CSI 370 — Final Project Write-Up

Conner Root

Contents

Part I
Project Description

1 Introduction

For my project, I intend to update the physics library I made as part of Game Physics to try to get it to run faster. Built for Unity3D, the library is written in C++ and is linked to Unity3D using a DLL and a C# script used as a wrapper to facilitate communication between them. More specifically, I will be adding assembly to the vector struct I use to facilitate force calculations.

By replacing all or parts of supporting functions with assembly equivalents, I hope to make it run more quickly with minimal (preferably zero) loss in accuracy. However, because this is a *game* physics engine not meant for very realistic simulations, some loss of accuracy is acceptable.

2 Methodology

I will follow a 3-Part process. In parts 1 & 2, I run some initial tests outside of Unity to determine if a new implementation would have theoretical benefits. If it does I may then test in-engine to see if there are actual practical benefits to the new approach.

Part 1 - Initial Tests:

1. Add current version of the function in question to an empty C++ project
2. Run the version **N** times with random inputs, time using the `chrono.h` library
3. Record the results
4. Repeat steps 2 & 3 using the next algorithm

Part 2 - Choosing the Algorithm:

When all potential replacement candidates have been tested for a particular piece of the library, their relative "replacement value" **R** will be assessed using the equation:

$$R = 0.6s + 0.4p$$

Where **s** is speed and **p** is precision relative to the original.

Part 3 - Compare & Implement In-Engine:

1. Replace relevant library code with the best algorithm from Part 2

2. On the Unity side, add timers around the i/o for the library
3. Run a predetermined simulation in a Unity scene
4. Record results and compare to the original

3 Challenges

There are two potential challenges that I foresee in this. First, when implementing this low-level code, the performance benefits may vary from processor to processor. Even if the processor supports the instructions, hardware fragmentation can still end up yielding different results, meaning that I may find or miss results that you would (not) find on another processor. To compensate for this, I will try to test it on as many different machines to see if results are consistent.

Additionally, if the gains in speed are not significant, there may be other software limitations from passing the data back and forth and processing it on the Unity side, so that even if there are technically gains in speed, they may not result in noticeable improvement in FPS. To try to avoid this, the final test simulation will push Unity to it's limits, because if it can run the normal version smoothly, no improvements will be particularly noticeable.

Part II
Testing

4 Initial Tests

For my initial tests, I tried various methods to speed up the square root, dot product and magnitude functions, as well as the addition and subtraction operators for the vector struct I have using inline Assembly and compiler intrinsics. The were all run in a Visual Studio project set to Release.

4.1 Square Root

4.1.1 C++ Method

```
float stdSqrt(float inVal) {
    return sqrt(inVal);
}
```

Category	Average	Std Dev
Time (μs)	0.030	0.215
Accuracy (%)	100.0	0.000

Table 1: N = 50,000

4.1.2 Fast Inverse Square Root

For this one, I tested the original code, but did not convert to any kind of inline assembly because it was out of my depths, and I still do not entirely understand how it works, to actually do so. Regardless, I found the original *Quake III* code interesting. There is an intrinsic for an inverse square root but I did not do anything with it as I do not use the inverse of the square root.

```
long i;
float x2, y;
const float threehalfs = 1.5f;

x2 = inVal * 0.5f;
y  = inVal;
i  = *(long*)&y;    // evil floating point bit level hacking
i  = 0x5f3759df - (i >> 1);  // what the fuck?
y  = *(float*)&i;
y  = y * (threehalfs - (x2 * y * y));    // 1st iteration

return 1.0f / y;
```

Category	Average	Std Dev
Time (μs)	0.033	0.275
Accuracy (%)	99.91	0.057

Table 2: N = 50,000

4.1.3 Newton Method using FPU

The Newton Method is a way to find an approximation of square root, but approximations often rely on a good initial guess. Using half of the value is one approach, but I used the average value of the ratio between the number and it's square root for 5 ranges using the following process:

$$\int_b^a \frac{1}{\sqrt{x}} dx = 2\sqrt{x}|_0^1 = \frac{2}{\sqrt{a}+\sqrt{b}}$$

This way, my initial guess will typically be much closer to the actual answer compared to just taking half.

```
// find initial guess
float initGuess;
if (inVal <= 1.0f)
    initGuess = inVal * 1.519f;
else if (inVal <= 49.0f)
    initGuess = inVal * 0.250f;
else if (inVal <= 144.0f)
    initGuess = inVal * 0.105f;
else if (inVal <= 400.0f)
    initGuess = inVal * 0.063f;
else
    initGuess = inVal * 0.040f;

float val, quart=0.25f, a;
_asm {
    fld inVal
    fdiv initGuess
    fadd initGuess
    fstp a

