Particle system report

Wolodymyr Krywonos

December 2020

1 The model - Snow globe

The particle system created was intended to be a snow globe. The particles are textured quad primitives and have collisions with a sphere, which is not rendered, using a GPU based approach in OpenGL. The interactive features of the system were (all controllable from within the application window) :

- Change the life time of the particle
- Change the dampening coefficient of the particle on collision
- Change the size of the Globe
- Change the gravity

Additionally the user is able to fly around the scene. The libraries used were:

- GLFW - to handle input, output and windows
- GLEW - windows only exposes OpenGL 1.1 entry points
- GLM - for mathematics

System specifications:

- Processor: AMD Ryzen 7 3700X 8-Core Processor (16 CPUs), 3.6GHz
- RAM: 16 GB 3000MHz
- NVIDIA GeForce RTX 2070 SUPER 8GB

2 Laws of motion

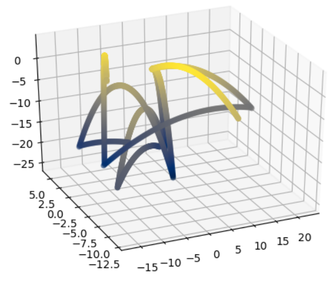
The laws of motion used to model the system were based in classic Newtonian physics for the motion of the particles with respect to gravity ($\odot$ will denote pairwise multiplication).

$$force = mass \odot acceleration$$

$$velocity_{t+\epsilon} = velocity_t + acceleration \times \epsilon$$

$$pos_{t+\epsilon} = pos_t \odot vec_{t+\epsilon}$$

Assuming the mass of the particle is one, the typical trajectory of a collision-less particle is parabolic; this is not unlike real life. The accuracy of this is linked to the magnitude of ϵ which is linked to the frame-rate, therefore the slower the program the less accurate the system is to real life; this effect can be seen in 1. The magnitude of gravity was 9.81, which is fairly standard.


(a) Particle sampled during simulation with one particle - high framerate

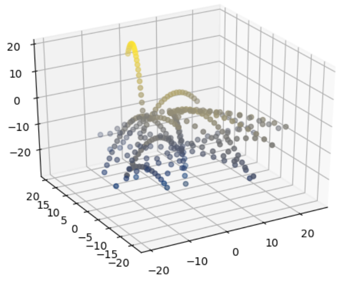

(b) Particle sampled during simulation with 500,000 particle - low framerate

Figure 1: Comparison of a single particle's sampled positions

For modelling the accuracy of colliding with a curved surface the new velocity of the particle was calculated under some assumptions: the size of the particle is infinitely small and the point of collision with the edge of the sphere forms a tangential plane. After which a basic reflection of the particle's velocity occurs around the normal vector of the plane. Since we are modelling a globe with center at the origin it is equal to the point of contact. Whilst the laws of motion are fairly sound in linear algebra, the system does not obey physical laws very well.

One issue that sometimes becomes present is particles will get stuck behind surfaces and become unable to escape. This is due to the new velocity, after dampening has been applied, being unable to cross the boundary of sphere's edge within a single time step. To counter act this when the particle's velocity adjusts the position is reset to the previous time-step. This appears more frequently the closer the incident angle gets to 0 and the greater the dampening effect becomes. The rough surface of the inside of the Globe a random variable was added to the equation, sampled from a multivariate standard normal distribution,although this does not preserve velocity.

$$velocity_{t+\epsilon} = (velocity_t + acceleration \cdot \epsilon) \odot |S|, S \sim \mathcal{N}$$

2.1 Efficiency analysis

The efficiency of the laws of motion were fairly well implemented however they were not the best for performances' sake. For example the basic law of motion, as described above, was implemented in a one to one fashion per time step. It could be argued, since the velocities are sampled from a distribution, that velocities could be stored in a precomputed lookup table between velocities at different time steps at the cost of precision, which would be largely unnoticed after rendering.

