COMP280 – Optimizing a Game Jam

James Berry 1902623

January 3rd, 2021

# Introduction

During the 2020 Global Game Jam last February, I teamed up with another programmer to create a game that we called “Day One” (named after the day one patch that has become so common).

This was a simple game where the player is tasked with solving bugs by typing as fast as possible, with the letters typed appearing on screen as projectiles to defeat the enemies.

I don’t have too many qualms with the efficiency of the project, due to it being a very simple game made for a game jam; but the fact that it was for a game jam also means that it will likely have many inefficiencies that I could correct.

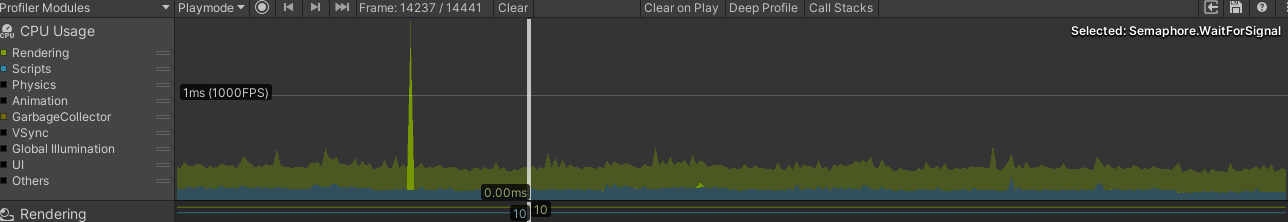
<https://gamesgit.falmouth.ac.uk/users/jb230725/repos/day-one/browse>

This is the project post-optimization, to check the old repository, check

the commit log and/or the screenshots in this document.

# Performance Profiling

To optimize the project I first have to find out what to prioritize. To find this I started the game and played through the first level, ending after waiting a second after the second level loaded:



This profiling reveals that there is no single place that I should be focusing my optimization, I can surmise this because there are no noticeable spikes save for the very occasional spike from Semaphore.WaitForSignal, which to my knowledge isn’t a change in performance caused by any scripts, but rather the main thread waiting for a signal from another thread before moving onto the next frame, and isn’t actually taking up any CPU time.

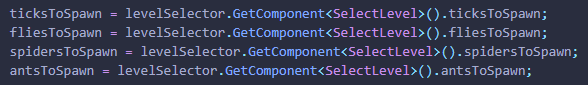
I also used the stats window to check if there was anything that was tanking the graphics (relatively speaking, since there is so little to render there that the lowest framerate that I got was 600), but this only revealed that it took longer to render frames when it was rendering more objects, which makes sense.

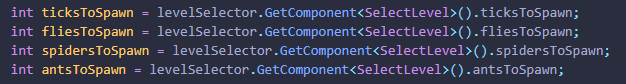
# Performance Improvements

The first thing that I looked at is the MainArray script, this handles all of the changes to the position array that a level is played on, this includes movement (just the actual moving part, not the pathing logic) and spawning.

I started with this because it is where the most logic operations happen within the game.

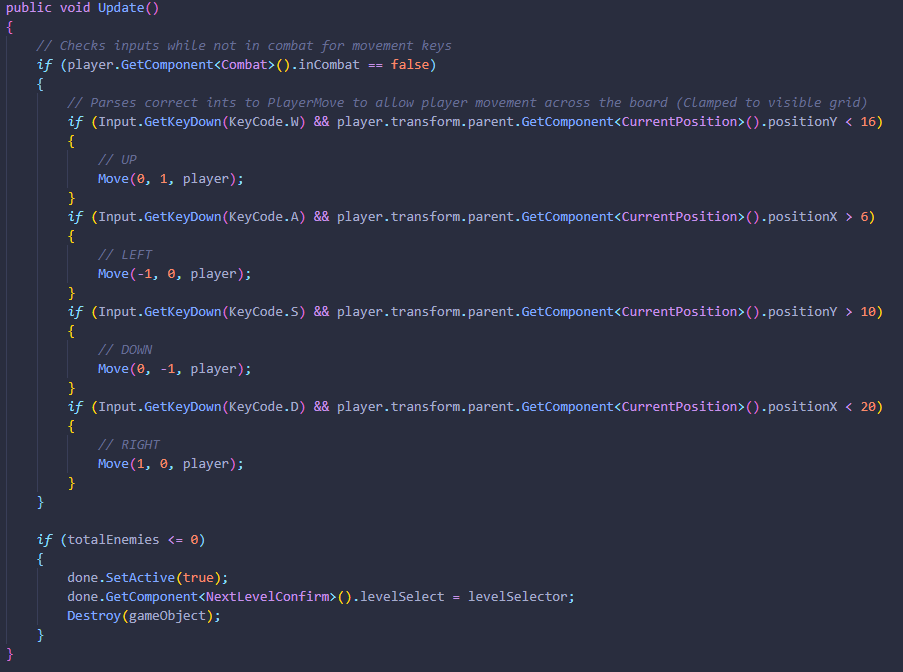
The first thing that I noticed was that I had several variables cached that are only used within the start function, since they are only used in one method, caching them would only slow it down as it would add the time it takes to cache, but not save any time:

