Lab 11 – Shader and MultiThreading

# Hands-On

In the hands-on shader unit the student will explore creating more complex shaders. These shaders will be written primarily in C for Graphics (Cg), with a little bit of Unity’s built-in Shaderlab.

## Section Three: Rim Lighting

In this section of the shader unit, the student will look at how to accomplish a rim lighting effect on their shaders. This will allow light to give highlights to the outline of the object.

### Step One: Setup

For the setup for this lab, you will need to create a new unity project (or open the unity project you have been using for in-class examples). You will then need a capsule, a material, and a shader. Keep your naming consisting, and name them something along the lines of RimShader, RimMaterial, RimCapsule.

Assign the material to the capsule, and assign the shader to the material.

Download the Shader Template provided. The shader template includes the lambert AND specular shader inside, and is ready for the rim shader to be added. Copy the cody inside the shader template and paste it into your rim shader.

Change the name of the shader to something more appropriate.

Look over the shader to verify that you understand what is happening!

Finally, open the shader in MonoDevelop or your preferred IDE.

### Step Two: Theory

To get the rim lighting you will effectively be doing the opposite of the lambert lighting. In the lambert lighting, the shading gets dimmer as it gets further from the light. The rim light get’s brighter as it gets closer to the edge.

You will then use the dot product of the view direction and the normal direction (similar to the specular highlights) so that you only get rim in the direction towards the light).



Notice the subtle pinkish-red outline on the capsule. This does NOT extend to the back of the object, it is only around the silhouette of the object as seen by the camera (player).

Creating this silhouette is done by creating a rim of light around the model based on the view direction, and then masking it out with the light direction.

With rim lighting, you only have two new lines of code (this can be done in one, but it is easier to understand if broken into two).

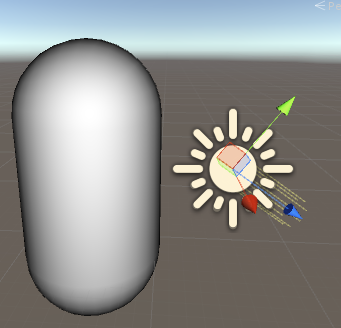
half objectRim = 1 – saturate(dot(normalize(viewDirection), normalDirection));

float3 rimLighting = max(0.0, dot(normalDirection,lightDirection) \* \_RimColor \* pow(rim,\_RimPower));

**Breaking this down:**

**float3 objectRim = dot(viewDirection, normalDirection)**

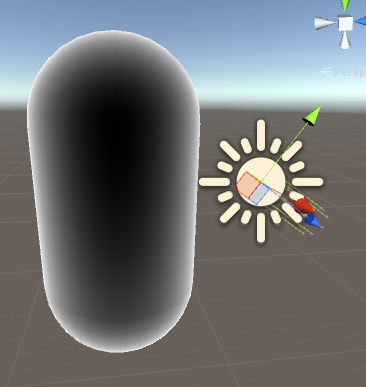
First you create the outline by using the dot product of the view and normal directions. If you remember the dot product, this means that as your view direction faces the same direction as the normal direction it will be one. When it is facing away, it will be zero. By doing this you get a gradient falloff from the center (where the normal is) to the outside (where the edge is!).



**half objectRim = 1 – saturate(dot(normalize(viewDirection), normalDirection));**

The goal isn’t to get the light to be black (0) around the edges, but for it to be white (1) around the edges. We want the light to be bright around the rim of the object.

To do this, you just want to invert the values. The easiest way to do this is to say 1 – dot(viewDirection, normalDirection).



Finally, there is some touch-up, such as making sure that the values are never less than zero or greater than one (as you have been doing with every shader thus-far).

To do this, you can start by normalizing the view direction, bringing the values of viewDirection between 0 and 1. Then you saturate the dot product to remove all values less than zero and greater than 1. You can use the max(0.0, dot(normalize(viewDirection), normalDirection)) as well if you want instead of saturate.

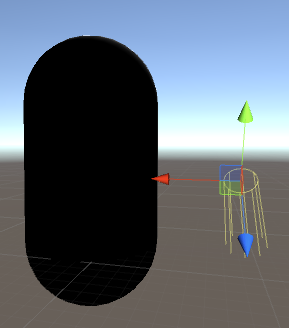
**float3 rimLighting = pow(objectRim, \_RimPower);**

To add control over how sharp the rim light is, you can use the pow function like you did for the specular lighting.

