Multi-threading in Unity

Advanced C# Software Construction Elective Course Synopsis

Author: William Blackney
Supervisor: Peter Levinsky
Date: 18.05.2020

Table of Contents

Table of Contents

Introduction

This project intends to answer the question *"How can multithreaded applications be safely implemented within the Unity Engine?"*. I have chosen the topic of threading within Unity mainly because it is of great personal interest to me. As a game developer, i'm always looking to expand my knowledge of game development, and much of the knowledge learnt within this project could also be applied to creating multithreaded applications outside of Unity.

I believe this project has greater consequences outside of my self and to other developers, as the subject of multithreading in Unity has seldom been explored until recently, yet it is an important part of any working game application. My hope is that this project will illuminate the path of learning to create multithreaded applications within Unity for myself, and for other aspiring developers.

To that end, this project will explore and create multithreaded solutions using Unity's newly developed 'C# Job System Package': a suite of libraries and packages used to help developers manage building multithreaded applications. Since the 'C# Job System Package' is not designed to be a standalone package and operates as a core piece of Unity's new 'Data Oriented Technology Stack' (DOTS), the solutions in the project will also implement some of these packages, libraries and tools from DOTS, particularly the 'Entity Component System' (ECS), the 'Native Libraries' package, and the 'Burst Compiler' package.

For the purposes of being thorough in my goal to understand multi-threading in Unity, and also to highlight the benefits of DOTS, this project will also explore traditional approaches to multithreading within Unity that were used before the development of DOTS.

Problem Definition

In order to help narrow the scope and focus of this project, i condensed the problem defintion down into 3 questions:

1. When and why should developers use multithreading instead of coroutines?
2. How can developers use threading to improve performance and create asynchronous functionality of operations that are restricted to the main thread?
3. How does the new 'Data Oriented Technology Stack' (DOTS) and the Unity 'Job System' package change multithreading implementation?

Method and Project Planning

When i started planning this project on the 4th of May 2020, i knew that i would have exactly 25 days to complete it. With this in mind, i scheduled and planned my activities to last 21 days, allowing myself 4 days extra in case of illness or underestimation of task duration.

Phase 1

- **General Information Gathering Phase + The Basics (3 DAYS)**
  - Watch youtube videos and read Microsoft documentation on threading (unrelated to unity)
  - Recomplete Peter's Levinsky's lessons/exercises on threading/parrallelism on moodle ('Brewery' exercise)
  - Create my own simple threading solution within VS19

Phase 2

- **'Old School' Threading In Unity (Pre Jobs System) (5 DAYS)**
  - Read Unity oficial API documentation on threading (deprecated v2018)
  - Watch youtube instuctional videos to get a general overview of the topic
  - Read about successful case studies on threading done by established developers (e.g. Richard Meredith from Bad North)
    - Replicate/Implement the threading code used by a case study.
  - Follow along / Complete youtube courses on threading (from 2017-2019)
  - Complete youtube courses on old school threading by 'Quill18'
  - Cement new knowledge with the 'Old School' threading methods by creating new Unity projects specifically for learning.
    - 'Level Loader' Project
    - '1000's of Zombies' Project

Phase 3

- **Threading with the Unity 'Job System' and DOTS (5 DAYS)**
  - Read Unity oficial API documentation on the 'Job System' library
  - Read Unity oficial API documentation on ECS (Entity Components System)
  - Watch youtube instuctional videos to get a general overview of the topic
    - Official unity courses
    - Official unity demonstrations/product reveals from 2020 at 'Unite Berlin' event.
    - Unnofficial courses and reviews of the new system (e.g. The course by 'Code Monkey')
  - Cement new knowledge by creating new Unity projects specifically for practicing 'Job System' implementation.
      - 'Level Loader' Project
      - '1000's of Zombies' Project