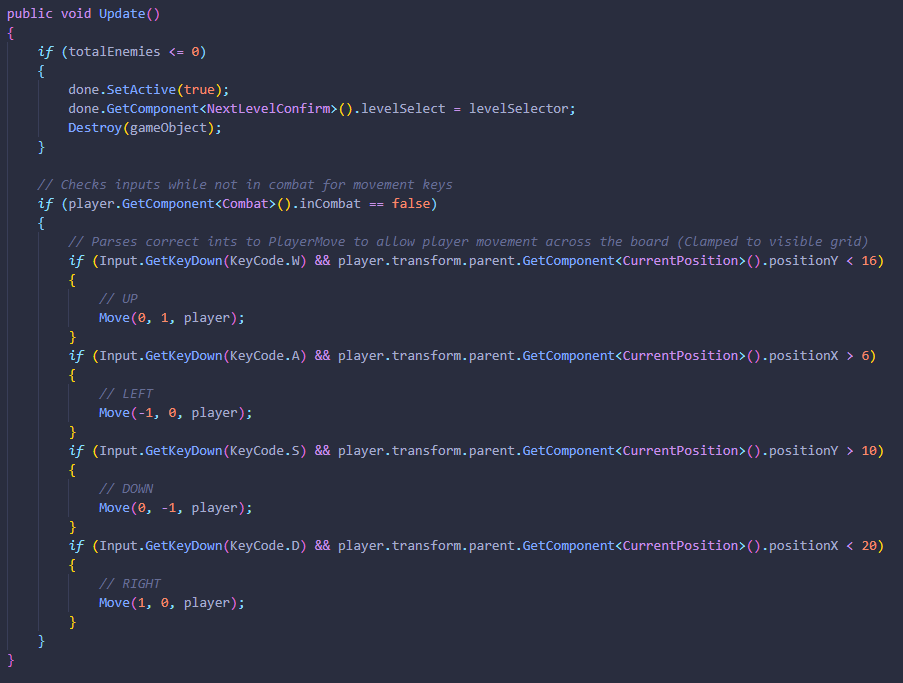




I also noticed a cached array called gridPositions, that isn’t used past setting the values in the Start method which I could remove, although it should be noted that if I had still needed this I would have made it a 2d vector since it is referencing positions in 2d space.

The next method is Update, and while I couldn’t see any major issues I did switch the two if statements:

