**Lab Report**

**Lab 12 - Multithreading**

**Craig Broskow**

**November 23, 2015**

Introduction:

Although this lab is titled "Multithreading", we started off by exploring Lambda expressions, which are used in multithreading to call a threaded method and pass in one or more arguments if desired. Lambda expressions are generally used to create delegates, which are basically pointers to methods. Because delegates simply point to a method, they are useful in multithreading to have lambda expressions create function calls on the fly and then pass them to new threads for separate execution from the main thread.

We also looked at some features of multithreading that are either useful or require attention because of potential pitfalls. Threads acessing the same memory location (such as a loop counter variable) may incorrectly access the data before it's had a chance to change ("captured variables"). We also looked at thread names and foreground vs. background threads.

Methods:

In the first part of the lab, we compared results between using a standard C# function to determine whether an age corresponds to a teenager versus a lambda expression that created a delegate to a one-line C# statement that returned a boolean expression. The results were, of course, identical since the logic in both instances was identical. The lambda expression simply allowed us to use a short-hand approach to create a delegate method.

We expanded on this simple introduction by using lambda expressions that contain: multiple parameters, typed parameters, multiple statements, and local variables within the lambda definition.

The real meat of this lab was to use lambda expressions to easily create threads that call methods within the containing class, and also pass parameters into those methods. Since threads generally just use delegates that contain method names as their parameters, this technique increases the power and ease of passing parameters into threads. (However, it should be noted that threads have access to variables within the calling Class, so the passing of parameters is useful but not absolutely necessary for thread versatility.)

As mentioned in the Introduction to this lab report, shared memory locations (like loop counter variables) can present problems for thread execution accuracy. Since thread execution is somewhat unpredictable, threads may actually access the shared memory location before its content has been updated and synchronized. This gives incorrect results, and is due to the nature in which C# handles the updating of variables (called "captured variables"). A fairly simple solution to this problem is to copy the shared memory location into a local, temporary variable for the thread. Since each iteration of the thread will have its own version of the local variable, there should be no problem with the thread incorrectly accessing out-of-date data.

Although there are System methods to differentiate threads, it is sometimes useful to assign names to threads. This makes it easier to identify which thread is making what changes when debugging a multi-threaded program.

Finally, we looked at foreground versus background threads. According to the lab instructions, when all foreground threads end, the application terminates. This aborts any running background threads. Although this is a significant point to keep in mind, there is no difference in priority or allocation of execution time between foreground and background threads. It seems mainly important to note that all running background threads abort when the last foreground thread finishes execution.

Conclusions:

This final lab explored some of the nuances of multi-threaded programming. It examined some short-hand techniques to make calling threads more convenient, and also looked at some of the subtle problems that can be encountered when running multiple threads. There is no question that multi-threaded programming is a very complicated subject. Using it correctly requires careful planning and understanding of the subject. Furthermore, some precautions (such as locking data objects) take away some of the advantages of multiple threads, and testing of multi-threaded applications must generally be more comprehensive than with single-threaded programs.

The fact that the Unity game engine is single threaded means that spawned threads cannot easily manipulate some of the data under Unity's control (such as game objects), and they cannot call the Unity API (which is not thread-safe). My conclusion is that multi-threading in a Unity environment is best used for very specific purposes, such as increasing the responsiveness of the user-interface, and possibly in loading some assets using background threads.

Postlab Question:

1. Explain what is displayed in the console and why it is displayed as it is.

The console displays the text "t2" on two separate lines.

Although the two threads are created after the string variable "text" is set to two different values ("t1" and "t2"), they are both *started after* variable "text" has been set to "t2". Therefore, both threads access the text variable when its value is "t2", and display the same thing.

On Your Own Problem:

My "Lab 12" Unity project contains two scenes: "TestScene" and "MainScene". Scene "TestScene" contains six game objects (capsules) that have attached scripts which run the instructional code in this lab, and also the postlab question answer.

Scene "MainScene" contains my solution to the "On Your Own" problem. It contains four game objects: three cubes and one sphere. These game objects use two scripts ("MovementScript" and "HealthScript") to perform the instructions contained within the problem.

Code:

// LambdaTest.cs

using UnityEngine;

using System.Collections;

public class LambdaTest : MonoBehaviour {

delegate bool isTeenager(Student student);

// Use this for initialization

void Start () {

Student student1 = new Student (5);

Student student2 = new Student (10);

Student student3 = new Student (15);

Student student4 = new Student (20);

Debug.Log ("Function student1 teenager? : " + funcIsATeen (student1));

Debug.Log ("Function student2 teenager? : " + funcIsATeen (student2));

Debug.Log ("Function student3 teenager? : " + funcIsATeen (student3));

Debug.Log ("Function student4 teenager? : " + funcIsATeen (student4));

isTeenager isATeen = s => s.age > 12 && s.age < 20;

Debug.Log ("Lambda student1 teenager? : " + isATeen (student1));

Debug.Log ("Lambda student2 teenager? : " + isATeen (student2));

Debug.Log ("Lambda student3 teenager? : " + isATeen (student3));

