a combination of CG (C for Graphics) code along with a "wrapper" of Unity's Shaderlab code to allow the shader to interact well with the Unity game engine. Again, rim lighting is somewhat similar, although opposite to, Lambert lighting.

We start by taking the dot product of the view vector and the normal vector. This gives us a gradient from the center to the silhouette of the object, but in the wrong direction. Therefore, we subtract the dot product from 1 in order to get the inverse effect. The result of that calculation effectively gives us the rim of the object.

In order to apply lighting to the rim, we use techniques very similar to Lambert lighting. We start with the dot product of the light vector and the normal vector. We then adjust that value by multiplying it by the rim factor previously calculated raised to a power specified by the designer. We finalize the calculation of the rim lighting by further multiplying the previous calculation by a designer-specified color, the light color and an attenuation factor. In the "On Your Own" problem for the shader section, we also add a color tint to the object as specified by the designer plus diffuse and specular lighting as shown in the second shader (called "RimShader1") in my "Lab 11" Unity project.

In the second part of this lab, we examine what happens when multiple threads are working with the same memory objects. Although we're working in Unity, we used the .NET library called "System.Threading" to implement threading because Unity itself is single-threaded. The common memory objects shared by our two threads are a string variable (called "\_threadOutput") and a boolean variable (called "\_stopThreads"). Having both of our threads read and write to those shared variables will result in the interesting and unpredictable effects from this threading portion of the lab.

The two methods that will be executed by our two threads are called "DisplayThread1" and "DisplayThread2". These threads simply display messages as long as the shared boolean variable \_stopThreads remains *false*. The Start method in the Unity project sets the \_stopThreads boolean variable to *true* after 10 seconds. In the inital version of this portion of the lab, we instantiate two threads in the Start method, and the results of this experiment show that the two threads step on each other's data and display debug messages that are unexpected based on the simple logic within the "DisplayThread" methods. Clearly, the two threads are changing shared memory without respect for what the other thread is expecting!

The specifics of starting threads is a little convoluted, and must follow a specific pattern to work correctly (and not crash the computer). First, delegates pointing to the threaded methods (DisplayThread1 and DisplayThread2) are assigned using the "ThreadStart" data object. (It is important to note that the references to the method names should **not** contain the standard instantiation parentheses in these statements.) Next, actual Thread objects are created that take the ThreadStart delegates as input parameters. Finally, the Thread objects are started using their "Start()" method calls.

Following this lesson on the unpredictable execution order of multiple threads, we also performed an "On Your Own" problem that executed four threads running three different methods. (The output of this experiment, which used the "OnYourOwn" scene and the "Threading2" script, were printed and attached to the end of this report.) Again, the results were inconsistent with might reasonably be expected by simply looking at the logic of the threaded methods. The threads executed in an unpredictable order, and two threads both set the value of a shared boolean variable...possibly because processor execution rules tend to favor Read operations over Write operations.

Conclusions:

Regarding the Shader portion of this lab, I will repeat the fact that I have little to no experience with computer graphics. Given that background, I still can't say that I completely understand the potential uses of custom shader programming. The effects of Rim Lighting are pretty cool looking, but I have no idea how much time and effort are normally used by game companies to write custom shaders...as opposed to using standardly available shaders. It would certainly take me some time to understand both the pros and cons of custom shaders, and also the consequences of making poor design decisions in their construction. It does appear that custom shaders can provide impressive results with relatively little coding, but I am concerned about knowing so little about their interaction with the wide variety of graphics hardware that they may run on...particularly on PC or mobile platforms.

The second portion of this lab, which dealt with multi-threaded programming, certainly demonstrated the potential pitfalls of using that technology without fully understanding the unpredictable nature of computer processor execution order. While the simultaneous use of multiple processors, core and threads certainly sounds like a great idea, the techniques used to control memory storage and execution order (such as locks and memory barriers) effectively remove at least some of the efficiency advantages of multi-threading. This portion of the lab was obviously dedicated to showing some potential problems with multi-threading, but also demonstrated the use of thread locks as a beginning technique for preventing some of multi-threading's disadvantages.

Postlab Question:

1. Why does Display Thread 2 and Thread 2 Output --> Hello Thread 2 display once after Display Thread 1 and Thread 1 Output --> Hello Thread 1?

