Multithreading & Games

Following American psychological association guidelines

Noaman Khalil

SAE Institute Dubai

Abstract

The goal of this paper is to demonstrate the understanding of Asynchronous processing otherwise known as multithreading in regards to their applications applied within video games. The paper also cover a method to do multithreading in Unity which will be covered using a real world example as it takes a long time to implement a working real world application worthy of demonstration .

Introduction

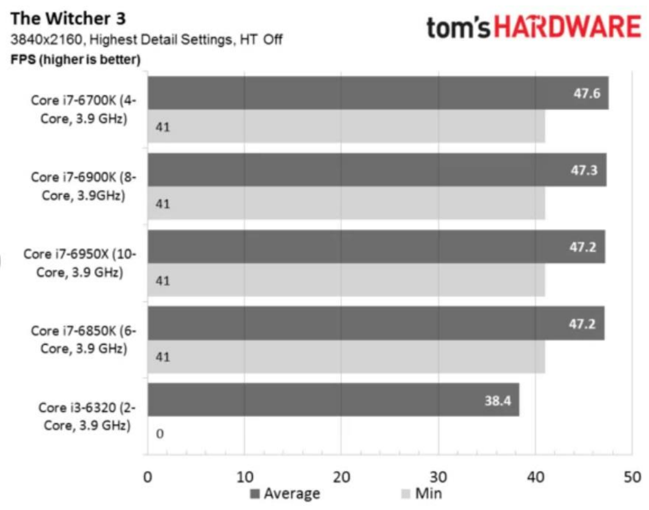
Firstly we must understand what Multi-threading is and how it works. Multithreading is a sort of execution show that enables different threads to exist inside the setting of a procedure to such an extent that they execute freely however share their procedures resources based on the needs of the operation. A thread keeps up a rundown of data pertinent to its execution including the need plan, special case handlers, an arrangement of CPU registers, and stack state in the address space of its facilitating procedure.

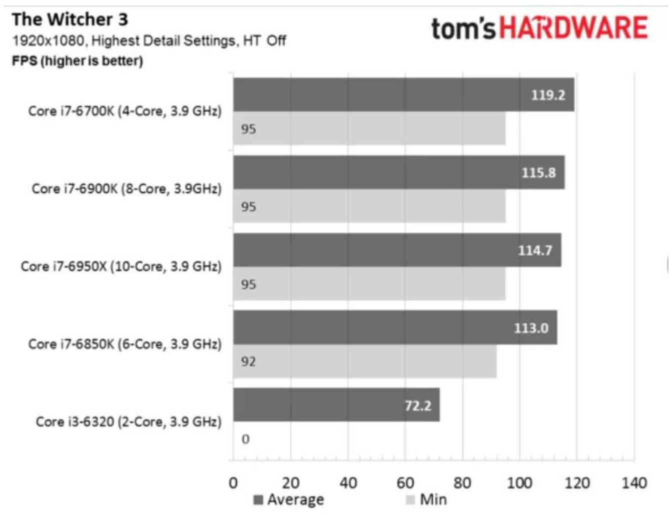
In modern computing each logical processor core has two threads which was not the case in early computing   wherein each processor had a single core. Interestingly enough the term multi-threading/ threading did not gain significant momentum until the 21st century when it became a popular term with the advent of Intel’s Hyper treading technology took off with the intel core I-x series of processors however its roots can be traced back to a corporation named “Digital Equipment corporation” or DEC for short, which was founded in 1957 according to Wikipedia. ("Digital Equipment Corporation," n.d.)

Next we must understanding the current tools used for the purpose of explanation for this paper, we will be mainly focusing around the popular game engine Unity3D. Unity3D was meant to do as its name implies , make games however the game it was made for “GooBall” did not do as well as its creators intended but , they did however realize the true potential of an engine and moved on to democratize game development. ("Unity (game engine)," 2018)

Common uses

Multithreading is a very commonly used hardware feature used to speed up processing for games, however during the 32bit era of games most games only had 2Cores/2threads which limited their use in games and many games at the time only utilized a single core for processing general processes while a large amount of graphics processing was also done on the processor as graphic cards from NVidia& the Radeon group were not as powerful however this change drastically with 64-bit processors and more powerful graphic accelerators along with the advent of CUDA & PhysX came into the market with games such as Witcher 3 taking advantage of the extra processing cores & multithreading to give better performance in games which is evident by the benchmarks below.