Debug.Log ("Lambda student4 teenager? : " + isATeen (student4));

}

bool funcIsATeen(Student currStudent)

{

return (currStudent.age > 12 && currStudent.age < 20);

}

// Update is called once per frame

void Update () {

}

class Student

{

public int age;

public Student()

{

age = 10;

}

public Student(int age)

{

this.age =age;

}

}

}

// Arguments.cs

using UnityEngine;

using System.Collections;

using System.Threading;

public class Arguments : MonoBehaviour {

void Start()

{

Thread thread = new Thread (() => DisplayMessage("Hello from the thread!"));

thread.Start ();

new Thread ( () =>

{

Debug.Log ("I'm running on another thread!");

Debug.Log ("This is so cool!");

}

).Start();

}

void DisplayMessage(string message)

{

Debug.Log (message);

}

}

// Captured.cs

using UnityEngine;

using System.Collections;

using System.Threading;

public class Captured : MonoBehaviour {

// Use this for initialization

void Start ()

{

Debug.Log("First for loop:");

for(int i = 1; i <= 10; i++)

{

new Thread( ()=> Debug.Log (i)).Start();

}

Thread.Sleep(1000);

Debug.Log("Second for loop:");

for(int i = 11; i <= 20; i++)

{

int temp = i;

new Thread( ()=> Debug.Log (temp)).Start();

}

}

// Update is called once per frame

void Update () {

}

}

// Names.cs

using UnityEngine;

using System.Collections;

using System.Threading;

public class Names : MonoBehaviour {

// Use this for initialization

void Start () {

Thread.CurrentThread.Name = "main";

Thread worker = new Thread (Go);

worker.Name = "worker";

worker.Start();

Go();

}

void Go()

{

Debug.Log ("Hello from " + Thread.CurrentThread.Name);

}

// Update is called once per frame

void Update () {

}

}

// ForegroundBackground.cs

using UnityEngine;

using System.Collections;

using System.Threading;

public class ForegroundBackground : MonoBehaviour {

// Use this for initialization

void Start ()

{

Thread worker = new Thread (Go);

worker.Name = "worker";

Debug.Log ("is worker background? " + worker.IsBackground);

worker.IsBackground = true;

Debug.Log ("is worker background? " + worker.IsBackground);

worker.Start ();

}

void Go()

{

Debug.Log ("Hello from " + Thread.CurrentThread.Name);

}

// Update is called once per frame

void Update () {

}

}

// Postlab.cs

using UnityEngine;

using System.Collections;

using System.Threading;

public class Postlab : MonoBehaviour {

// Use this for initialization

void Start ()

{

string text = "t1";

Thread t1 = new Thread( () => Debug.Log (text));

text = "t2";

Thread t2 = new Thread (() => Debug.Log (text));

t1.Start ();

t2.Start ();

}

// Update is called once per frame

void Update () {

}

}

// MovementScript.cs

using UnityEngine;

using System.Collections;

using System.Threading;

public class MovementScript : MonoBehaviour {

private const float DELTA\_TIME = 5.0f;

public float xSpeed = 0.0f;

public float ySpeed = 0.0f;

public float zSpeed = 0.0f;

private float xPosition = 0.0f;

private float yPosition = 0.0f;

private float zPosition = 0.0f;

private float changeDirectionTime = 0.0f;

private bool threadIsWorking = false;

private float dTime = 0.0f;

// Use this for initialization

void Start ()

{

xPosition = transform.position.x;

yPosition = transform.position.y;

zPosition = transform.position.z;

changeDirectionTime = Time.time + DELTA\_TIME;

}

// Update is called once per frame

void Update ()

{

if ((changeDirectionTime > 0.0f) && (Time.time > changeDirectionTime))

{

changeDirectionTime = Time.time + DELTA\_TIME;

xSpeed = Random.Range(-0.5f, 0.5f);

ySpeed = Random.Range(-0.5f, 0.5f);

zSpeed = Random.Range(-0.5f, 0.5f);

}

xPosition = transform.position.x;

yPosition = transform.position.y;

zPosition = transform.position.z;

dTime = Time.deltaTime;

threadIsWorking = true;

new Thread( ()=> CalcNewPosition(dTime)).Start();

while (threadIsWorking)

{

// wait, or do something else

}

transform.position = new Vector3(xPosition, yPosition, zPosition);

} // end method Update

void CalcNewPosition(float dTime)

{

xPosition += xSpeed \* dTime;

yPosition += ySpeed \* dTime;

zPosition += zSpeed \* dTime;

threadIsWorking = false;

}

} // end class MovementScript

// HealthScript.cs

using UnityEngine;

using UnityEngine.UI;

using System.Collections;

public class HealthScript : MonoBehaviour {

private const int STARTING\_HEALTH = 5;

public int health;

public Text healthDisplay;

// Use this for initialization

void Start ()

{

health = STARTING\_HEALTH;

healthDisplay.text = "Health: " + health.ToString();

}

void OnTriggerEnter(Collider hitter)

{

health--;

healthDisplay.text = "Health: " + health.ToString();

}

} // end class HealthScript