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.

Summary: Due to the way that particles are created, the fact that the starting value for a particle's rotation is the same for all particles, and the fact that for a given cycle, all particles' rotation value is increased by the same amount, the end result is that there are groupings of particles (~1500 each) with the exact same rotation value and thus the same sin() and cos() values. Instead of recalculating the sin() and cos() value for each particle, you can just reuse the values for each particle in the grouping. This optimization is also made possible by the fact that the list is ordered oldest to youngest so the rotation value will consistently decrease between groupings as you iterate through the list.

Below will be the breakdown of each of the topics covered above. There will also be code snippets (taken from the baseline code to show that this optimization is possible using the original structure) and runtime data plots.

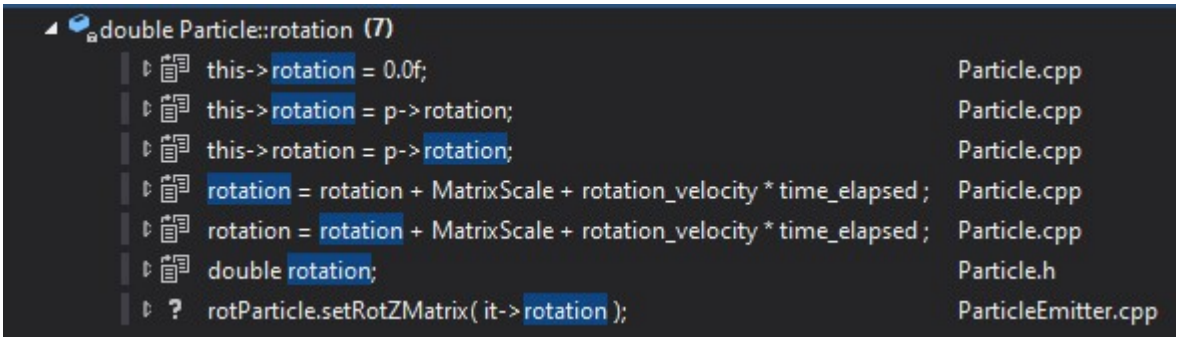
The way that particles are created: Looking at ParticleEmitter::update(), we can see that the number of times the SpawnParticle() function is called is determined by time_elapsed (since spawn_frequency is fixed). SpawnParticle() creates one Particle at a time but the loop runs many times per call to ParticleEmitter::update(); about ~25,000 times on the first call, ~14,000 in the third call, and about ~1600 on all other calls till there are 200,000 particles.

```
void ParticleEmitter::update()
{
	// get current time
	double current_time = globalTimer.GetGlobalTime();

	// spawn particles
	double time_elapsed = current_time - last_spawn;

	// update
	while (spawn_frequency < time_elapsed)
	{
		// spawn a particle
		this->SpawnParticle();
		// adjust time
		time_elapsed -= spawn_frequency;
		// last time
		last_spawn = current_time;
	}


	// total elapsed
	time_elapsed = current_time - last_loop;

	...

}
```

The starting value for a particle's rotation is the same for all particles: Reviewing the Particle constructor, it can be seen that the starting value is hardcoded to 0.0f.

```
Particle::Particle()
{
	// construtor
	this->life = 0.0f;
	this->position.set( 0.0f, 0.0f,  -6.0f );
	this->velocity.set( -1.0f, 0.0f,  0.0f );
	this->scale.set( 1.0f, 1.0f, 1.0f );
	this->rotation = 0.0f;
	this->rotation_velocity = 0.15f;
	this->next = nullptr;
	this->prev = nullptr;
}
```

The Particle's rotation value is only modified in the Particle::Update() function: Looking at the below screenshot of Visual Studio's "Find All References" result for rotation, it is only modified in the Particle::update() function which is called after the SpawnParticle loop in ParticleEmitter::update().



All Particle's rotation values are increased by the same amount within a given ParticleEmitter::update() cycle: First note that time_elapsed is defined outside the loop inside ParticleEmitter::update() and is not change inside the loop. Next note that the highlighted while loop will iterated through the entire particle list. Lastly note that Particle.Update() is called for every particl;e with the same time_elapsed value.