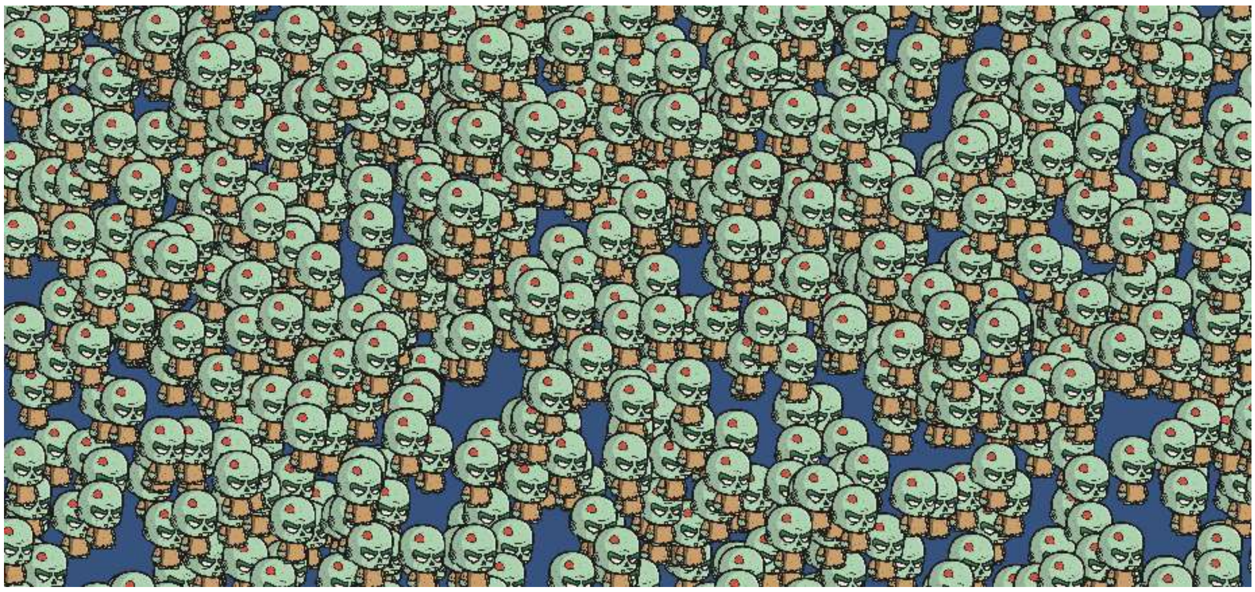
Phase 4

- **Create practical threading solutions within my current project: 'Hereos Of Herp Derp' (5 DAYS)**
  - Astar Pathfinding
  - Level/Scene Loading (Async)
  - Canvas Rendering

Phase 5

- **Complete Synopsis Documentation (3 DAYS)**
  - Prepare the 10 page document for submission
  - Prepare 10 minute exam preperation
  - Prepare for 20 minute interview

One Thousand Zombies!



Overview

In order to adequately test, design, implement and explore the true power of all the different approaches to multi-threading in Unity, i created a simple project called 'One Thousand Zombies!'. The functionality of the application is simple: instantiate 1000 'Zombie' game objects, then move them up and down the screen. In order to fairly compare the performance of different multithreading approaches ('Old School' threading, Jobs System Threading, etc), each approach was implemented within in this project, and performed the exact same task of moving zombies up and down the screen.

'Old School' Threading Implentation

Initial Problem

Before i can explain the implementation of my pre-job system threading solution, it is important to understand an underlying restriction within Unity: **the Unity API cannot be called on a thread that is NOT the main thread.** This is a common feature of many API's, and for good reason. Multiple threads accessing/modifying the same object simultaneously can have horrible, unpredictable consequences. Any attempt to access the Unity API from a child thread will cause an error. How can this restriction be circumnavigated?

Solution Explanation

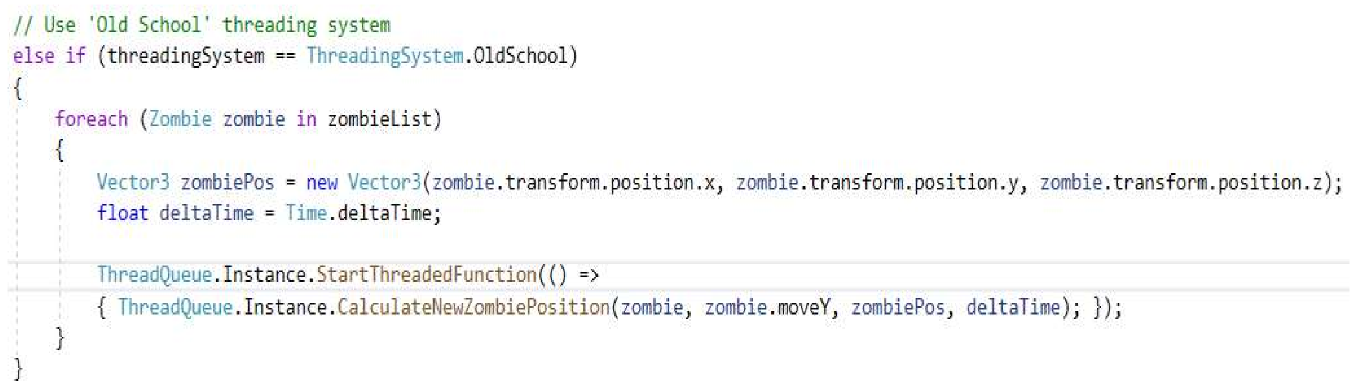
The step by step proccess to moving the zombies off the main thread is as follows:

1. Use a function to calculate (but not move) the zombies new position on a child thread.
2. Use that same function on the child thread to create an anonymous function that

moves the zombie (this allows us to safely call the Unity API from a child thread)
3. Add the anonymous function to a queue of 'Actions' awaiting execution on the main thread.
4. During the next Update function call (on the main thread), call all queued functions in the function queue, thus moving the zombie.