(Angelini, 2016)

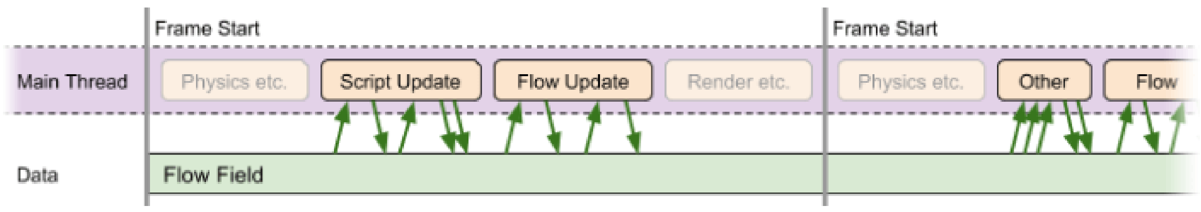
The problem

Unity does not support multithreading in its API thus when you inherit from "monobehaviour" and try to use threading via the system.Threading namespace unity (version 5.6) would not run the thread and not display the code in it. In previous versions of unity there would be large errors thrown in the console which would stop the project from playing however this would be if any tasks are being loaded onto the main thread and can be avoided if the tasks that are being threaded manually i.e.: outside unities job system without interrupting the main thread, a form of multithreading could be achieved as can be seen in the example section below.

The example

After some research online I found a small developer post outlining how to do multithreading for a small game called "Bad North" wherein the developer reduced the thread time on mobile from 2.75ms to 0.25ms , this reduces the time taken by the main thread to do the task and in turn speeds up the overall process .

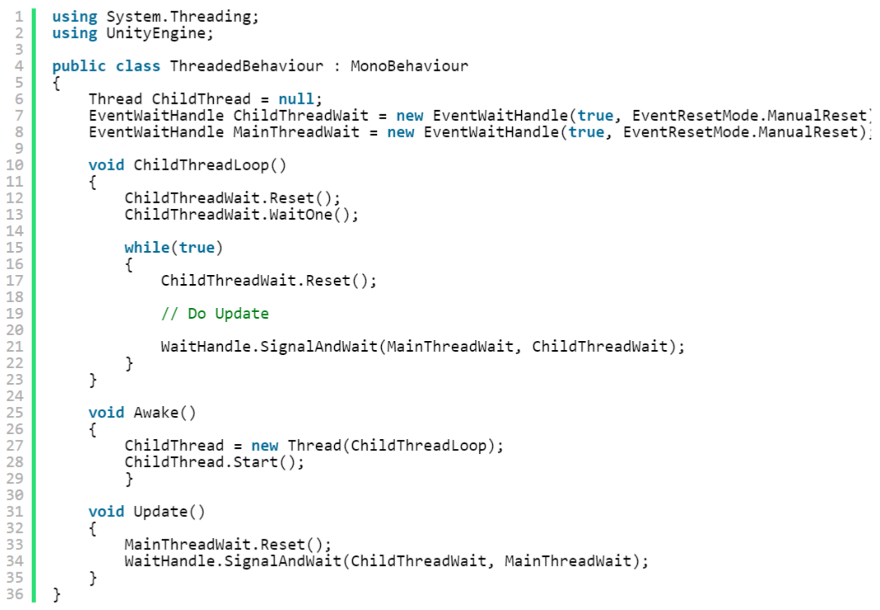
To start off the developer explains how everything is executed on the main thread which is illustrated in the image below.



(Meredith, 2017)

The illustration shows how the scripts read and write to the flow field over the period of a single frame which is shown by the green arrows. During the flow update, numerous reads and writes are done to the data, the flow update is what the developer aims to move to a separate thread which is the most costly part of the process.

The developer's solution was to create a thread which would run in a simple infinite loop which would later match that loop with the man Update loop in Unity. The code below is how it was done but it generates about 500B of garbage per frame.