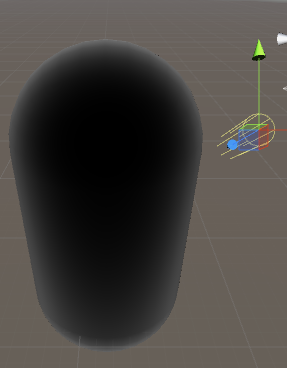




float3 rimLighting = max(0.0, dot(normalDirection,lightDirection) \* pow(objectRim,\_RimPower));

To make the rim only show on the same side as the light (not to show on the backside) you multiply it by the dot product of the normal and the light direction. The same was as was done in the specular, as well as originally in the lambert shader.

(Notice the camera is viewing the object on the opposite side from where the light is hitting the object and there is no rim)

(Notice the camera is viewing the object at the exact angle that the light is hitting the object and the rim is very apparent)

**float3 rimLighting = max(0.0, dot(normalDirection, lightDirection) \* \_RimColor \* pow(objectRim, \_RimPower));**

Finally you can add a basic colour tint to the rim, so that the rim can be any colour the designer desires!



### Step Three: Coding

With the theory covered, you can now start to write this shader. As in class, most of the theory will be covered again as you go through writing the code for this shader.

Start off by using the Shader Template provided:

|  |
| --- |
| Shader "RimLightingWithLightEffect" //Name  {  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      //Final Lighting  float3 finalLight = (ambientLight + diffuseLighting +  specularLighting) \* float3(\_Color.rgb);  //Test Lighting  return float4(finalLight, 1.0);  }    ENDCG  }  }  } |

With the template imported, you can start to work on the rim lighting. The first thing you will do is declare a float rimLighting (this will store all the rim lighting information, similar to specularLighting and diffuseLighting).

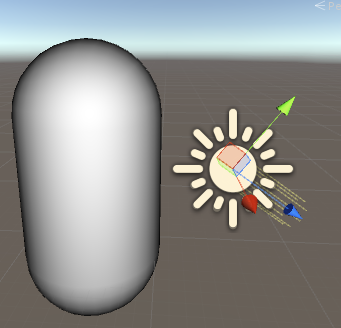
Also, change the finalLight to output the rimLighting so you can see your changes to the shader.

|  |
| --- |
| //Rim Lighting  float rimLighting =  //Final Lighting  float3 finalLight = rimLighting;  return float4(finalLight, 1.0); |

The first step in calculating rim lighting is to get the angle between the view direction and the normal direction. Since the normal direction is the direction directly away from the surface of the model, and the view direction is how you are looking at the model you can use this angle to determine how bright/dark the model should be. With rim lighting you typically want everything to be based off of the player’s view. You want the rim to be lit up depending on where the user is viewing the rim of the object, the silhouette.

By finding the angle between the surface normal and the view direction, you can pinpoint how far away from the normal the “rim” is. This is the first step.

|  |
| --- |
| //Rim Lighting  float rimLighting = dot(viewDirection, normalDirection); |



The next thing you will want to do (just for safety) is to normalize the viewDirection – remember if you are working with lighting you typically always want the values between 0 and 1. By normalizing the view direction you can keep the values between 0 and 1 neatly. You aren’t going to see much of a difference on your model when you normalize it, but artifacts **can** happen if you don’t.

|  |
| --- |
| //Rim Lighting  float rimLighting = dot(normalize(viewDirection), normalDirection); |

The next thing you want to do is again force the resulting dot product to be between 0 and 1.

You have been using max(0.0, dot(…)) to accomplish this. You can use that here, but you can also use the saturate function as a short-hang to the max(0.0, dot(…));

|  |
| --- |
| //Rim Lighting  float rimLighting = saturate(dot(normalize(viewDirection), normalDirection)); |

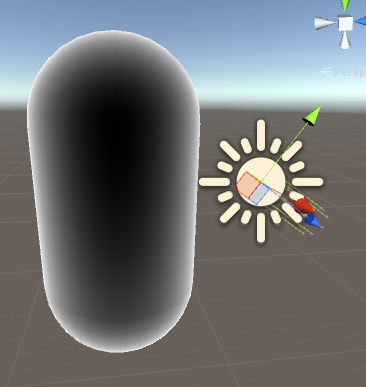
Again, you aren’t going to see much of a difference in your shader at this point. This is just to prevent future problems should they arise.

When explaining what rim lighting is, it was said that rim lighting **highlights** the silhouette of your object. However, with your current object you are actually darkening the silhouette, not highlighting it. You are getting brighter towards the center and darker towards the outline of the object.

