Erik Parker
CSC461
2022 Fall
Particle Project Analysis

Abstract

The particle simulator optimization project is a treasure trove of optimization possibilities. This paper will briefly outline and categorize possible optimizations, provide justification, and predict the impact to the run time. Numeric measures may be hard to predict so this paper will measure in “major”, “moderate”, “moderate-small”, “small”, and “can be ignored” and focus on the categories of “major”, “moderate”, and “can be ignored”.

Summary of Potential Optimizations

Below is the abbreviated list of the potential optimizations with the full list in appendix A

- Major
  - Sequential memory allocation
  - Do not create a second list
  - Limit multiple run throughs of the list
  - Use Proxy+SIMD to speed up M * M * M * M * M
- Moderate
  - Set default constructors to =default (Particle, Vect4D, Matrix)
  - Change all datatypes from double to float
  - Remove all unneeded data variables from Particle.h
- “Can be ignored”
  - Avoid effort to optimize because they are only required to run once or twice per run

Category: Major

This category is defined as optimizations that could lead to >30% speed boost. These are the first items to tackle since they can impact the list of further optimizations.

Sequential memory allocation

The baseline codebase creates a new particle object on the heap and adds it to a linked list all within a spawn loop (ParticleEmitter::SpawnParticle() line 49). This is a reliable technique but slow because a “new” allocator and the Particle constructor is called 200,000 times. Then the next and prev pointer was to be update in the current, prev, and next particle in the list, another 4+ updates per particle. Once created, walking through the list requires pointers to jump around memory and a hot/cold optimization won’t help because all the data (no need for a “cold” object) of every element is used (can’t just iterate through the list and grab the data you want).

Pre-allocating the whole 200,000 particle block ahead of time with 1 “new” call saves 199,999 calls to “new” (the memory allocator is the slower part compared to the constructor). Then when you “spawn” particles you just use the placement new operator on the next available memory. Iterating through the list would now only require a single Particle pointer and for you to increment it by 1 particle length i.e. ptr++. This allows the compiler/OS to further optimize the movement from one object to the next.

Do not create a second list

The link list of particles starting at ParticleEmitter::headParticle is all that should be required. All other containers of particles (like ParticleEmitter::drawBuffer) don't persist beyond the cycle it was created and the operations (that aren't unnecessary) that use these particles don't care which list the data come from. Not only is having more than one list inefficient from a memory perspective, but it takes additional time to create (and allocate new memory) and populate the list with clones from the primary linked list. And on deletion, the list must be cleared which further increases the computational cost as it must clear each node. I expect this optimization will result in one of the greatest speed boosts.

Limit multiple walks of the list

There are 6 occurrences of the list being iterated (4 loops and 2 calls to std::list::clear()) per cycle, all occurring in ParticleEmitter.cpp. If we eliminate all other lists or contains and only use the link list starting at ParticleEmitter::headParticle then there no reason to have more than 2 iterations; 1 in ParticleEmitter::draw() and 1 in ParticleEmitter::update(). All other iterations are used to populate the ParticleEmitter::drawBuffer, clear it, or manual remove each element. This will save on 200,000 times 4+ operations.

Use Proxy+SIMD to speed up M * M * M * M * M

The calculation of 5 matrixes multiplied together using the Matrix::operator*() operator results in 4 calls to that function representing 9 Matrix constructor calls (8 default + 1 copy), 448 temporary variables (7*16*4), 256 float multiplications (16*4*4), and 192 float additions (16*3*4). This is all per particle per cycle.

The proxy alone will remove 7 of the Matrix constructor calls. Adding SIMD will shrink the calls down to 64 _mm_set_ps1, 64 _mm_mul_ps, and 48 _mm_add_ps1 which each take slightly longer but handle 4 calculations at once. I expect this to be one of the other primary speed booster due to the number of saved operations.

Further optimization can be applied by specializing the code for this specific operation. This would involve analyzing the 5 matrices and determining how their initial values are represented in the final answer. Many of the calculations are unneeded since there are so many zero valued elements.

Category: Moderate

This category is defined as optimizations that could lead to a speed boost between 10% and 30%. These boosts could further magnify the “major” category optimizations and should also be focused on at the start.