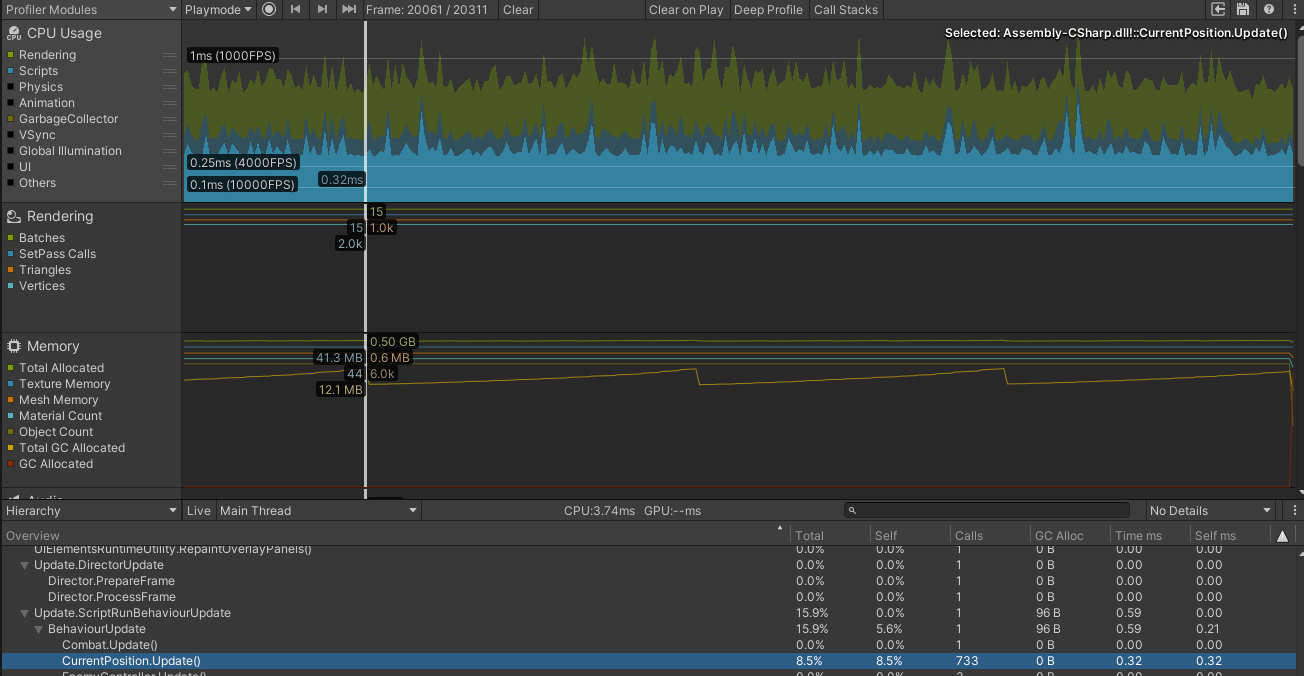




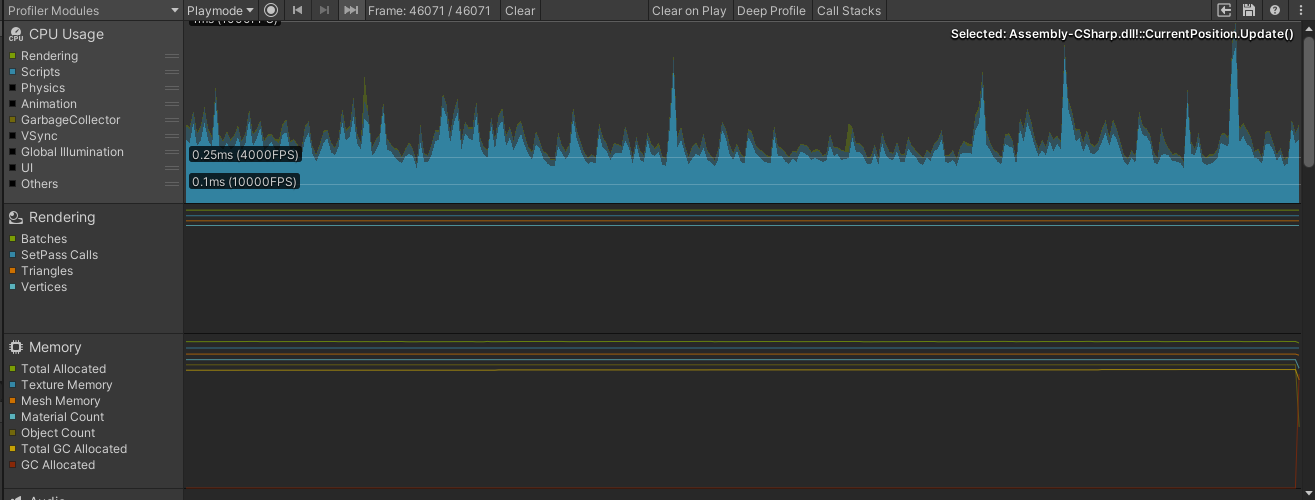
This is because one of the if statements checks if the object should be destroyed, and isn’t dependent on values that could potentially be edited in the other if statement; therefore, by checking if the object could be destroyed first, it could save a miniscule amount of time since it doesn’t have to check the rest of the method. This really is a negligible performance boost, and only happens once per level end at a time in play where it wouldn’t be noticed either way, but it is an improvement nonetheless.