To solve this, you can simply invert the value of your dot product. In other words, where it is 1 (black) it should be 0 (white). Where it is 0 (white) it should be 1 (black) or -1 (also black).

To invert this simply and quickly, you can subtract the result of the dot product from 1. Since you are gaurenteed the dot product will contain a value of 0 to 1 (thanks to the saturate) you can rest assured that by doing 1 – dotProduct you will get the inverted value.

|  |
| --- |
| //Rim Lighting  float rimLighting = 1 - saturate(dot(normalize(viewDirection), normalDirection)); |



The result of this is the actual rim of your object (the white parts). This means that you need to rename your rimLighting to something like objectRim or actualRim or whatever makes sense for you to notate that the resulting value is the actual rim of the object, not the rimLighting itself!

|  |
| --- |
| //Rim Lighting  float actualRim = 1 - saturate(dot(normalize(viewDirection), normalDirection)); |

To start lighting this, you will need a float4 (because you are going to be working with colours and the sort – this will be the actual rim LIGHTING instead of just the rim itself) named rimLighting.

|  |
| --- |
| //Rim Lighting  float actualRim = 1 - saturate(dot(normalize(viewDirection), normalDirection));  float4 rimLighting = |

The first thing you will want to do is take the dot product of the normal direction and the light direction, just as in the lambert shader.

|  |
| --- |
| //Rim Lighting  float actualRim = 1 - saturate(dot(normalize(viewDirection), normalDirection));  float4 rimLighting = dot(normalDirection, lightDirection);  //Final Lighting  float3 finalLight = rimLighting; |



This should look very familiar, it is just the lambert shader you first created (nothing fancy happening here). You will want to make sure that you normalize and control the minimum value (via max(0.0 , dot(…)) or saturate()) to stop anything from appearing on the backside of the object.

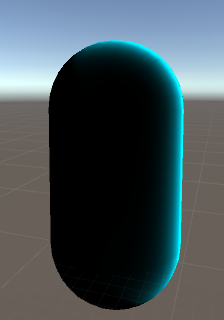
Next you will want to apply the Rim ontop of the lambert. If you remember in the specular shader there was an arbitrary number that controlled the strength of the specular lighting. With rim lighting you have that as well, and it is a property that is defined in the shader template named \_RimPower. It is just an arbitrary float that the designer can adjust to change the strength of the rim light. As with the specular light, you adjust the strength of the rim lighting with the pow function.

|  |
| --- |
| //Rim Lighting  float actualRim = 1 - saturate(dot(normalize(viewDirection), normalDirection));  float4 rimLighting = saturate(dot(normalDirection, lightDirection)) \*  pow(actualRim, \_RimPower); |



The next step is to tint the rim. You should be familiar with tinting shaders at this point. You will want to multiply your rimLighting by the colour that you want to tint it with. You can also add in the light color (and attenuation) to make the colour of the light affect the rim!

|  |
| --- |
| //Rim Lighting  float actualRim = 1 - saturate(dot(normalize(viewDirection), normalDirection));  float4 rimLighting = attenuation \* \_LightColor0.rgb \* \_RimColor \*  saturate(dot(normalDirection, lightDirection)) \* pow(actualRim, \_RimPower); |



### Wrap Up

You now have the ability to work with rim lighting on an object, and make the rim lighting dependent on how the user is viewing the object as well as how the light is shining on the object.

#### Think About It!

### On Your Own!

1. Calculate the light final to include the diffuse, specular, and a color tint that the user specifies to tint the object itself (not the rim!)

# Hands-On

In the threading unit, the student will get hands-on experience with starting and stopping multiple threads in a c# program. Although unity will be used to facilitate this, all threading-specific code will be utilized using c#’s System.Threading API.

## Section One: Starting To Thread

In this section, the student will examine simple threading implementations, as well as a view some of the pitfalls of threading.

### Step One: Setup

To start this lab, you will need a unity project and a single script. You can name this script whatever you want.

Attach the script to your main camera.

Open the script in your preferred IDE.

### Step Two: Helper Functions

Creating a thread is a rather simple process. It is almost too easy. Keep in mind that while the act of threading is very simplistic and very easy, threading properly is incredibly complex.

Let's take a look at a two-threaded scenario. In our example, we will create two threads in C#: Thread 1 and Thread 2, both running in their own while loop. The threads won't do anything useful, they will just print out a message saying which thread they are part of. We will utilize a shared memory class member called \_threadOutput. \_threadOutput will be assigned a message based upon the thread in which it is running.