The efficiency of calculating results between the collision between the wall and particle was done using quaternions. Compared with the original approach of using the multiplication of rotation vectors combined with angles in each of the basis' vectors, this approach is quite efficient.

$$v' = Rot_x \cdot Rot_y \cdot Rot_z \cdot v$$

I used the equations:

$$q = \begin{pmatrix} pos_x \sqrt{\frac{1-cos(\theta)}{2}} (\sqrt{pos \cdot pos})^{-1} \\ pos_y \sqrt{\frac{1-cos(\theta)}{2}} (\sqrt{pos \cdot pos})^{-1} \\ pos_z \sqrt{\frac{1-cos(\theta)}{2}} (\sqrt{pos \cdot pos})^{-1} \\ \sqrt{\frac{1+cos(\theta)}{2}} \end{pmatrix}$$

$$v' = qvq^*$$

To calculate the rotation of the velocity vector around the normal vector of the plane, which is also the position of particle. These can be simplified down using algebra because $\theta = \frac{\pi}{2}$.

$$q = \sqrt{\frac{1}{2}} \begin{pmatrix} pos_x \frac{1}{||pos||_2^2} \\ pos_y \frac{1}{||pos||_2^2} \\ pos_z \frac{1}{||pos||_2^2} \\ 1 \end{pmatrix}$$

Since the distance of the point of collision is on the edge of a sphere it will remain constant, equal to the radius. Becoming:

$$q = \sqrt{\frac{1}{2}} \begin{pmatrix} pos_x \frac{1}{r} \\ pos_y \frac{1}{r} \\ pos_z \frac{1}{r} \\ 1 \end{pmatrix}$$

This requires a minimal number of vector multiplications. From what I researched there is no more efficient way of calculating vector rotation.

3 Performance analysis

I chose a GPU based rendering approach which achieves very good performance for smaller particle numbers (about 1 frame every millisecond). This most likely limited by the bandwidth between GPU and CPU. This latency can be seen between at the low particle numbers in the average frame computation of CPU and CPU + GPU.

No. particles	Average frametime of standard simulation (ms)	Average frametime of CPU only (ms)
1	0.0166127	0.00027697
10	0.016424	0.000366765
100	0.016555589	0.000125227
1000	0.0166647	0.000986268
10000	0.0166247	0.00931853
50000	0.0415074	0.0363246
100000	0.056786	0.0533391
250000	0.1066688	0.106483
500000	0.202394	0.192666
1000000	0.392215	0.36513

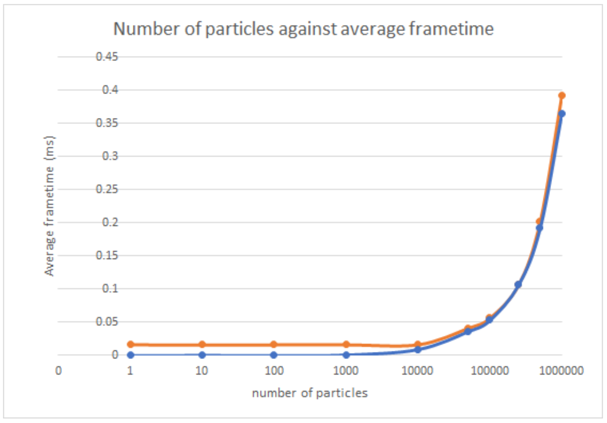


Figure 2: Frametime comparison of rendering using CPU (blue) against GPU (orange)

These results were all run over the period of 20 seconds over 10 runs under the same conditions. As you can see the rendering speed is bottle-necked by the CPU's performance, this is shown by the almost one to one increase in frametime between CPU and CPU + GPU. The GPU rendering never falls below 0.0166 ms on average which is the refresh rate of the monitor suggesting that this is limited by this.

In spite of all of this the frame time objectively remains quite low overall, due to relative efficiencies in the code and the rendering being performed on the GPU. The GPU based rendering was done with minimal overhead as both fragment and vertex shaders were relatively simple. To help the performance of the CPU various measures were taken in place, for example since the life times of each particle generated were fixed the particles were stored in a queue (manipulated using pointers) with the particles at the front of the queue being those with the shortest lifespan. This has the benefit of being able to skip sections of computation, such as stopping generating particles early.

To improve the CPU's performance I can think of two main improvements both involving parallelism. The first would be using multi-threading based approaches, most likely posix threads for C++, to fill the particle buffer before being sent to the GPU for rendering. The second would be to use general purpose GPU computations using a library such as CUDA. This has two benefits: using the GPU's parallelized architecture for increased performance (matching code to architecture) and also minimizing the latency between the CPU and GPU.