# Impact of the Improvements

Since I had made all of the optimization improvements that I could find in the most labour-some script, I decided to re-do the analysis:



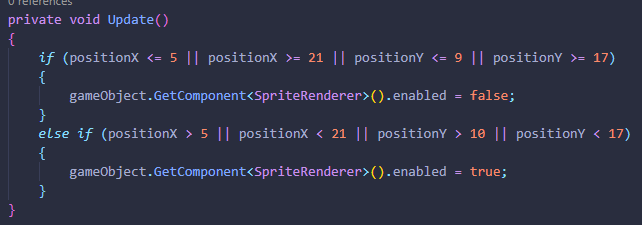
This was a pretty big shock to me, it appeared that I had managed to un-optimize it. However when I looked at the reason for it, it was due to 733 calls to the CurrentPosition.Update method which is way more than expected, which lead me to the conclusion that I had mis-captured the level loading and therefore the instantiation of the entire grid, rather than the gameplay as a whole.

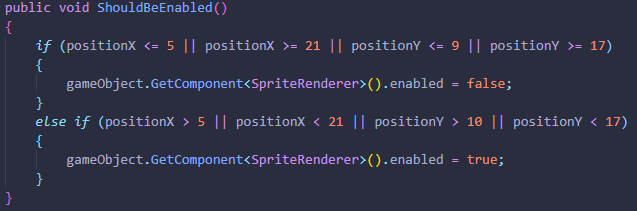


Correctly capturing this revealed that I was right, and had captured the wrong part, however the CurrentPosition.Update method was still being called way too often, considering the nature of the code in it only being necessary on movement, therefore I needed to optimize it.

# Second Pass Performance Improvements

I needed to change the update to be called less often, so I edited the update method to be a generic void method to be called on movement. I was able to do this because the movement wasn’t fluid but done in steps after a certain interval, so I didn’t need to check every frame, but rather every movement step. This meant that I could change the script to just be reference-able by the movement method:

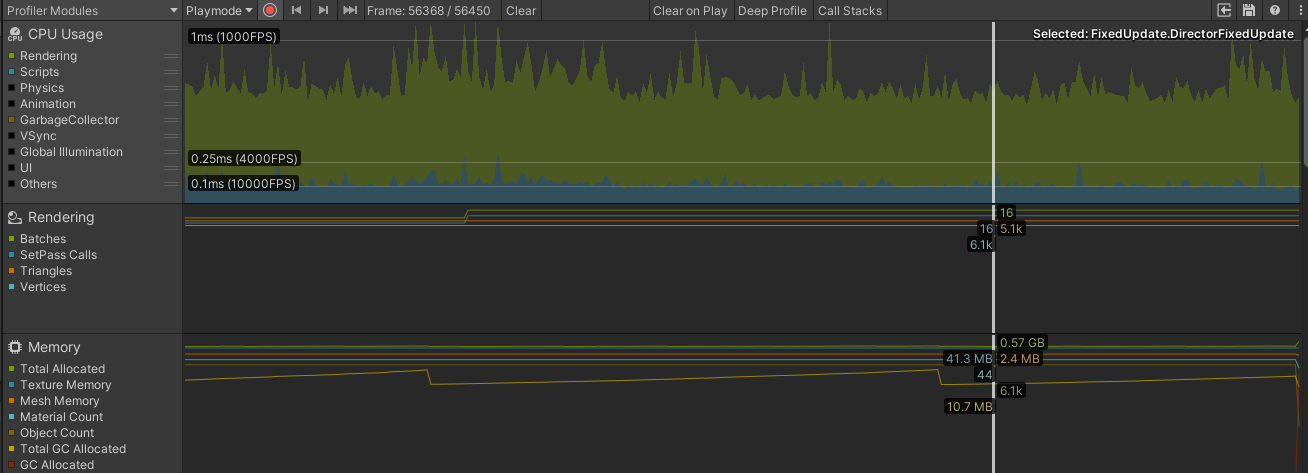




And then added a call for the method when necessary:



# Final Improvement Check



This massively reduced the overhead as it reduced the calls likely by an order of magnitude compared to previously, this means that the total improvement was able to improve script speed per frame by about .15ms from .25ms on average, to .1ms on average.

# Conclusion

# Based on the results of the profiler, I believe that I have managed to drastically improve the game’s performance. With what were previously spikes into 1ms now becoming spikes into .25ms. There are definitely improvements that can be made to more significantly impact performance, however I cannot see any way to accomplish this without a refactor of most of the scripts.