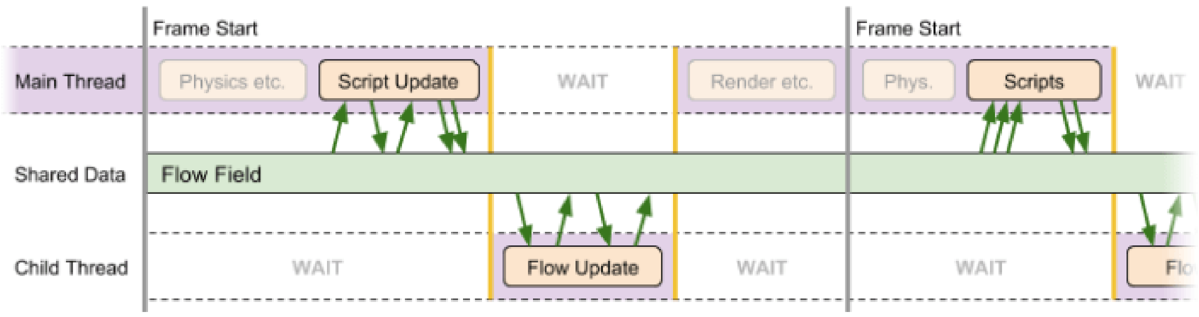
```
using System.Threading;
using UnityEngine;

public class ThreadedBehaviour : MonoBehaviour
{
    Thread ChildThread = null;
    EventWaitHandle ChildThreadWait = new EventWaitHandle(true, EventResetMode.ManualReset)
    EventWaitHandle MainThreadWait = new EventWaitHandle(true, EventResetMode.ManualReset);

    void ChildThreadLoop()
    {
        ChildThreadWait.Reset();
        ChildThreadWait.WaitOne();

        while(true)
        {
            ChildThreadWait.Reset();

            // Do Update

            WaitHandle.SignalAndWait(MainThreadWait, ChildThreadWait);
        }
    }

    void Awake()
    {
        ChildThread = new Thread(ChildThreadLoop);
        ChildThread.Start();
        }

    void Update()
    {
        MainThreadWait.Reset();
        WaitHandle.SignalAndWait(ChildThreadWait, MainThreadWait);
    }
}
```

(Meredith, 2017)

The essential thing here are the two EventWaitHandle variables , which are to quadrate the threads . While a thread invokes to wait on a wait handle which is reset (e.g line 13), it will square until set () is called on that EventWaitHandle by a different thread. The SignalAndWait() function is the same as calling Set() and WaitOne() on the two parameters () which would discharge one thread , and obstruct the present one.

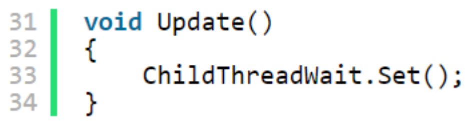
The awake() in the ThreadBehaviour class will actualize a child thread and start it , whch will begin running the code in the ChildThreadLoop and directly wait on the ChildThreadWait . It will remain in the block state until the update function is called.

Below is an image of how the process would look like which is not real asynchronous threading as yet because we have to wait for the script update to be finished before the flow update can begin however some of the work is pushed to another thread .



(Meredith, 2017)

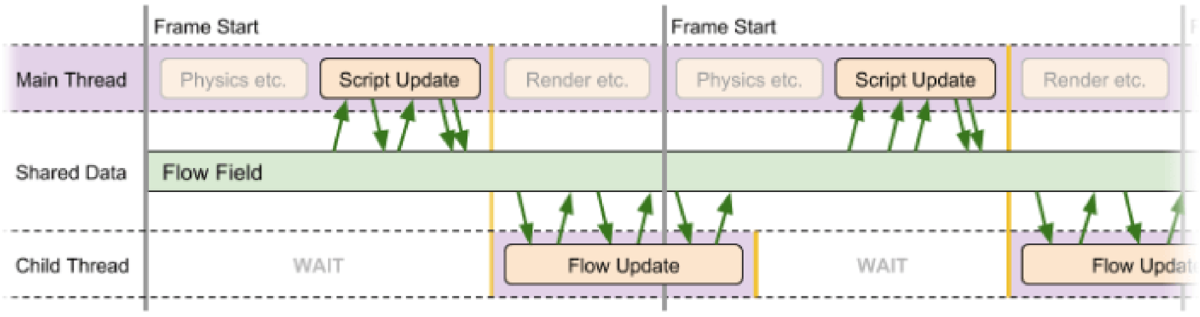
In order to make this true asynchronous computing the main thread must be stopped from blocking by changing the update function to:



```
void Update()
{
    ChildThreadWait.Set();
}
```

(Meredith, 2017)

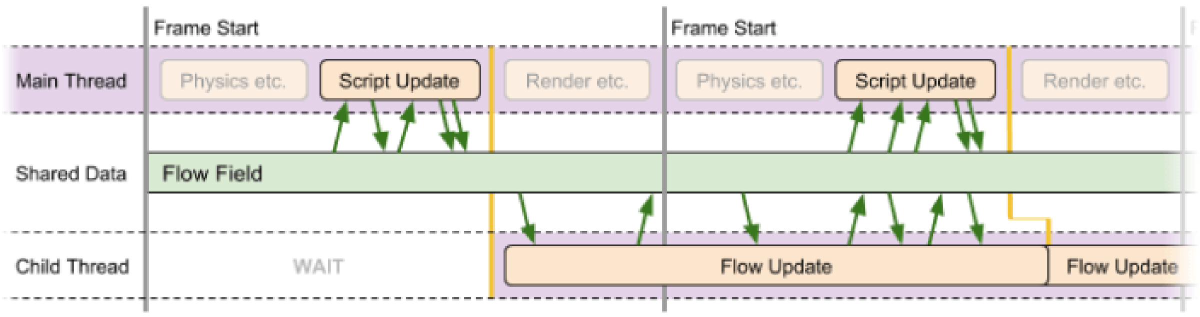
By making the above changes the threads would look like the image below.