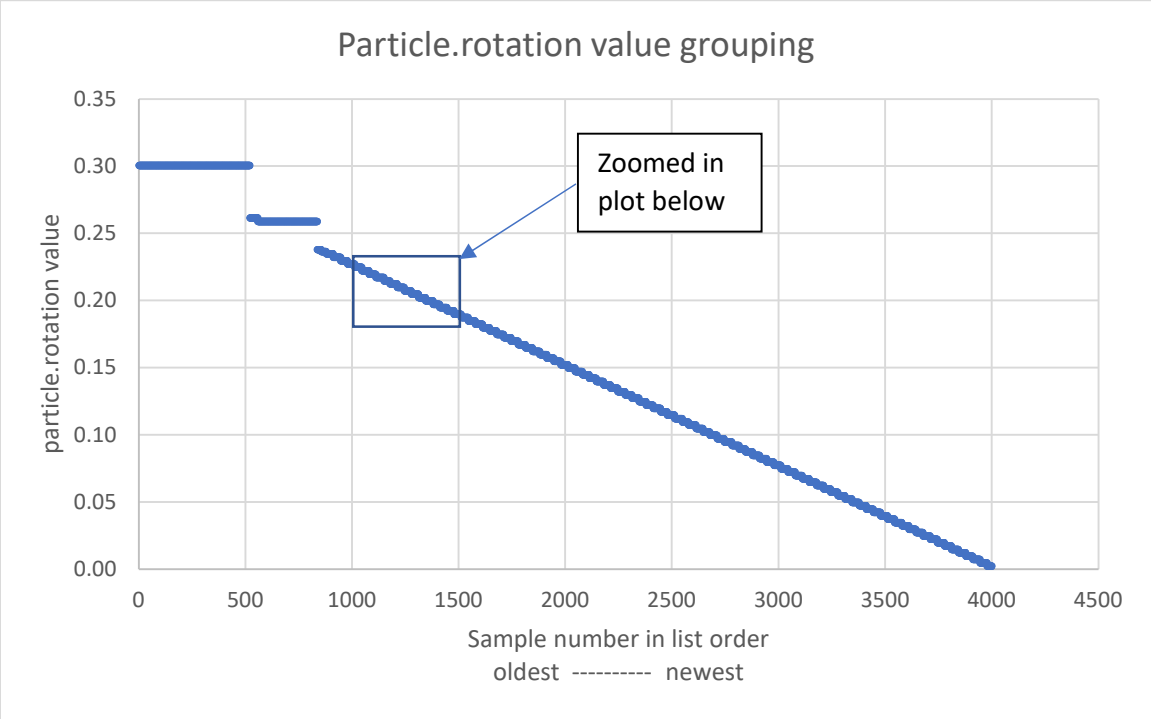
Moving to the Particle.Update() function, observe that after optimization the value that is added to rotation is (rotation_velocity * time_elapsed) where rotation_velocity 0.15f for all particles.