The DisplayThread1 method acquires and re-acquires its lock on the class so quickly that the DisplayThread2 method never has an opportunity to acquire its own lock until DisplayThread1 stops due to the value of boolean variable \_stopThreads being set to *true*. However, DisplayThread2 is already inside its while loop [i.e., "while(\_stopThreads == false)"] waiting to acquire a lock. When DisplayThread2 gets its lock, it executes its Debug.Log statements **once**...after which it drops out of the while loop because it recognizes that the value of \_stopThreads is now *true*.

On Your Own Problems:

For the "On Your Own" problem in the first part of this lab, the "Lab 11" Unity project contains a second capsule called "RimCapsule1", which uses a material called "RimMaterial1" and a custom shader called "RimShader1". This second shader demonstrates the calculation of the final light including diffuse, specular, and a custom color tint specified by the user to apply to the capsule itself.

For the "On Your Own" problem in the second part of this lab, the "Lab 11B" Unity project contains a scene called "OnYourOwn" and a script called "Threading2.cs". The Main Camera in the OnYourOwn scene calls the Threading2 script and provides the solution to this problem. The output from the program is attached as the last page of this report.

Code:

This lab uses two Unity projects: "Lab 11" and "Lab 11B". The two Shader scripts below are for the "Lab 11" Unity project, while the two C# scripts below are for the "Lab 11B" Unity project.

// RimShader.shader

Shader "Custom/RimShader"

{

Properties //Properties block that the user can adjust

{ //Interfaces with Unity Inspector

\_Color("Color",Color) = (1.0, 1.0, 1.0, 1.0)

\_SpecColor("Specular Color", Color) = (1,1,1,1)

\_Shininess("Shininess", Float) = 10

\_RimColor("Rim Color", Color) = (1,1,1,1)

\_RimPower("Rim Power",Range(0.1,10)) = 3.0

}

SubShader

{

Pass

{

Tags{ "LightMode" = "ForwardBase" }

CGPROGRAM

#pragma vertex vertexProgram

#pragma fragment fragmentProgram

//user defined variables

uniform float4 \_Color;

uniform float4 \_SpecColor;

uniform float \_Shininess;

uniform float4 \_RimColor;

uniform float \_RimPower;

//Unity defined Variables

uniform float4 \_LightColor0;

struct vertexInput

{

float4 vertex : POSITION;

float3 normal : NORMAL;

};

struct vertexOutput

{

float4 pos : SV\_POSITION;

float4 posWorld : TEXCOORD0;

float3 normalDir : TEXCOORD1;

};

vertexOutput vertexProgram(vertexInput input)

{

vertexOutput output;

output.posWorld = mul(\_Object2World, input.vertex);

output.normalDir = normalize(float3(mul(float4(input.normal, 0.0), \_World2Object).xyz));

output.pos = mul(UNITY\_MATRIX\_MVP, input.vertex);

return output;

}

float4 fragmentProgram(vertexOutput input) : COLOR

{

float3 normalDirection = normalize(input.normalDir);

float3 viewDirection = normalize(\_WorldSpaceCameraPos.xyz - float3(input.posWorld.xyz));

float3 lightDirection;

float attenuation = 1.0;

lightDirection = normalize(float3(\_WorldSpaceLightPos0.xyz));

//Lighting

float3 ambientLight = UNITY\_LIGHTMODEL\_AMBIENT.rgb;

//Lighting - Diffuse

//This is one line or multiple lines. If you prefer it one-line, delete the uncommented lines.

//If you prefer it multiple lines, delete the commented lines

//~~~~~~~~~~~~~~~ ONE-LINE ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//float3 diffuseLighting = attenuation \* float3(\_LightColor0.rgb) \* max(0.0, dot(normalDirection, lightDirection));

//~~~~~~~~~~~~~~~ MULTIPLE LINES ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

float3 diffuseLighting = dot(normalDirection, lightDirection);

diffuseLighting = max(0.0, diffuseLighting);

diffuseLighting = float3(\_LightColor0.rgb) \* diffuseLighting;

diffuseLighting = attenuation \* diffuseLighting;

//Lighting - Specular

//This is one line or multiple lines. If you prefer it one-line, delete the uncommented lines.