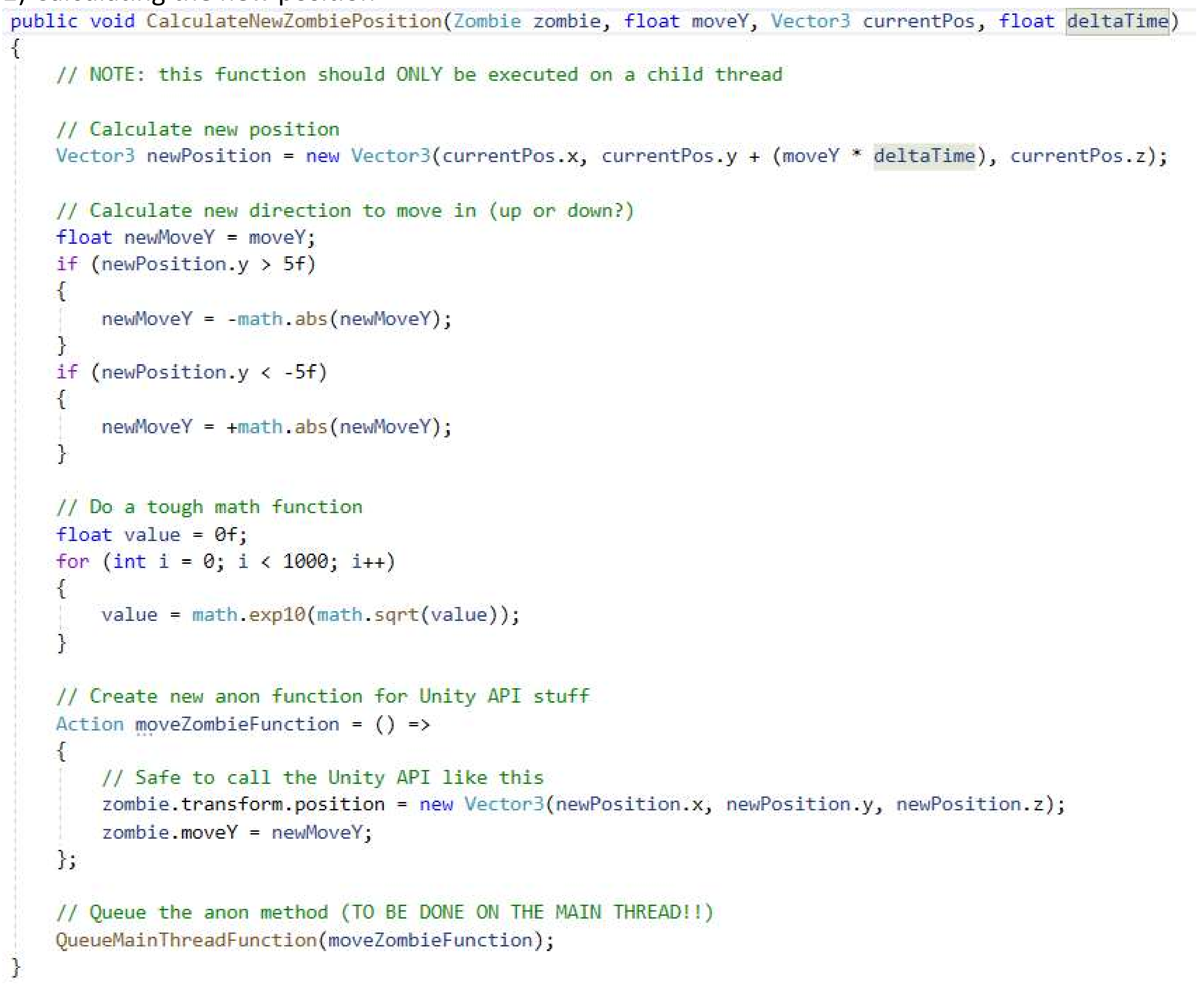
Code Examples

1) Starting the process of multithreaded zombie movement



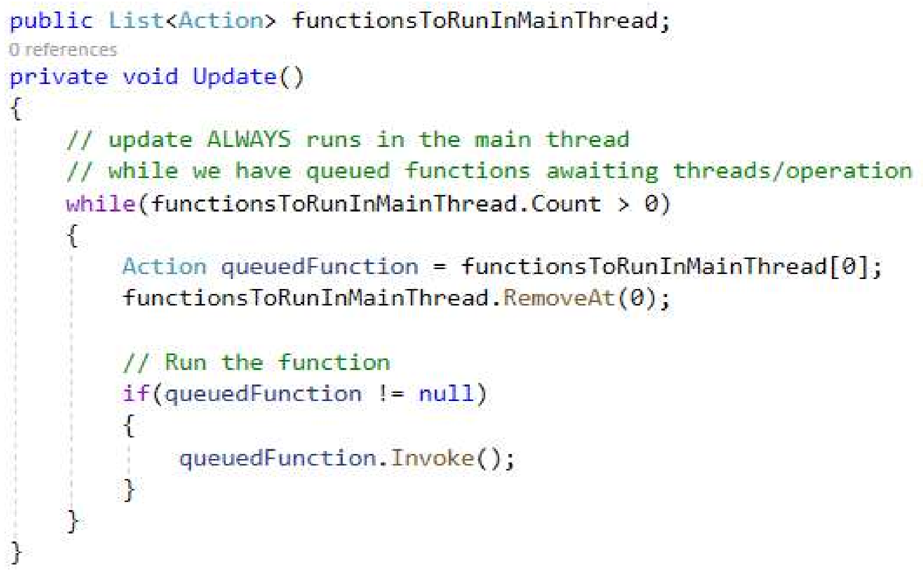
```
// Use 'Old School' threading system
else if (threadingSystem == ThreadingSystem.OldSchool)
{
    foreach (Zombie zombie in zombieList)
    {
        Vector3 zombiePos = new Vector3(zombie.transform.position.x, zombie.transform.position.y, zombie.transform.position.z);
        float deltaTime = Time.deltaTime;

        ThreadQueue.Instance.StartThreadedFunction(() =>
        { ThreadQueue.Instance.CalculateNewZombiePosition(zombie, zombie.moveY, zombiePos, deltaTime); });
    }
}
```

2) Calculating the new position



```
public void CalculateNewZombiePosition(Zombie zombie, float moveY, Vector3 currentPos, float deltaTime)
{
    // NOTE: this function should ONLY be executed on a child thread

    // Calculate new position
    Vector3 newPosition = new Vector3(currentPos.x, currentPos.y + (moveY * deltaTime), currentPos.z);

    // Calculate new direction to move in (up or down?)
    float newMoveY = moveY;
    if (newPosition.y > 5f)
    {
        newMoveY = -math.abs(newMoveY);
    }
    if (newPosition.y < -5f)
    {
        newMoveY = +math.abs(newMoveY);
    }

    // Do a tough math function
    float value = 0f;
    for (int i = 0; i < 1000; i++)
    {
        value = math.exp10(math.sqrt(value));
    }

    // Create new anon function for Unity API stuff
    Action moveZombieFunction = () =>
    {
        // Safe to call the Unity API like this
        zombie.transform.position = new Vector3(newPosition.x, newPosition.y, newPosition.z);
        zombie.moveY = newMoveY;