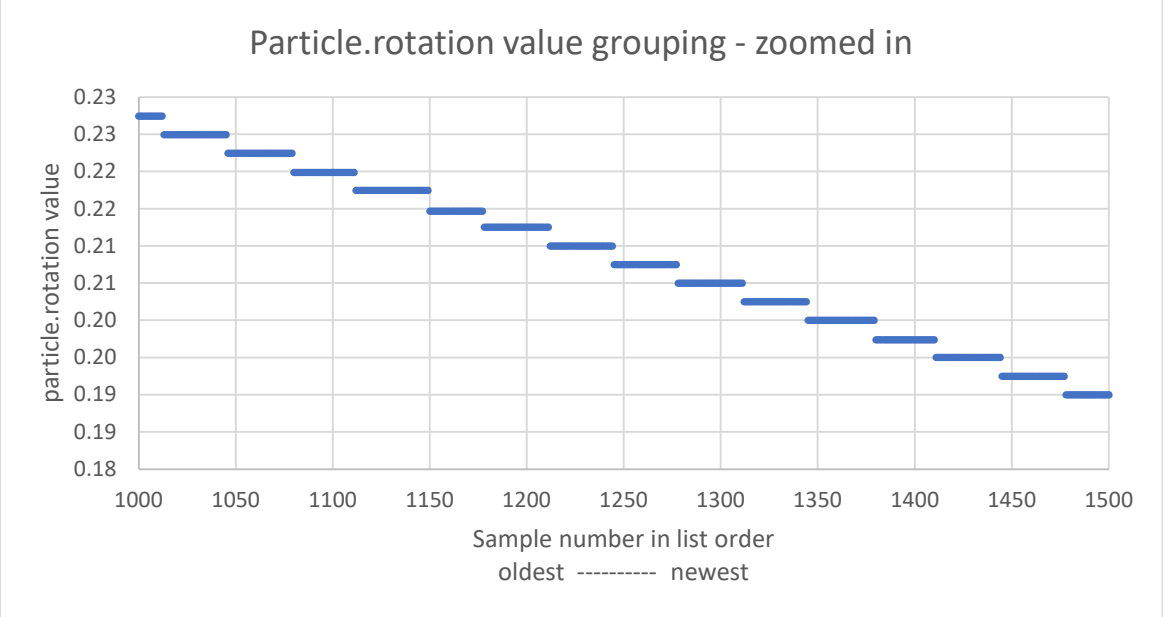
```
void ParticleEmitter::update()
{
	...
	// total elapsed
	time_elapsed = current_time - last_loop;

	Particle *p = this->headParticle;
	// walk the particles

	while( p != nullptr )
```

```
        {
                // call every particle and update its position
                p->Update(time_elapsed);

                ... [code to remove old particles]
        }

        ...
}

void Particle::Update(const double& time_elapsed)
{
        ...

        double MatrixScale = -1.3*tmp.Determinant();

        // serious math below - magic secret sauce
        life += time_elapsed;
        position = position + (velocity * time_elapsed);
        Vect4D z_axis(0.0f, 0.0f, -5.0f);
        Vect4D v(-12,1,0);
        position.Cross( z_axis, v);
        v.norm(v);
        position = position + v * 0.05 * life;

        if( MatrixScale > 1.0 )
        {
                MatrixScale = 1.0/MatrixScale;
        };

        rotation = rotation + MatrixScale 0 + rotation_velocity * time_elapsed ;
}
```

Putting it all together: Particles are created in groups and added to the list with the same starting rotation value. Then all particles on the list have their rotation values increased by the same amount. Then more particles are added to the list. Then all particles on the list have their rotation values increased by the same amount. This keeps repeating till there are 200,000 particles then no more are created but they all still get increased by the same about.

If you wait till the last particle is created then you run through the list and record the rotation values of every 50th particle (4,000 of 200,000), you will get a plot like below.





Now that we see that there are about ~100 groups (see histogram below), how can we use this to our advantage to save on having to run sin() and cos() 200,000 times and instead only run it ~100 time.

Here's a code snippet from an optimized version of the particle simulator that uses this information:

```
void ParticleEmitter::draw() const
{
	Matrix tmp(Matrix::IDENTITY_MATRIX);
	__m128 m128Tmp;

	// iterate throught the list of particles
	Particle* p = headParticle;
1	float tmpRot = p->rotation;

	float cosVar = cos(p->rotation);
2	float sinVar = sin(p->rotation);

3	float tmpRotLimit = tmpRot - FLOAT_EQUAL_TOLERANCE;


	do
	{
		assert(p != nullptr);

		// placing this at the beginning allows p==pParticleBlockEnd to go
		//	into the while where tailParticle==pParticleBlockEnd might be true
		if (p == pParticleBlockEnd)
		{
			p = pParticleBlockStart;
		}

		// if value is different enough, recalculate sin and cos
4		if(p->rotation < tmpRotLimit)
		// actual code is ((tmpRot - p->rotation) > FRACTIONAL_ROTATION_TOLERANCE)
		but rearranged for optimization
		{
			tmpRot = p->rotation;
5			tmpRotLimit = tmpRot - FLOAT_EQUAL_TOLERANCE;
			cosVar = cos(p->rotation);
			sinVar = sin(p->rotation);
		}
		...
	}
}
```

Explanation:

1. The first particle's rotation value is saved.
2. The first particle's sin() and cos() values are saved
3. A limit is defined. Since we are using floats, and we know the next groups' rotation value will be lower, this threshold is the current rotation value minus a very small value used to differentiate floats (I used `0.000001f`).
4. Every time we loop, we check if the current particle's rotation is beyond the calculated threshold.
5. If it is, we've hit our next group. We need to calculated and save all the relevant values.