(Meredith, 2017)

That is what's aimed however it is not confirmed it will work thus the developer worked on an experiment to identify and resolve the issue.

It can't be determined how long that flow update will take in comparison to other threads, tests could be done to see if the threads finish in time however multithreaded systems are non-deterministic and they can't be proven safe by testing . But they can be made safe by design.



(Meredith, 2017)

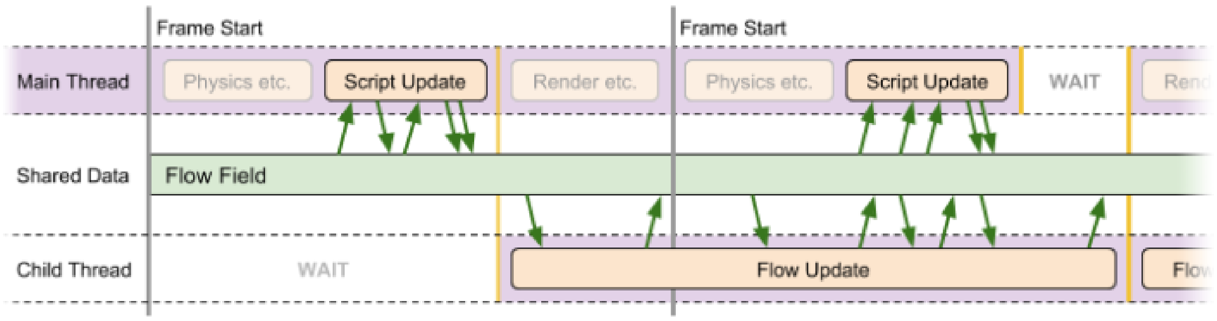
Above is an example of how the process would look like if the flow update takes longer than expected, in the second frame both the threads would be reading and writing from the same flow field which is undefined and problematic behavior, at the same time the main thread is also trying to restart the child thread which would cause multitude of errors.

The first thing to do would be to make sure that the child thread cannot take longer than the main thread ,this is achieved blocking the main thread at the beginning of the update() function .

```
void Update()
{
    MainThreadWait.WaitOne();
    MainThreadWait.Reset();

    ChildThreadWait.Set();
}
```

(Meredith, 2017)

It is to be noted that on the first update the process does not wait as the MainThreadWait variable starts in its “set” state however in the future frames while the child thread is running the main thread will get held up as illustrated in the image below.



(Meredith, 2017)

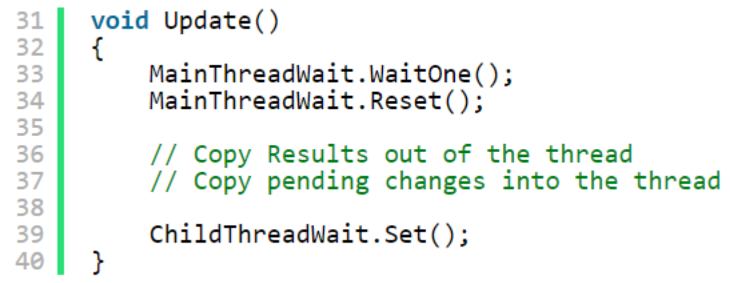
Both threads are in sync however both threads need to work with the same data field in parallel. The approach chosen by the developer is to restructure data so that nothing is directly shared between the two threads directly.

Next, the data would need to be divided into three sets that communicate with the following actions:

- Results of the algorithm which would be used for sampling

- Pending changes, this will be used for adding.
- Working data, which is used to generate various data.