    };

    // Queue the anon method (TO BE DONE ON THE MAIN THREAD!!)
    QueueMainThreadFunction(moveZombieFunction);
}
```

3) Execution from the main thread (post calculation)



```
public List<Action> functionsToRunInMainThread;
0 references
private void Update()
{
    // update ALWAYS runs in the main thread
    // while we have queued functions awaiting threads/operation
    while(functionsToRunInMainThread.Count > 0)
    {
        Action queuedFunction = functionsToRunInMainThread[0];
        functionsToRunInMainThread.RemoveAt(0);

        // Run the function
        if(queuedFunction != null)
        {
            queuedFunction.Invoke();
        }
    }
}
```

Jobs System Threading Implentation

Overview and Initial Problem

Unity's *C# Job System* is not designed to operate on its own. Instead, it operates as one part of the DOTS System. Because of this, creating a working multi-threaded solution using the C# Job System required the installation/importing of all the packages and libraries associated with DOTS, the main ones being

1. Entity Component System Package
2. Jobs Package
3. Native Collections Package
4. Burst Compiler Package
5. Mathematics Library

Solution Explanation

The first step in creating a multithreaded operation with the job system is to create a new struct object that inherits from one of the 'Job' interfaces from the Jobs Package namespace (in our example, the interface used is 'IjobParallelFor'). The method 'Execute' is inherited from the job interface, and it is within this method that we write the code that calculates the new position of a zombie.

The benefits of implementing an 'Ijob' interface are:

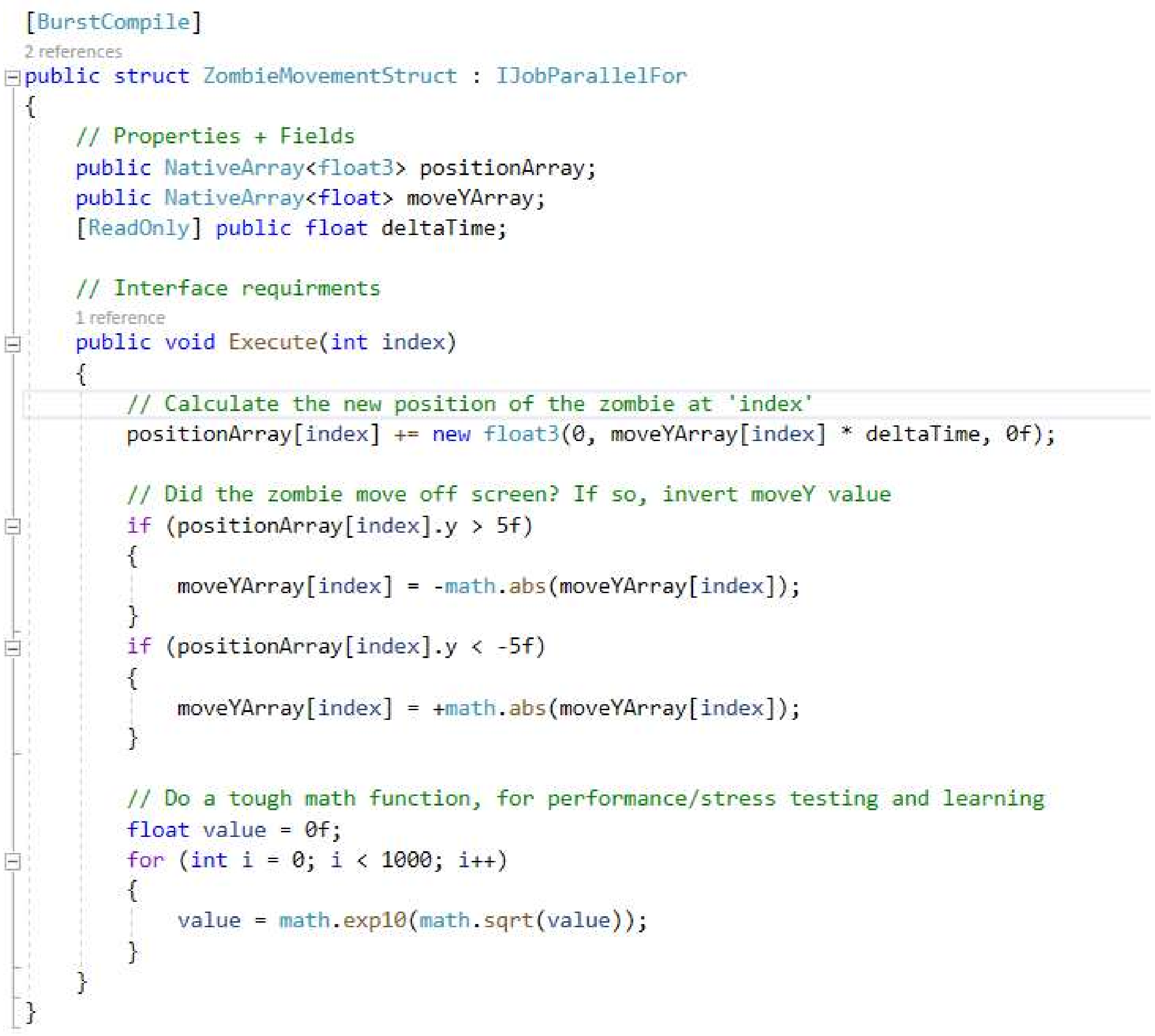
1. We do not need to manually create/terminate child threads
2. We do not need to implement and manage our own thread pool system
3. We do not need to manually schedule and track the child threaded function
4. We do not need to create a system that creates and queues anonymous functions to be executed in the main thread post child thread calculation

However, the consequences of this system are:

1. For every function we want to be performed on a child thread, we need to create a whole new struct that implements an Ijob interface. If there are 700 different functions in our application we want to be executed from a child thread, we would need to create 700 different structs. **This is not scaleable**.
2. Any arrays/lists/etc from the 'Native Collections' library need to be disposed of manually when they move out of scope. Failure to do this will leak memory without throwing an error, which could make finding and debugging a memory leak problem difficult.