Last data metric, below is a histogram showing all 99 groups and their sizes in the 4000 data point sample and how big they likely are if applied to all 200,000.

Data from plots			# of particles in group if proportionally applied to all 200,000
Group #	Group rotation value	# of particles in group	
1	0.3007770	520	26000
2	0.2617950	37	1850
3	0.2590540	279	13950
4	0.2381480	18	900
5	0.2367660	24	1200
6	0.2349580	33	1650
7	0.2324660	35	1750
8	0.2298680	32	1600
9	0.2274600	34	1700
10	0.2249480	33	1650
11	0.2224690	34	1700
12	0.2198900	32	1600
13	0.2174930	38	1900
14	0.2146670	28	1400
15	0.2125240	34	1700
16	0.2099800	33	1650
17	0.2074900	33	1650
18	0.2050120	34	1700
19	0.2025020	33	1650
20	0.2000250	35	1750
21	0.1973870	31	1550
22	0.1950300	34	1700
23	0.1925060	33	1650
24	0.1899940	34	1700
25	0.1875100	33	1650
26	0.1850230	33	1650
27	0.1825280	34	1700
28	0.1800130	33	1650
29	0.1775130	33	1650
30	0.1750160	33	1650
31	0.1725240	34	1700
32	0.1699670	34	1700

33	0.1674810	34	1700
34	0.1649220	33	1650
35	0.1624550	34	1700
36	0.1598870	33	1650
37	0.1573610	33	1650
38	0.1549510	33	1650
39	0.1524350	34	1700
40	0.1499070	33	1650
41	0.1474100	34	1700
42	0.1448600	32	1600
43	0.1424510	33	1650
44	0.1399590	34	1700
45	0.1374430	33	1650
46	0.1349450	34	1700
47	0.1324490	34	1700
48	0.1298950	32	1600
49	0.1274700	34	1700
50	0.1249450	33	1650
51	0.1224570	33	1650
52	0.1199340	35	1750
53	0.1173560	31	1550
54	0.1149740	35	1750
55	0.1124090	36	1800
56	0.1096510	29	1450
57	0.1074970	35	1750
58	0.1049120	33	1650
59	0.1024210	32	1600
60	0.0999653	34	1700
61	0.0974462	33	1650
62	0.0949583	36	1800
63	0.0923051	34	1700
64	0.0897608	30	1500
65	0.0874713	33	1650
66	0.0849950	33	1650
67	0.0825026	33	1650
68	0.0800190	34	1700
69	0.0775074	33	1650
70	0.0749992	33	1650
71	0.0725233	34	1700
72	0.0700113	33	1650
73	0.0675119	33	1650
74	0.0650110	34	1700
75	0.0625191	33	1650
76	0.0600191	33	1650

77	0.0575231	34	1700
78	0.0550189	33	1650
79	0.0525033	33	1650
80	0.0500066	34	1700
81	0.0475053	33	1650
82	0.0450134	33	1650
83	0.0425027	34	1700
84	0.0400099	33	1650
85	0.0375024	33	1650
86	0.0350076	34	1700
87	0.0325145	33	1650
88	0.0300246	33	1650
89	0.0275175	33	1650
90	0.0250217	34	1700
91	0.0225245	33	1650
92	0.0200223	34	1700
93	0.0175012	33	1650
94	0.0149882	33	1650
95	0.0125125	34	1700
96	0.0099976	33	1650
97	0.0074926	34	1700
98	0.0049703	33	1650
99	0.0024991	23	1150