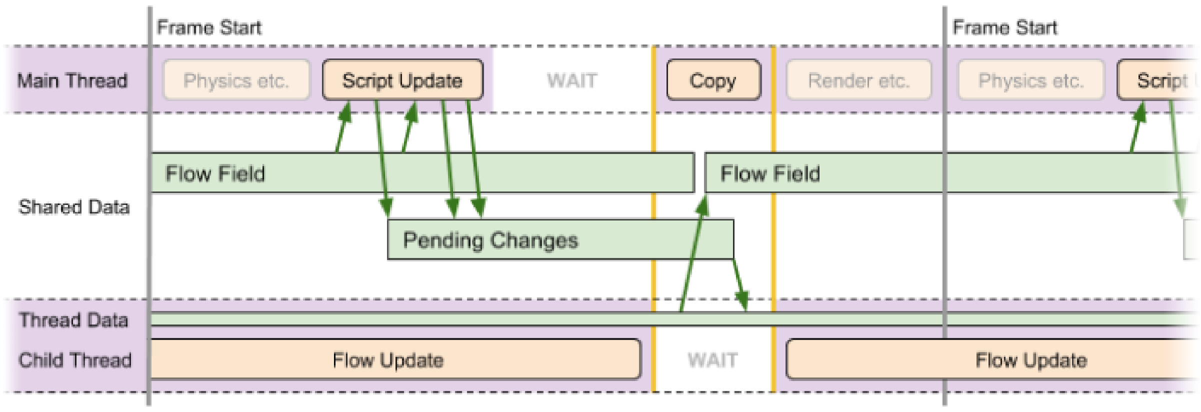
The developer does not go into the specifics of which data structures are used as they would be more application specific however the order of operations in the main thread Update() are as below :



```
void Update()
{
    MainThreadWait.WaitOne();
    MainThreadWait.Reset();

    // Copy Results out of the thread
    // Copy pending changes into the thread

    ChildThreadWait.Set();
}
```

(Meredith, 2017)

With the addition of the code, the thread diagram is now changed more and looks more complicated.

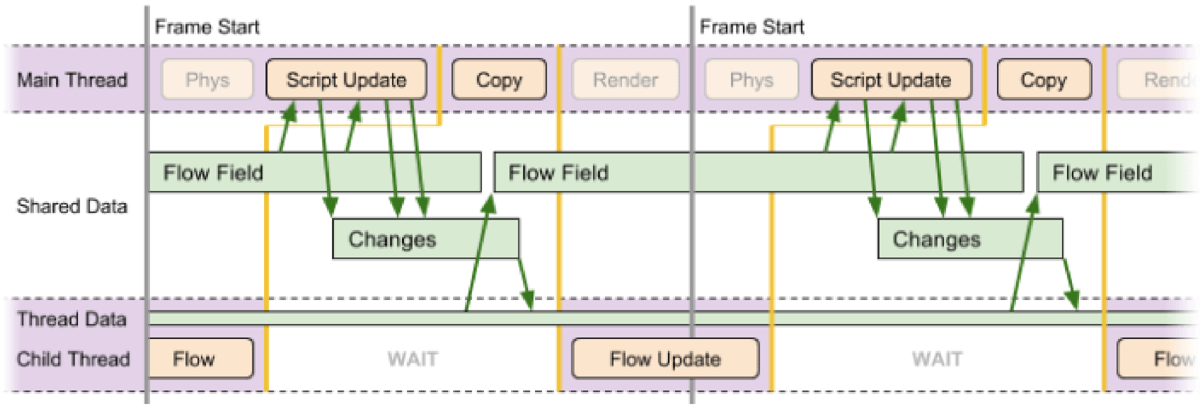


(Meredith, 2017)

As the developer points out that the time axis is not to scale, the operations conducted would be exceedingly swift. The main thread is utilized to do the duplication of data in from and to the child thread as it’s easy to know that there won’t be any conflicts and allows the main thread to assume the role of master in the operation and sync key operations.

It is estimated that the flow update will not take longer than the main thread thus it is extended artificially as part the thought experiment however if the main thread does take longer there won’t be any conflicts between the threads. The child thread is only working with a stream of its own data and thus by definition the two threads are in proper sync and are multithreaded while being thread safe.

Now, the final Asynchronous process would look like the image below.



(Meredith, 2017)

Conclusion

Multithreading in engines such as unity can be time consuming and require the developer to jump through hoops and restructure large amounts of data in order to utilize asynchronous processing however this is a clever trick/ emulation of the process which yield better performance than putting large chunks of codes in coroutines . The method can also be utilized as a way to offload larger calculations to a separate thread and take load off the main thread.

References

Angelini, C. (2016, October 21). [benchmarks]. Retrieved from http://www.tomshardware.com/reviews/multi-core-cpu-scaling-directx-11,4768-8.html

Digital Equipment Corporation. (March). Retrieved March 24, 2018, from https://en.wikipedia.org/wiki/Digital_Equipment_Corporation

Meredith, R. (2017, July 23). [illustration of process]. Retrieved from http://www.richardmeredith.net/2017/07/simple-multithreading-for-unity/

Unity (game engine). (2018, March 23). Retrieved March 24, 2018, from https://en.wikipedia.org/wiki/Unity_(game_engine)