//If you prefer it multiple lines, delete the commented lines

//~~~~~~~~~~~~~~~ ONE-LINE ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//float3 specularLighting = max(0.0, dot(normalDirection, lightDirection)) \* attenuation \* float3(\_LightColor0.rgb) \* float3(\_SpecColor.rgb) \* pow(max(0.0, dot(reflect(-lightDirection, normalDirection), viewDirection)), \_Shininess);

//~~~~~~~~~~~~~~~ MULTIPLE LINES ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

float3 specularLighting = reflect(-lightDirection, normalDirection);

specularLighting = dot(specularLighting, viewDirection);

specularLighting = max(0.0, specularLighting);

specularLighting = pow(specularLighting, \_Shininess);

//This controls the colouring of the specular light,

//As well as stopping it from going to the backside of the object

//AND adding the specular lighting falloff.

float3 specularColouring = dot(normalDirection, lightDirection);

specularColouring = max(0.0, specularColouring);

specularColouring = attenuation \* specularColouring;

specularColouring = float3(\_SpecColor.rgb) \* specularColouring;

specularLighting = specularColouring \* specularLighting;

//Rim Lighting

float3 actualRim = 1 - saturate(dot(normalize(viewDirection), normalDirection));

float3 rimLighting = attenuation \* \_LightColor0.rgb \* \_RimColor \*

saturate(dot(normalDirection, lightDirection)) \* pow(actualRim, \_RimPower);

//Final Lighting

float3 finalLight = rimLighting;

//Test Lighting

return float4(finalLight, 1.0);

}

ENDCG

}

}

}

// RimShader1.shader

Shader "Custom/RimShader1"

{

Properties //Properties block that the user can adjust

{ //Interfaces with Unity Inspector

\_Color("Color",Color) = (1.0, 1.0, 1.0, 1.0)

\_SpecColor("Specular Color", Color) = (1,1,1,1)

\_Shininess("Shininess", Float) = 10

\_RimColor("Rim Color", Color) = (1,1,1,1)

\_RimPower("Rim Power",Range(0.1,10)) = 3.0

}

SubShader

{

Pass

{

Tags{ "LightMode" = "ForwardBase" }

CGPROGRAM

#pragma vertex vertexProgram

#pragma fragment fragmentProgram

//user defined variables

uniform float4 \_Color;

uniform float4 \_SpecColor;

uniform float \_Shininess;

uniform float4 \_RimColor;

uniform float \_RimPower;

//Unity defined Variables

uniform float4 \_LightColor0;

struct vertexInput

{

float4 vertex : POSITION;

float3 normal : NORMAL;

};

struct vertexOutput

{

float4 pos : SV\_POSITION;

float4 posWorld : TEXCOORD0;

float3 normalDir : TEXCOORD1;

};

vertexOutput vertexProgram(vertexInput input)

{

vertexOutput output;

output.posWorld = mul(\_Object2World, input.vertex);

output.normalDir = normalize(float3(mul(float4(input.normal, 0.0), \_World2Object).xyz));

output.pos = mul(UNITY\_MATRIX\_MVP, input.vertex);

return output;

}

float4 fragmentProgram(vertexOutput input) : COLOR

{

float3 normalDirection = normalize(input.normalDir);

float3 viewDirection = normalize(\_WorldSpaceCameraPos.xyz - float3(input.posWorld.xyz));

float3 lightDirection;

float attenuation = 1.0;

lightDirection = normalize(float3(\_WorldSpaceLightPos0.xyz));

//Lighting

float3 ambientLight = UNITY\_LIGHTMODEL\_AMBIENT.rgb;

//Lighting - Diffuse

//This is one line or multiple lines. If you prefer it one-line, delete the uncommented lines.

//If you prefer it multiple lines, delete the commented lines

//~~~~~~~~~~~~~~~ ONE-LINE ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//float3 diffuseLighting = attenuation \* float3(\_LightColor0.rgb) \* max(0.0, dot(normalDirection, lightDirection));

//~~~~~~~~~~~~~~~ MULTIPLE LINES ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

float3 diffuseLighting = dot(normalDirection, lightDirection);

diffuseLighting = max(0.0, diffuseLighting);

diffuseLighting = float3(\_LightColor0.rgb) \* diffuseLighting;

diffuseLighting = attenuation \* diffuseLighting;

//Lighting - Specular

//This is one line or multiple lines. If you prefer it one-line, delete the uncommented lines.

//If you prefer it multiple lines, delete the commented lines

//~~~~~~~~~~~~~~~ ONE-LINE ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//float3 specularLighting = max(0.0, dot(normalDirection, lightDirection)) \* attenuation \* float3(\_LightColor0.rgb) \* float3(\_SpecColor.rgb) \* pow(max(0.0, dot(reflect(-lightDirection, normalDirection), viewDirection)), \_Shininess);

//~~~~~~~~~~~~~~~ MULTIPLE LINES ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

float3 specularLighting = reflect(-lightDirection, normalDirection);

specularLighting = dot(specularLighting, viewDirection);

specularLighting = max(0.0, specularLighting);

specularLighting = pow(specularLighting, \_Shininess);