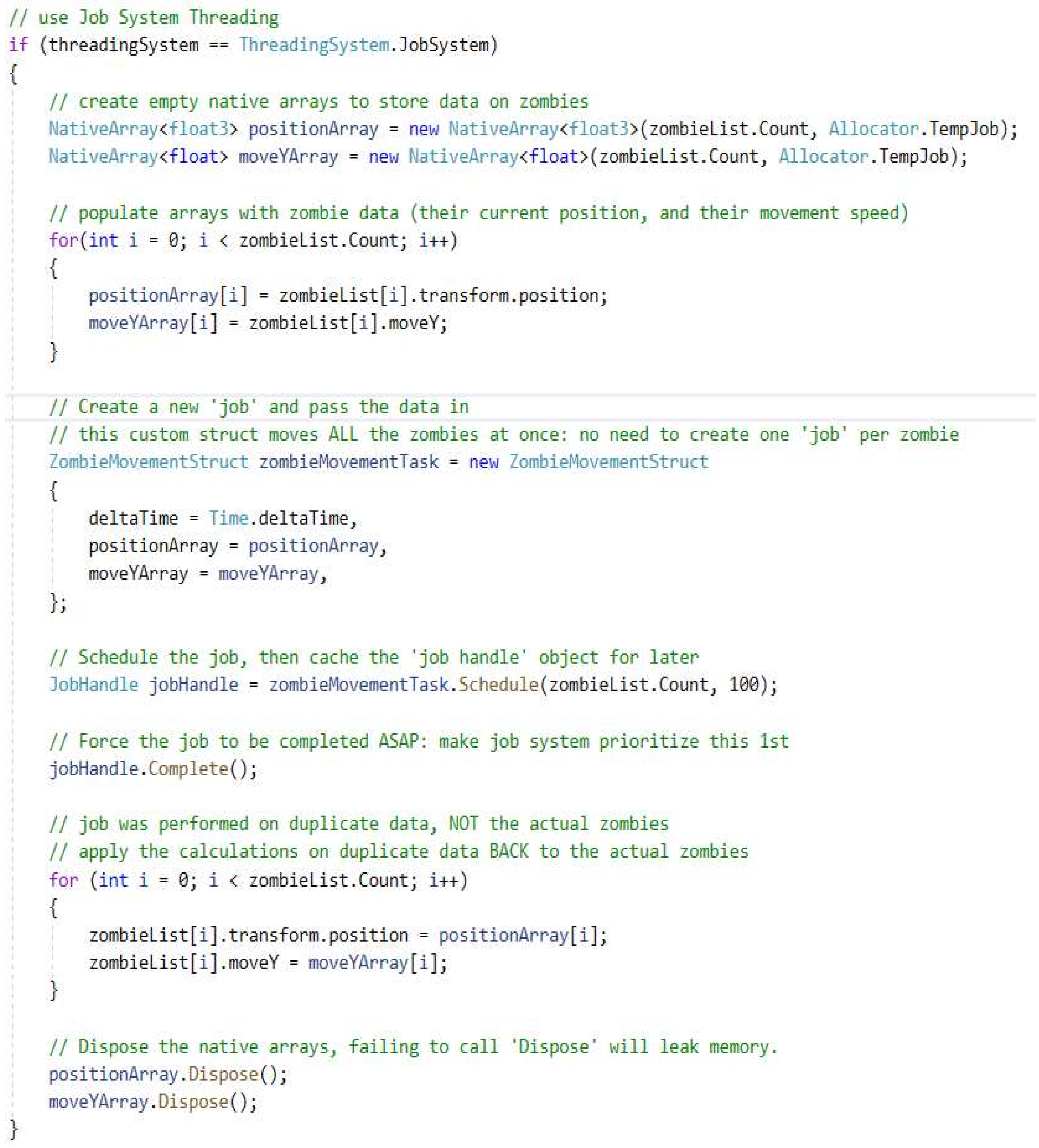
Code Examples

1) Creating a struct to calculate the new position of a zombie



```
[BurstCompile]
2 references
public struct ZombieMovementStruct : IJobParallelFor
{
    // Properties + Fields
    public NativeArray<float3> positionArray;
    public NativeArray<float> moveYArray;
    [ReadOnly] public float deltaTime;

    // Interface requirments
    1 reference
    public void Execute(int index)
    {
        // Calculate the new position of the zombie at 'index'
        positionArray[index] += new float3(0, moveYArray[index] * deltaTime, 0f);

        // Did the zombie move off screen? If so, invert moveY value
        if (positionArray[index].y > 5f)
        {
            moveYArray[index] = -math.abs(moveYArray[index]);
        }
        if (positionArray[index].y < -5f)
        {
            moveYArray[index] = +math.abs(moveYArray[index]);
        }

        // Do a tough math function, for performance/stress testing and learning
        float value = 0f;
        for (int i = 0; i < 1000; i++)
        {
            value = math.exp10(math.sqrt(value));
        }
    }
}
```