```

```
    fld inVal
    fdiv a
    fstp ST(1)

    fld quart
    fmul a
    fadd ST(0),ST(1)
    fstp val
}

return val;
```

Category	Average	Std Dev
Time (μs)	0.067	0.140
Accuracy (%)	99.73	0.472

Table 3: N = 50,000

4.1.4 Inline Intrinsics

```
float inline asmFastSqrt(float inVal)
{
    return _mm_cvtss_f32(
            _mm_sqrt_ps(_mm_set1_ps(inVal))
        );
}
```

Category	Average	Std Dev
Time (μs)	0.030	0.046
Accuracy (%)	100.0	0.000

Table 4: N = 50,000

4.2 Vector Addition

Note: From here forward, no approximations are used and therefore accuracy was always 100%, so accuracy is excluded from the result tables.

4.2.1 C++ Method

```
forceVec forceVec::operator+(forceVec rhs)
{
   forceVec newV;

   newV.x = x + rhs.x;
   newV.y = y + rhs.y;
```

```
    newV.z = z + rhs.z;

    return newV;
}
```

Category	Average	Std Dev
Time (μs)	0.031	0.046

Table 5: N = 1,000

4.2.2 FPU

```
float nX = x, nY = y, nZ = z;

_asm {
    fld nX
    fadd rhs.x
    fstp nX

    fld nY
    fadd rhs.y
    fstp nY

    fld nZ
    fadd rhs.z
    fstp nZ
}

return forceVec(nX, nY, nZ, 'f');
```

Category	Average	Std Dev
Time (μs)	0.041	0.049

Table 6: N = 1,000

4.2.3 Intrinsics

```
float thisVec[4] = { x,y,z,0 };
float otherVec[4] = { rhs.x, rhs.y, rhs.z,0 };

_mm_store_ps(thisVec,
          _mm_add_ps(_mm_load_ps(thisVec),
                 _mm_load_ps(otherVec))
         );

return forceVec(thisVec[0], thisVec[1], thisVec[2], 'f');
```

Category	Average	Std Dev
Time (μs)	0.034	0.047

Table 7: N = 1,000

4.2.4 SSE Packed Operations

```
float const thisVec[4] = { x,y,z,0 };
float const otherVec[4] = { rhs.x, rhs.y, rhs.z,0 };
float newVec[4];

_asm {
    movups xmm0, [thisVec]
    movups xmm1, [otherVec]
    addps xmm0, xmm1
    movups [newVec], xmm0
}

return forceVec(newVec[0], newVec[1], newVec[2], 'f');
```

Category	Average	Std Dev
Time (μs)	0.040	0.049

Table 8: N = 1,000

4.3 Vector Subtraction

4.3.1 C++ Method

```
forceVec forceVec::operator-(forceVec rhs)
{
    forceVec newV;

    newV.x = x - rhs.x;
    newV.y = y - rhs.y;
    newV.z = z - rhs.z;

    return newV;
}
```

Category	Average	Std Dev
Time (μs)	0.031	0.046

Table 9: N = 1,000

4.3.2 Intrinsics

```
float thisVec[4]  = { x, y, z, 0 };
float otherVec[4] = { rhs.x, rhs.y, rhs.z, 0 };

_mm_store_ps(thisVec,
          _mm_sub_ps(_mm_load_ps(thisVec),
                 _mm_load_ps(otherVec))
         );

return forceVec(thisVec[0], thisVec[1], thisVec[2], 'f');
```

Category	Average	Std Dev
Time (μs)	0.033	0.047

Table 10: N = 1,000

4.3.3 SSE Packed Operations

```
float thisVec[4]  = { x, y, z, 0 };
float otherVec[4] = { rhs.x, rhs.y, rhs.z,0 };

_asm {
    movups xmm0, [thisVec]
    movups xmm1, [otherVec]
    subps xmm0, xmm1
    movups[thisVec], xmm0
}

return forceVec(thisVec[0], thisVec[1], thisVec[2], 'f');
```