//This controls the colouring of the specular light,

//As well as stopping it from going to the backside of the object

//AND adding the specular lighting falloff.

float3 specularColouring = dot(normalDirection, lightDirection);

specularColouring = max(0.0, specularColouring);

specularColouring = attenuation \* specularColouring;

specularColouring = float3(\_SpecColor.rgb) \* specularColouring;

specularLighting = specularColouring \* specularLighting;

//Rim Lighting

float3 actualRim = 1 - saturate(dot(normalize(viewDirection), normalDirection));

float3 rimLighting = attenuation \* \_LightColor0.rgb \* \_RimColor \*

saturate(dot(normalDirection, lightDirection)) \* pow(actualRim, \_RimPower);

//Final Lighting

float3 finalLight = (rimLighting + ambientLight + diffuseLighting + specularLighting) \* float3(\_Color.rgb);

//Test Lighting

return float4(finalLight, 1.0);

}

ENDCG

}

}

}

// Threading.cs

using UnityEngine;

using System.Collections;

using System.Threading;

public class Threading : MonoBehaviour {

//Variable that both threads will access

private string \_threadOutput = "";

//Bool to stop threads from running

private bool \_stopThreads = false;

/// <summary>

/// Loop thread 1 continuously

/// Assign the thread information into \_threadOutput

/// </summary>

void DisplayThread1()

{

while(\_stopThreads == false)

{

lock(this) //Lock on the CURRENT instance of the CLASS for thread#1

{

Debug.Log ("Display Thread 1");

\_threadOutput = "Hello Thread1";

//Put the thread to sleep (stop processing) for 1000 ms

//This is to simulate a lot of processing and calculations that

//would normally occur in a thread

Thread.Sleep(1000);

Debug.Log ("Thread 1 Output --> " + \_threadOutput);

}//Release the lock for thread 1 here

}

}

/// <summary>

/// Loop thread2 continuously

/// Assign the thread information into \_threadOutput

/// </summary>

void DisplayThread2()

{

while(\_stopThreads == false)

{

lock(this)//Lock on the CURRENT instance of the CLASS for thread#2

{

Debug.Log ("Display Thread 2");

\_threadOutput = "Hello Thread2";

//Put the thread to sleep (stop processing) for 1000 ms

//This is to simulate a lot of processing and calculations that

//would normally occur in a thread

Thread.Sleep(1000);

Debug.Log ("Thread 2 Output --> " + \_threadOutput);

}//Release the lock for thread 2 here

}

}

/// <summary>

/// Triggers to flag to stop threads from running

/// This is so unity doesn't lock up!

/// </summary>

void StopThreads()

{

\_stopThreads = true;

}

// Use this for initialization

void Start () {

//Invoke the stop threads after 10 seconds so you can review the results

Invoke ("StopThreads", 10);

//Create two new threads starts that points to the functions

//that the threads will be running

ThreadStart firstThread = new ThreadStart (DisplayThread1);

ThreadStart secondThread = new ThreadStart (DisplayThread2);

//Create the two threads

Thread thread1 = new Thread (firstThread);

Thread thread2 = new Thread (secondThread);

//Start the two threads

thread1.Start ();

thread2.Start ();

}

// Update is called once per frame

void Update () {

}

}

// Threading2.cs

using UnityEngine;

using System.Collections;

using System.Threading;

public class Threading2 : MonoBehaviour {

private bool done = false;

void Function1()

{

for (int i = 0; i < 10; i++)

{

Debug.Log("x");

}

}

void Function2()

{

for (int i = 0; i < 10; i++)

{

Debug.Log("y");

}

}

void Function3()

{

if (!done)

{

Debug.Log("Done");

done = true;

}

}

// Use this for initialization

void Start () {

//Create four new threads starts that points to the functions

//that the threads will be running

ThreadStart firstThread = new ThreadStart (Function1);

ThreadStart secondThread = new ThreadStart (Function2);

ThreadStart thirdThread = new ThreadStart (Function3);

ThreadStart fourthThread = new ThreadStart (Function3);

//Create the four threads

Thread thread1 = new Thread (firstThread);

Thread thread2 = new Thread (secondThread);

Thread thread3 = new Thread (thirdThread);

Thread thread4 = new Thread (fourthThread);

//Start the four threads

thread1.Start ();

thread2.Start ();

thread3.Start ();

thread4.Start ();

}

// Update is called once per frame

// void Update () {

//

// }

}

**Attached below is the printed output from the "On Your Own" problem for the second part of this lab.**