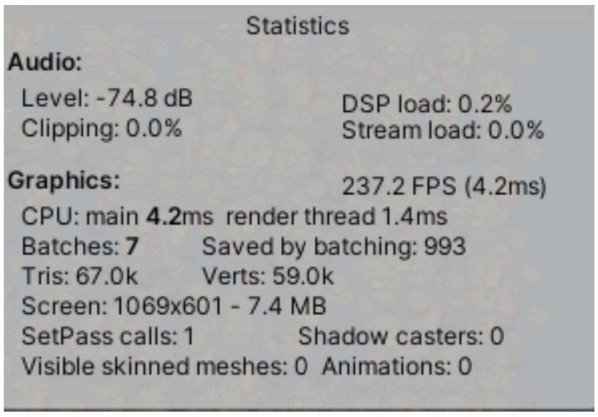
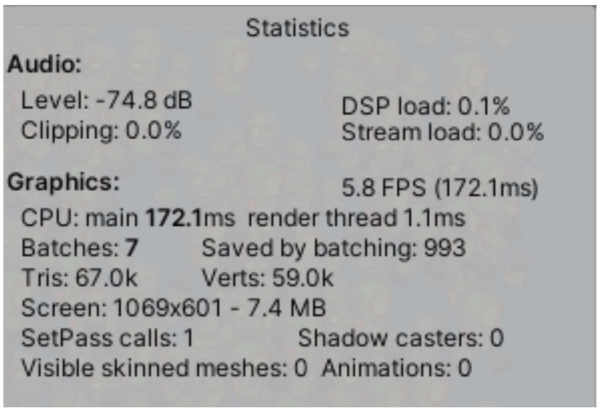
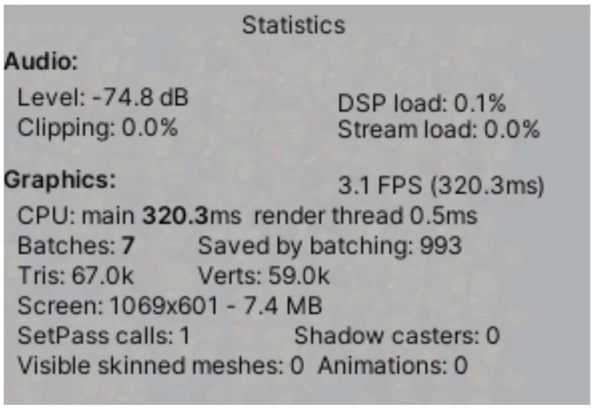
2) Starting the process of calculating the new zombie postion (on a child thread), then moving it (on the main thread)



```
// use Job System Threading
if (threadingSystem == ThreadingSystem.JobSystem)
{
    // create empty native arrays to store data on zombies
    NativeArray<float3> positionArray = new NativeArray<float3>(zombieList.Count, Allocator.TempJob);
    NativeArray<float> moveYArray = new NativeArray<float>(zombieList.Count, Allocator.TempJob);

    // populate arrays with zombie data (their current position, and their movement speed)
    for(int i = 0; i < zombieList.Count; i++)
    {
        positionArray[i] = zombieList[i].transform.position;
        moveYArray[i] = zombieList[i].moveY;
    }

    // Create a new 'job' and pass the data in
    // this custom struct moves ALL the zombies at once: no need to create one 'job' per zombie
    ZombieMovementStruct zombieMovementTask = new ZombieMovementStruct
    {
        deltaTime = Time.deltaTime,
        positionArray = positionArray,
        moveYArray = moveYArray,
    };

    // Schedule the job, then cache the 'job handle' object for later
    JobHandle jobHandle = zombieMovementTask.Schedule(zombieList.Count, 100);

    // Force the job to be completed ASAP: make job system prioritize this 1st
    jobHandle.Complete();

    // job was performed on duplicate data, NOT the actual zombies
    // apply the calculations on duplicate data BACK to the actual zombies
    for (int i = 0; i < zombieList.Count; i++)
    {
        zombieList[i].transform.position = positionArray[i];
        zombieList[i].moveY = moveYArray[i];
    }

    // Dispose the native arrays, failing to call 'Dispose' will leak memory.
    positionArray.Dispose();
    moveYArray.Dispose();
}
```

Comparing The Solutions

The functionality of calculating and moving 1000 zombies up and down the screen was performed within the context of each threading implementation, and the results were recorded...

Measurements	System Used
 Statistics Audio: Level: -74.8 dB DSP load: 0.2% Clipping: 0.0% Stream load: 0.0% Graphics: 237.2 FPS (4.2ms) CPU: main **4.2**ms render thread 1.4ms Batches: **7** Saved by batching: 993 Tris: 67.0k Verts: 59.0k Screen: 1069x601 - 7.4 MB SetPass calls: 1 Shadow casters: 0 Visible skinned meshes: 0 Animations: 0	**C# Jobs System Solution** - 237 Frames per Second - 4ms Frame duration
 Statistics Audio: Level: -74.8 dB DSP load: 0.1% Clipping: 0.0% Stream load: 0.0% Graphics: 5.8 FPS (172.1ms) CPU: main **172.1**ms render thread 1.1ms Batches: **7** Saved by batching: 993 Tris: 67.0k Verts: 59.0k Screen: 1069x601 - 7.4 MB SetPass calls: 1 Shadow casters: 0 Visible skinned meshes: 0 Animations: 0	**Non multi-threaded solution** - 5.8 Frames per Second - 172.1ms Frame duration
 Statistics Audio: Level: -74.8 dB DSP load: 0.1% Clipping: 0.0% Stream load: 0.0% Graphics: 3.1 FPS (320.3ms) CPU: main **320.3**ms render thread 0.5ms Batches: **7** Saved by batching: 993 Tris: 67.0k Verts: 59.0k Screen: 1069x601 - 7.4 MB SetPass calls: 1 Shadow casters: 0 Visible skinned meshes: 0 Animations: 0	**'Old School' Threading Solution** - 3.1 Frames per Second - 320.3ms Frame duration

Performance Comparisons

When it comes to moving a thousand zombies up and down the screen, the 'C# Sharp Job System' implementation was the fastest and most efficient, and by quite a significant margin (40x faster than the single threaded solution!). Interestingly, the 'Old School' implementation was actually slower then the single threaded solution, and although the average frame rate was 3.1 FPS, it occasionaly dipped as slow as 1.2 FPS.

Reviewing The Results Of The 'Old School' Method

As evidenced by the testing results, implementing an 'Old School' approach to threading in Unity not only fails to improve performance, it actually degrades it. This is due to a number of underlying issues that do not arise when using the 'C# Job System' approach.