The first thing you need to do is declare the using statement, using System.Threading.

Next, define two private variables, the first one a string and the second one a bool.

|  |
| --- |
| public class Threading : MonoBehaviour {  //Variable that both threads will access  private string \_threadOutput = "";  //Bool to stop threads from running  private bool \_stopThreads = false; |

Next, you will define two functions. Each function will eventually run on a separate thread. You can call these something simple, such as DisplayThread1 and DisplayThread2. The only thing you will be doing in these functions is continuously loop until \_stopThreads is false, displaying some information into the debug.

While doing this however, you will adjust the \_threadOutput variable, assigning into it a string that represents what thread is running (such as “Thread 1!”).

|  |
| --- |
| /// <summary>  /// Loop thread 1 continuously  /// Assign the thread information into \_threadOutput  /// </summary>  void DisplayThread1()  {  while(\_stopThreads == false)  {  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);  }  } |

Again, in this function you are continuously running while \_stopThread is not true. You will then debug out that you are going to display the \_threadOutput while inside the first thread.

You adjust \_threadOutput to display that you are in thread1.

Next, you pause the thread for a moment. This is to simulate intense calculations that are usually going on inside of threads.

After your thread has done it’s invisible calculations, it displays what \_threadOutput contains (the string variable that was declared earlier).

The next function (DisplayThread2) does the exact same thing, except it should have a 2 where the 1’s are in the previous function:

|  |
| --- |
| /// <summary>  /// Loop thread 1 continuously  /// Assign the thread information into \_threadOutput  /// </summary>  void DisplayThread1()  {  while(\_stopThreads == false)  {  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);  }  }  /// <summary>  /// Loop thread2 continuously  /// Assign the thread information into \_threadOutput  /// </summary>  void DisplayThread2()  {  while(\_stopThreads == false)  {  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);  }  } |

Finally, you will create a function to set \_stopThreads to true after 10 seconds:

|  |
| --- |
| /// <summary>  /// Triggers to flag to stop threads from running  /// This is so unity doesn't lock up!  /// </summary>  void StopThreads()  {  \_stopThreads = true;  } |

### Step Three: Starting Threads

Next (inside of the Start function) you will create your threads.

Again, this is very simple to do! (Yay!).

The first thing to understand is that threads can be created multiple ways. You will start by creating a thread using a delegate. This delegate is called ThreadStart. ThreadStart will allow you to start a thread, and then assign what function the thread will compute.

The next thing you will do is create the threads themselves, assigning them the ThreadStart delegate that you setup earlier.

Finally, you will start the threads on their work.

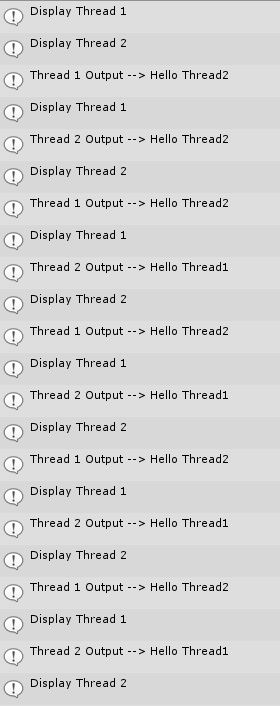
|  |
| --- |
| // Use this for initialization  void Start () {  //Invoke the stop threads after 10 seconds so you can review the results  Invoke ("StopThreads", 10);  //Create two new threads starts that points to the functions  //that the threads will be running  ThreadStart firstThread = new ThreadStart (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 ();    } |

NOTE: Where the new ThreadStarts are created, it is NOT an accident that DisplayThread1 is not called as a method! There are no parameters being passed. Do **NOT** include () when assigning the method to the delegate! ***ALSO the Invoke MUST come before starting the threads (if you do not heed this advice then you may crash your unity and your computer).***

The next thing for you to do is run unity!

Make sure when you run unity you do so with Collapse turned OFF.

You should get something in your console similar to the following:



Your mileage may vary (meaning it might not be EXACTLY the same as the one shown above). The results of this code is shown in figure 2. Look carefully at the results. You will notice that the program gives some surprising output(if you were looking at this from a single-threaded mindset). Although you clearly assigned \_threadOutput to a string with a number corresponding to the thread to which it belongs, that is not what it looks like in the console.

You would expect to see the following from your code

**Thread 1 Output --> Hello Thread 1** and **Thread 2 Output --> Hello Thread 2**, but for the most part, the results are completely unpredictable.

