Brian Bauman
CSC 461

Final Project
Graduate Student Analysis

Overview

As a CS student that has not chosen the game design specialty, this was my first exposure to code involving linear algebra, physics, etc. The first thing I wanted to do when deciding how to tune this system was to understand its flow. Here is a high-level overview of the actions the system performs:

(1) Gather settings, create initial display values, create ParticleEmitter
(2) Loop until cancelled, recording and displaying time taken for updating (a, b) and drawing (c)
  - (a) Perform some linear algebra to update the camera matrix
  - (b) Update each particle in the system
    - (i) Attempt to spawn particles if enough time has elapsed
    - (ii) For each particle, update the position/velocity or remove due to age
    - (iii) Move particles to an STL list
  - (c) Draw each particle in the system
    - (i) Do some GL "goo" to get the appropriate camera matrices
    - (ii) Perform matrix multiplication to calculate the Particle's draw matrices
    - (iii) Set new values for the Particle's draw matrices
    - (iii) Remove particles from the STL list

As you can see, the majority of the improvements to this system will come in either the particle updating or the particle drawing.

Initial profiler results

Before being able to assess how much time individual changes will shave off, you need to understand which sections of the code are being run the most often. Improvements to these sections will have the largest effect on the total performance of the system. By letting the system run while capturing profiling information, the breakdown is as follows:

ParticleEmitter::draw() took roughly 65% of the time. Most of this was external code, likely the GL libraries. Of the ~4% portion that was within our own code, it was largely deletes and frees.

ParticleEmitter::update() took roughly 25% of the time. Most of this time spent was within our own code, largely setting up the STL lists used and updating the Particle objects themselves

This gives insight on the right things to attack first. As certain problems are solved, others will bubble up — this is only the initial state before any changes have been made. Lastly, the initial time taken (in release mode, on my machine) was ~100 ms for ParticleEmitter::draw() and ~33 ms for ParticleEmitter::update(). These numbers will become the denominator for suggestions described below.

Performance improvement suggestions

With an understanding of what the code is doing and where the time is currently spent, the below are a few proposed changes that will speed up the system. For each, I'll try to explain what is slow, what can be done to speed it up, and a rough estimate for the improvement (relative to the initial times listed above). Each estimate will assume that the change is being made in a vacuum, not in conjunction with the other suggestions.

<u>SIMD for Matrix and Vect4D operations</u>

- Problem: Currently, all Matrix and Vect4D operations are done using individual elements of the matrices and vectors, written out and performed individually. This creates an excess of operations that can be condensed if processed together.

- Solution: By using SIMD operations, we can perform multiple operations at once. Because the majority of this math is in the ParticleEmitter::draw() function, whose performance is dwarfed by the external code.

- Result: draw() decreases by 20 ms

<u>Keep Particle objects around in memory rather than deleting and recreating them</u>

- Problem: In the initial version of the code, Particle objects are deallocated in memory when they expire. Later, they are reallocated as new objects. The profiler showed that news and deletes are hit often — removing these calls would increase performance.

- Solution: Rather than delete and create Particle objects, we could leave "expired" particle objects around in a separate list. This would allow us to place them on the "inactive" list when they expire, and pull "inactive" particles off that list when we would like to create anew. Because the total number of particles in the system is capped at the specified amount, a new particle should not need to be created once the system starts up. Much fewer new and delete calls take place.

- Result: update() decreases by 10 ms, draw() decreases by 10 ms

<u>Removing STL lists from the codebase</u>

- Problem: The majority of the profiler's time spent in the update() function (almost 20 of the 25%) was spent setting up and tearing down the STL lists. Cleary, a faster data structure would provide serious performance gains.

- Solution: Replacing the list with a more appropriate STL container or a home-grown data structure would allow the application to only spend the requisite time creating and deleting this information rather than this general-purpose solution that likely include superfluous processing for our application.

- Result: This was a large portion of the code, according to the profiler — almost 20%. I'd expect this change to decrease the update() times by 30 ms

<u>Convert data types to float from double</u>

- Problem: Essentially this entire code is set up to use doubles instead of floats. What this does is allow you to have double the precision for your calculations at the cost of twice the space in memory. Space in memory can translate to decreased performance because the system's cache can not contain as many bigger objects (doubles) as it can smaller objects (floats). It can contain only half as many.

- Solution: Switch from doubles to floats. As long as we don't need the precision of a double, we shouldn't be paying it's performance tax. As Keenan says, "We went to the moon with floats".

- Result: Because this one plays mostly with caching, I don't think the result will be as notable. I'd estimate 0.5 ms at most from both update() and draw()

<u>Proxy objects allow for purposed multiplication of multiple matrices</u>

- Problem: In the code, each iteration requires matrix multiplication involving five matrices. The system currently interprets this as best it can — (((((M * M) * M) * M) * M) * M) — requiring the matrix multiplication code to be executed five times.

- Solution: Instead, if a proxy object were used, we could save the evaluation of this result until we stored all variables from all matrices into a single object. This would allow us to perform a single multiplication. With the use of SIMD, this greatly reduces the number of operations that need to occur.

- Result: The draw() function where the majority of this code exists was the heaviest hitter in the profiler, though a large portion of it was in external code. I expect a maximum of 20 ms reduced with this fix, probably closer to 10 ms.

Other considerations

Here I list a few other performance considerations that I didn't describe in detail due to the length constraint of this report. They can all be considered to be less impactful than the suggestions described above.

- Object constructors don't set all properties, but should
- Reorganization of the structure of the Vect4D for data alignment that allows SIMD usage
- Reorganization of the structure of the Particle object for better cache usage when looping through variables

Final thoughts

At a certain point, the code will take time to execute — there is no way to optimize it below a certain threshold and it become fruitless. By using tools like the profiler, and by comparing resultant times with initial times, the best approach is likely a guess and check approach. One of the hardest questions to answer in this report was the estimate of the time saved with each change, especially just from viewing the code! I hope my numbers end up being gross underestimates of time saved… for the Pot Noodle, of course!