Firstly, there is a small amount of overhead that occurs when creating and terminating a thread. When creating/terminating 1 or 2 threads at a time the overhead is negligible. However, the solution creates a new thread 1000 times a frame, creating significant overhead. Compounding this problem is the fact that there simply isn't enough cores in my computer (or probably anybody elses) to schedule, run a function on, and terminate 1000 threads within a single frame efficiently, which creates a 3rd problem: context switching. This occurs because there simply isnt enough processing time to be fairly divided between all the threads, and switching between the continuation of one thread to another adds more time and overhead to the execution.

Generally speaking, the desire to implement multithreaded applications comes from the need to prevent 'hanging' of the main thread, or to improve performance. My conclusion from this is that implementing an 'Old School' system of multithreading in Unity is only a viable and pratical option if your only goal is to prevent 'hanging' the main thread, and performance is not an issue (e.g generating a large part of the game world that is off in the distance). However, an 'Old School' approach to multithreading simply cannot be used to improve performance and increase the speed of operations within Unity.

Conclusions

1) When and why should developers use multithreading instead of coroutines?

Both coroutines and child threaded functions can prevent hanging of the main thread, but do so by different means. A coroutine divides the execution of a function with a 'yield' statement. Once a 'yield' statement is encountered, the function effectively pauses its execution, and the application continues with the remainding tasks placed on the stack. At the commencment of the next frame, the coroutine begins where it left off in the previous frame (from the 'yield' statement). While this is great for preventing hanging on the main thread by slicing a tough function into smaller pieces, it is not a very efficient use of computer performance, because the coroutine will yield when the programmer tells it to, rather than yielding when the coroutine's execution begins to degrade frame rate. How can a developer know exactly how many times a function should be sliced up? How big should each slice of the execution be in order to make the most efficient use of processing power?

For this reason, developers should use multithreading over coroutines when

- a function needs to be executed as quickly as possible
- a function does not need to reference the Unity API.
- The result of a function does not need to be synchronized with the main thread.

2) How can developers use threading to improve performance and create asynchronous functionality of operations that are restricted to the main thread?

A developer can certainly use multithreading to create asynchronous functionality within Unity, but cannot increase performance via multi threading with an 'Old School' approach: it's main purpose is to prevent hanging of the main thread. This is due to the fact that many of the most performance intensive operations that a developer would like to implement as multithreaded functions either cannot be done on a child thread (atomic operations, operations to access the Unity API, etc), or can be done, but create too much overhead from creating/terminating threads, context switching, tracking the state of threads in a pool, and queuing Unity API safe functions.

Fortunately, a developer can drastically improve the performance of their application by using a 'C# Job System' approach to multithreading. However, the boost in performance has very little to do with multithreaded functionality, and everything to do with:

- differences in memory access, usage and GC allocation between classes and structs (or rather, between reference types and data types)
- the unity ' Burst Compiler'
- the data oriented nature of the 'Entity Component System', opposed to object oriented design
- the unity 'Native Collections' package

*3) H*ow does the new 'Data Oriented Technology Stack' (DOTS) and the Unity 'Job System' package change multithreading implementation?

DOTS profoundly changes not only the way that multithreaded functionality is implemented within unity, but also the entire design and structure of the application by moving away from the object orientated paradigm and into the data oriented paradigm. Although using DOTS for multithreading comes with its own challenges and complications, it removes many of the tedious and challenging tasks associated with implementing multi threaded applications, such as task scheduling, creating/terminating threads, and circumnavigating the restriction of being unable to call the Unity API from a child thread. **Answering this question in more detail will be the main focus of my final presentation.**

Project Reflections

Overall, i believe my project was successful in answering the questions it had from the outset, and feel that i've learned a lot more about multi-threading then i thought i would. I did not anticipate that by exploring Unity's DOTS system, i would be forced to learn a huge amount about low level programming concepts (memory managment, GC allocation, pointers, etc), but am very happy that i did. In the future, i believe i should do more primary research before beginning a project such as this. While i originally set out to learn about multithreading in unity, i ended up spending a huge amount of time learning about DOTS instead. Better planning a more thorough pre-research would have informed me that the 'C# Job System' is only one cog in the DOTS machine, and that this project should have focused on DOTS from the beginning.

References

Unity DOTS General Overview - CodeMonkey
https://www.youtube.com/watch?v=Z9-WkwdDoNY

Implementing DOTS: Unity GDC 2019 Demonstration – Unity Technologies
https://www.youtube.com/watch?v=QbnVELXf5RQ

Pre DOTS/Jobs System Threading In Unity – Quill18
https://www.youtube.com/watch?v=ja63QO1Imck

C# Job System Documentation – Unity Technologies
https://docs.unity3d.com/Manual/JobSystem.html

Bad North: Multithreading Case Study – Richard Meredith
https://80.lv/articles/simple-multithreading-for-unity/