Category	Average	Std Dev
Time (μs)	0.035	0.048

Table 11: N = 1,000

4.4 Dot Product

4.4.1 C++ Method

```
float dotProduct(forceVec a, forceVec b)
{
    return a.x * b.x + a.y * b.y + a.z * b.z;
}
```

Category	Average	Std Dev
Time (μs)	0.030	0.046

Table 12: N = 1,000

4.4.2 Intrinsics

```
float thisVec[4]  = { a.x, a.y, a.z, 0 };
float otherVec[4] = { b.x, b.y, b.z, 0 };
float result;

_mm_store_ps(&result,
             _mm_dp_ps(_mm_load_ps(thisVec),
                _mm_load_ps(otherVec), 0xFF)
            );

return result;
```

Category	Average	Std Dev
Time (μs)	0.031	0.046

Table 13: N = 1,000

4.5 Vector Magnitude

4.5.1 C++ Method

```
float forceVec::magnitude()
{
    return sqrt(x * x + y * y + z * z);
}
```

Category	Average	Std Dev
Time (μs)	0.030	0.046

Table 14: N = 1,000

4.5.2 Intrinsics

Magnitude is a dot product with itself and then a square root.

```
float thisVec[4] = { x,y,z,0 };

return _mm_cvtss_f32(
            _mm_sqrt_ps(
                _mm_dp_ps(
                    _mm_load_ps(thisVec),
```

```
                        _mm_load_ps(thisVec), 0xFF))
            );
```

Category	Average	Std Dev
Time (μs)	0.029	0.045

Table 15: N = 1,000

5 Choosing the Algorithm

For Part 2, I put every method into a spreadsheet to evaluate how good or bad of a replacement they are. See the full table of average time to execute, the average accuracy, and then the comparison to the C++ method for both, and then finally the ***R*** value calculated using the equation listed in Part 1.

Approach	Time (μs)	Accuracy	S	P	R
Square Root					
C++ Method	0.03	100.0	1.00	1.00	1.00
Inline Intrinsic	0.03	100.0	1.00	1.00	1.00
Fast Inverse Sqrt	0.03	99.91	0.92	1.00	0.95
My Ass Newton	0.07	99.73	0.45	1.00	0.67
Vector Addition					
C++ Method	0.03	100.0	1.00	1.00	1.00
Intrinsics	0.03	100.0	0.91	1.00	0.95
SSE Packed Ops	0.04	100.0	0.77	1.00	0.86
FPU Operations	0.04	100.0	0.75	1.00	0.85
Vector Subtraction					
C++ Method	0.03	100.0	1.00	1.00	1.00
Intrinsics	0.03	100.0	0.92	1.00	0.95
SSE Packed Ops	0.03	100.0	0.87	1.00	0.92
Dot Product					
C++ Method	0.03	100.0	1.00	1.00	1.00
Intrinsics	0.03	100.0	0.99	1.00	1.00
Vector Magnitude					
C++ Method	0.03	100.0	1.00	1.00	1.00
Intrinsics	0.03	100.0	1.04	1.00	1.02

Table 16: Full Results

While by and large the methods I tried to replace the old C++ with are slower, there are some that resulted in virtually no change, and two where I had beaten the compiler! There are a few methods that

are slower, but the intention is to insert assembly, so slower methods will still be used if there is not a replacement that is faster.

It should also be noted that while I make direct comparisons between the methods, the averages themselves also sometimes had significant variation so this data should not be used outside of this paper for a variety of factors that are discussed in a later section.

5.1 Square Root

Approach	R	Time (μs)	Change
C++ Method	1.000	0.0302	0.0%
Inline Intrinsic	1.003	0.0301	-0.5%
Fast Inverse Sqrt	0.954	0.0327	8.2%
My Ass Newton	0.671	0.0666	120.4%

Table 17: Expanded square root results

Thankfully, the intrinsic method has the highest R value, as square root is only ever used in magnitude, and intrinsics are easy to incorporate without having to dedicate a whole new function. While it has been deemed better using the R function, only a 0.5% reduction in speed could very easily be a difficulty in proper timing when it comes to a nanosecond of difference.

5.2 Vector Addition

Approach	R	Time (μs)	Change
C++ Method	1.000	0.0307	0.0%
Intrinsics	0.947	0.0337	9.8%
SSE Packed Ops	0.864	0.0397	29.3%
FPU Operations	0.848	0.0411	33.9%

Table 18: Expanded vector addition results

Once again, intrinsics seem to be the best option. This time though, my new method resulted in a 9.8% increase in run-time.