Set default constructors to =default (Particle, Vect4D, Matrix)

The baseline code has default values defined in the constructor of the objects but these values are never used or needed in the codebase because immediately afterwards the object is being assigned a new value. Setting those default values in the constructor takes extra time in the explicit calls to the constructor but also in the implicit calls during mathematical operations or during function calls and value returning.

The default constructors should be blank with the expectation that their values will be set later. Taking this a step further, set the function to be “=default” and let the compiler optimize even further. This will further boost the optimization results of the Sequential Memory Allocation by no longer having 200,000 constructions with lots of data setting, instead nothing will be called from the constructor speeding along the allocation process.

Change all datatypes from double to float

This change may seem simple, but it should not be underappreciated. Operations with float generally take less power than with a double, floats take up less space, and for most applications, a float provides more than enough precision. Potentially the greatest benefit is that since floats are smaller, more operations can be performed in parallel and in SIMD operations.

Remove all unneeded data variables from Particle.h

If the Sequential Memory Allocation optimization is implemented, then the next and prev member variables are no longer needed. Starting with the Particle::diff_Row vectors, they are never changed outside of the drawBuffer list that was remove. Line 64 of Particle.cpp is the only time the variables are used and it will always result in MatrixScale==0 since the diff_row values are always all zero and the Determinant of a zero matrix is 0. Therefore, MatrixScale can be replaced by a 0 and the diff_Row variables can be removed. The same approach can be applied to curr_Row and prev_Row variables and those can get removed too.

This will lead to a decent speed boost since matrix calculations are so much greater than float. This also increases the effectiveness of the Sequential Memory Allocation and list iteration since the object size has more than cut in half.

Category: Can be ignored

When optimizing, it’s also equally as important to know where to avoid speeding resources. In a program where 200,000 objects are iterated and modified many times a second, places in the code that aren’t in that path can generally be ignored. Examples of this can be found in Matrix::Determinant() and Particle::GetAdjugate(). Both are ugly but both are only called twice (also some code motion) so there’s no real reason to look into improving this code.

Depending on the optimization outcome, the code involved in the creation of Particle might also be de-prioritized since is such a small fraction of the overall run metrics. An example of this is ParticleEmitter::Execute(). Its kind of ugly but only runs 200,000 times per 20 seconds. This is compared to the code that runs 200,000 time many times a second.

Appendix A: Full list of potential Optimizations

Potential Optimizations:

- Major
    - Sequential memory allocation
    - Limit multiple run throughs of the list
    - No reason to have multiple lists
    - Convert M * M * M * M * M
- Moderate
    - Set default constructors to =default (Particle, Vect4D, Matrix)
        - Nothing in the code relies on any of these object types to have a preset value.
        - The existing value is immediately overwritten to something else
        - SPEED UP: less time spent setting default values that aren't needed.
    - Change all datatype double to float
        - Smaller variables and less calculations
    - Remove all unneeded data variables from Particle.h
        - prev_Row, diff_Row, and curr_Row vect4D object are never actually used.
        - next and prev can be removed with sequential memeory
- Moderate-small
    - Limit use of sin(), cos(), sqrt()
    - Vector union with __m128
        - Ability to create a Vect4D with "1" variable instead of 3 or 4
        - Direct compatibility with SIMD
    - Declare Big 4 and set to "= delete", "=default", or define custom implementation; in that order of preference.
    - Use initializer list in all constructors
    - Reorganize Particle member variables for pad and alignment optimizations
- Small
    - Remove Enum.h
    - Remove Vect4D::set()
    - Remove Vect4D::operator[]
    - Remove Matrix::operator[]
- can be ignored
    - Avoid effort to optimize because they are only required to run once or twice per run
        - Matrix::getTransMatrix()
        - Matrix::Determinant()
        - Matrix::GetAdjugate()
        - Matrix::Inverse()

Appendix B: Sine Cosine Optimization Justification

Goal: Provide evidence that this is a High-Fidelity optimization that leads to faster run times without any sacrifice to data accuracy.

TBS (to be supplied)