Christian Loera

CSC 461 – Optimized C++

Particle System Write-Up

**System Analysis**

After reviewing the GameParticles’ modifiable source files, I have immediately noticed a few red flags that are a detriment to the performance of the particle system and should be corrected with caution as to not ruin the original functionality of the particle system. Therefore, I will use a top-down approach when refactoring code, starting with the easiest modifications and progressively moving towards large, complex code refactoring. This will be guided by techniques and theories learned throughout the quarter, such as data alignment, Hot Cold structures, compiler setting customizations, and SIMD.

Currently, my times without any modifications are ~40-43 milliseconds for update, ~73-76 milliseconds for draw, and ~113-119 milliseconds in total time. After running the performance profiler to track the CPU usage on my code the percentages mirror the timings displayed on the command prompt. The ParticleEmitter draw function is using 52.45% of the total CPU and the update function is using 29.75% of the total CPU. Upon further inspection of these analytics, I can see that ‘External Code’ (35.23%) and Matrix::operator\* (6.56%) are using the majority of the CPU in draw. In the update function ParticleEmitter::SpawnParticle (3.22%), Particle::Update (7.06%), and std::\_List\_buy::\_Buynode (14.39%) are the major proponents for CPU usage. I am hoping that I could significantly reduce these percentages and cut my total time by half, bringing my total time to ~57 milliseconds.

When looking at the detailed report generated by the profiler, I found that 28% of the CPU usage in the draw function is coming from the glDrawArray function. Since we are not supposed to make large changes to the OpenGL code I will have to refactor code elsewhere in the draw function. Therefore, my next option would have to be optimizing the Matrix::operator\* which is currently at 6.7%. This is due to one line of code that is multiplying 5 matrices at once and storing it into a temporary variable. If I could apply SIMD and implement enough proxies to do all the math operations as one large operation, then I predict I could save about 20-25 milliseconds.

Another function that is being called often is \_free at 3.8%. I’m assuming this coming from delete drawBuffer for loop and the drawBuffer.clear() function call at the bottom of the draw function. Since drawBuffer is declared as a std::list, it makes sense why there is such a performance drain from these lines. This is due to the fact that it is easier to remove nodes at the front of a linked list as opposed the end, which in this case is being done by calling pop\_back(). If drawBuffer was implemented with a vector container instead then this would could easily improve my timing by 15 milliseconds, since vectors are better suited for the pop\_back function. An even better solution would be to remove drawBuffer all together and just use headParticle to traverse through the list of particles, since this is already implemented for me. This would remove the for loop at the bottom of draw, saving me approximately 20-25 milliseconds. Moreover, if the Particle class was implemented as a Hot Cold structure I could see even more gains in performance speed because it would increase traversal speed of the list and result in more data cache hits. In total, I expect to save about ~50-55 milliseconds on draw resulting in a final draw time of ~20 milliseconds.

The result of refactoring the Particle class as a Hot Cold structure could improve my program as a whole and improve the update function by as much as 15%. According the detailed report of the update function, calls to \_Buynode are currently using 14.7% of the CPU which is due to calls to the push\_back function by drawBuffer. As I have previously stated above, if I could refactor the Particle class as a Hot Cold structure and remove the drawBuffer, I predict that my timings for the update function will be reduced by half. This will bring my update function from ~40 milliseconds to as low as 20 milliseconds.

The next most called function in ParticleEmitter’s update function is Particle::Update. Stepping into the Update function I immediately noticed a lot of temporary variables. I’m assuming I could easily remove and might actually be extra bloat in the function to make the program run slower. The report is showing less than 1% CPU usage for all temporaries which does not seem detrimental to the program as a whole, but considering the large number of particles that have to be updated I predict this could reduce my timing for update by 4 milliseconds. Also, if I apply SIMD to the operator\* and operator+ functions of Vect4D I think I could save another millisecond since they are both reported at 0.5%.

The last most called function in update is SpawnParticle, which is currently at 3.3%. It seems like there are very little things that I could improve upon in SpawnParticle because a lot of lines of code are reported to be using less than 1% of the CPU. Two worthwhile things that I could optimize are the construction of particles and the Execute function. Since they don’t have such a huge impact on update I might only save 1 millisecond. Overall, I expect my final update time to be about ~10 milliseconds.

**Work Log**

Please refer to the Work\_Log.txt file in my GameParticle – student depot to get an extensive and detailed view of my progress on the final project.

**Reflection**

Optimizing the particle system was an interesting exercise on applying all concepts taught throughout the semester and yielded some surprising results. I was able to optimize my program by 2.6x where my update function is reporting ~4-5 milliseconds and my draw function is at ~40-41 milliseconds, bringing my total time to ~45 milliseconds. Compared to my original total time prediction of ~57 milliseconds I was able to bring my final total time even lower than I predicted, although, my predicted times for draw and update were off.

My predictions for the draw function were much more optimistic than I anticipated them to be. This was due to the fact that my predicted gains from implementing SIMD and proxies were not as beneficial as I hoped they would be. At best I decreased my time by 5 milliseconds, which was nowhere near my estimate of 20-25 milliseconds. It seems as though I’ve overestimated the impact that the Matrix multiplication operations had on draw. However, my prediction on drawBuffer’s impact on draw was very close. When I refactored drawBuffer to be a vector instead of a list I saved ~10-15 milliseconds. When I removed drawBuffer completely and used headParticle to traverse through all my particles in draw, my performance speed decreased by 5 milliseconds. After refactoring the Particle class as a Hot Cold structure I was able to save another 2 milliseconds. So in total I was able lower my draw time by ~27 milliseconds after implementing these optimizations.

In contrast, my predictions for update were slightly more conservative than I expected them to be. When I removed the drawBuffer from ParticleEmitter and refactored my Particle class as a Hot Cold structure I reduced my update time by ~25 milliseconds, which was a little more than I expected. Also, when I implemented my optimizations for Particle::Update and SpawnParticle functions I was able to save another ~5-7 milliseconds, which was very close to my original predictions. These optimizations all together saved me ~20-22 milliseconds on the update function.

Some surprising optimizations that I didn’t realize would have such a huge impact was removing temporaries and removing invariants from loops. I went through every single file removed all temporaries and invariants from my code and I ended up saving about ~5-10 milliseconds which was a lot more time than I expected to save from such simple changes. This also forced me to think about the data I’m using in my classes and how I’m initializing variables. In the end, my code looked a lot cleaner and readable.

Overall, this project was enjoyable, interesting, and filled in a lot of gaps in terms of all the concepts that we’ve learned throughout the quarter. Optimizing this particle system made me realize just how much I’ve learned about optimization techniques and C++ in general. This was also a good introduction to working with computer graphics which is something that I look forward to learning in future classes.