Sometimes you see **Thread 2 Output --> Hello Thread 1** and **Thread 1 Output --> Hello Thread 2**. The thread output does not match the code! Even though, you look at the code and follow it with your eyes, \_threadOutput = "Hello Thread 2", Sleep, Write "Thread 2 --> Hello Thread 2", this sequence you expect does not necessarily produce the final result.

The reason you see the results you do is because in a multithreaded program such as this one, the code theoretically is executing the two methods DisplayThread1 and DisplayThread2, simultaneously. Each method shares the variable, \_threadOutput. So it is possible that although \_threadOutput is assigned a value "Hello Thread1" in thread #1 and displays \_threadOutput two lines later to the console, that somewhere in between the time thread #1 assigns it and displays it, thread #2 assigns \_threadOutput the value "Hello Thread2". Not only are these strange results, possible, they are quite frequent as seen in the output. This painful threading problem is an all too common bug in thread programming known as a **race condition**. This example is a very simple example of the well-known threading problem. It is possible for this problem to be hidden from the programmer much more indirectly such as through referenced variables or collections pointing to thread-unsafe variables. Although in your code the symptoms are blatant, a race condition can appear much more rarely and be intermittent once a minute, once an hour, or appear three days later. The race is probably the programmer's worst nightmare because of its infrequency and because it can be very very hard to reproduce.

### Step Four: Winning the Race

The best way to avoid race conditions is to write thread-safe code.

If your code is thread-safe, you can prevent some nasty threading issues from cropping up. There are several defenses for writing thread-safe code. One is to share memory as little as possible. If you create an instance of a class, and it runs in one thread, and then you create another instance of the same class, and it runs in another thread, the classes are thread safe as long as they don't contain any static variables. The two classes each create their own memory for their own fields, hence no shared memory.

If you do have static variables in your class or the instance of your class is shared by several other threads, then you must find a way to make sure one thread cannot use the memory of that variable until the other class is done using it.

The way you prevent one thread from effecting the memory of the other class while one is occupied with that memory is called **locking**. C# allows you to lock your code with either a Monitor class or a lock { } construct. (The lock construct actually internally implements the Monitor class through a try-finally block, but it hides these details from the programmer).

In the example, you can lock the sections of code from the point in which you populate the shared \_threadOutput variable all the way to the actual output to the console. You lock your **critical section** of code in both threads so you don't have a race in one or the other.

The quickest and dirtiest way to lock inside a method is to lock on the “this” pointer. Locking on the “this” pointer will lock on the entire class instance, so any thread trying to modify a field of the class while inside the lock, will be **blocked**. Blocking means that the thread trying to change the variable will sit and wait until the lock is released on the locked thread. The thread is released from the lock upon reaching the last bracket in the lock { } construct.

Take a look at how this works:

|  |
| --- |
| /// <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  }  } |

The only thing added to this code was the locks. So when thread1 inters it’s function, it will check for the \_stopThreads. If it is false, it will then lock the instance of the class. This lock will stop ANY other threads from working on the class. Any other thread that wants to access this class must wait for thread1 to release the lock. This means if your second thread wishes to run its function (DisplayThread2) it must wait for the first thread (running DIsplayThread1) to release the lock.

Run unity and observe the difference in the output.

You should get Display Thread 1 and Thread 1 Output --> Hello Thread 1 ten times. Then you get one output of Display thread 2 and Thread 2 output --> Hello Thread 2.

The first thread dominates the debug, because there is not enough time after thread 1 releases its lock for thread2 to acquire a new lock before thread1 acquires its lock again.

However, notice that display thread 2 displays AFTER the invoke has set \_stopThreads to true (This happens 10 seconds into the program, each thread displays its output every 1 second, if display thread 1 is printed 10 times then the invoke will happen before the 11th print…right?)

### Wrap Up

You should now have a very fundamental view on how to start a thread, and what the race is when examining threads, as well as one of the many solutions on how to avoid the race.

### Post-Lab

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?

### On Your Own!

Create a new script with four new threads and three new functions.

Function 1: This should print the value “x” 10 times.

Function 2: This should print the value “y” 10 times.

Function 3: This should check to see if a flag variable (done) is false. If it is not true, it should debug “Done” and then set the variable done to true.

Thread 1 : Calls function 1

Thread 2 : Calls function 2

Thread 3: Calls function 3

Thread 4: Calls function 4

Include your output after running this program once in your lab report.