5.3 Vector Subtraction

Approach	R	Time (μs)	Change
C++ Method	1.000	0.0305	0.0%
Intrinsics	0.951	0.0332	8.9%
SSE Packed Ops	0.924	0.0349	14.4%

Table 19: Expanded vector subtraction results

Very similar to addition, intrinsics are the best option but have a slower run-time. Compared to addition though, the subtraction method only has a 8.9% increase, and you may notice the SSE Packed Operations also was much faster than in addition. This is partially due to my own improvements over the course of the project, where some optimizations do not make it to methods developed earlier in the project.

5.4 Dot Product

Approach	R	Time (μs)	Change
C++ Method	1.000	0.0303	0.0%
Intrinsics	0.996	0.0305	0.7%

Table 20: Expanded dot product results

Due to the trend of intrinsics being the best option, I only tested intrinsics with dot products. The result was a 0.7% increase in run-time, which will likely end up being largely negligible.

5.5 Magnitude

Approach	R	Time (μs)	Change
C++ Method	1.000	0.0303	0.0%
Intrinsics	1.025	0.0291	-4.0%

Table 21: Expanded magnitude results

Finding a vector's magnitude with intrinsics resulted in a 4% reduction in run-time. Part of this is due to the average deviation, and part may be that when dot product and square root intrinsics are combined you receive a better benefit.

6 Choosing the Algorithm

Unity does not have a high-precision clock making it difficult to benchmark changes, so timing is done using ticks for this piece of testing. Also, testing individual functions would be irrelevant, so I will test using multiple force equations at once that each use an assortment of the vector equations. My physics project also was less developed than I thought, so no FPS testing will occur as originally intended as I do not have the time to devote to updating the library and Unity project.

In Unity, testing is done by having 3 blocks with Drag, Friction, Gravity and Normal forces turned on, to try and simulate more of an actual load, where multiple objects are calculating multiple forces each frame. Stats are gathered by wrapping the function call in the library with a C# stopwatch and returning the number ticks. The simulation will be ran for $\tilde{1}5$ seconds before printing the results.

Note: stats for Gravity are not shown because it does not use any of the updated vector equations.

6.1 Drag

While subtraction is expected to be slightly slower, magnitude is expected to be faster and is called more, so i had expected that there will be slight improvements here. There was a small reduction in performance, however.

Functions/Operators	# Calls
Subtraction Operator	1
Magnitude	3

	N	Ticks
Before Assembly	2346	7
After Assembly	2319	8

6.2 Friction

Functions/Operators	# Calls
Addition Operator	1
Magnitude	3
Dot Product	1

For friction, the number of calls for each function/operator depends on other factors, and therefore it was difficult to guess the performance change, but I believed it would slow, as the new addition and dot product methods are slower than the original. Performance did not end up changing.

	N	Ticks
Before Assembly	2346	14
After Assembly	2319	14

6.3 Normal

Functions/Operators	# Calls
Magnitude	1
Dot Product	1

Normal force, because the dot product only made a small increase in speed, and magnitude made a more significant reduction in speed, I expected an improvement in performance. I ended up being right on this one.

	N	Ticks
Before Assembly	2346	9
After Assembly	2319	8

Part III
Conclusion

7 Thoughts on Results

I do not believe that my work has led to any tangible results apart from a moderate improvement to my ability to use intrinsics. While the average ticks went down one for normal force, it went up one for drag, and since Friction, that makes the most calls to modified functions, did not change at all, my best guess is that either there was no impact on performance, or it was too little to detect at the level of testing I was doing. 1 tick is not a big enough to really know if there was an impact on performance from anything that I did.

8 Problems With Research

I think the problems with my researched can be summed up in the following 3 major lacks.

8.1 Lack of Accurate Testing

- In the testing program, I drop the first result because it takes significantly longer, but that may be the more accurate result, as a real run would not drop the first result
- Running a single equation 1000+ times in a row likely makes it easy for CPU optimizations that use predictive techniques

8.2 Lack of Robust Testing

- Could have also tested a sequence of instructions to to make the tests more difficult
- Very few "particles" (my equivalent to a Unity3D rigidbody) used

8.3 Lack of Earlier Preparation

- Updating vector structs to use float arrays before performing any logic to reduce overhead going into the assembly

9 A Few Things to do Better

Converting from structs to even just `float[3]` arrays or a better way of packing the data may simplify passing data into intrinsics/inline assembly that would reduce steps needed to get the output.

My methodology was also poor. There was a varying number of other processes running on my machine, which could impact benchmarks, in addition to the other issues with how data was actually collected. Testing on a machine with nothing else running, with more prep and research on proper testing processes would be required.