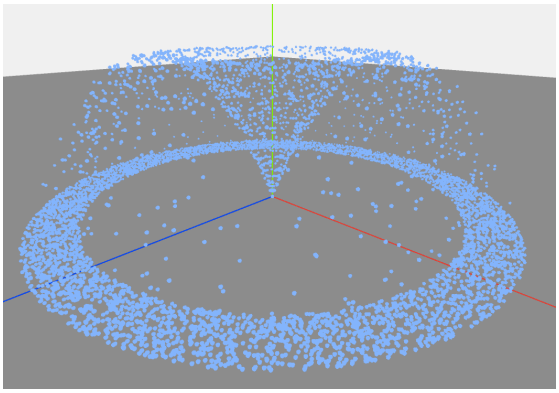
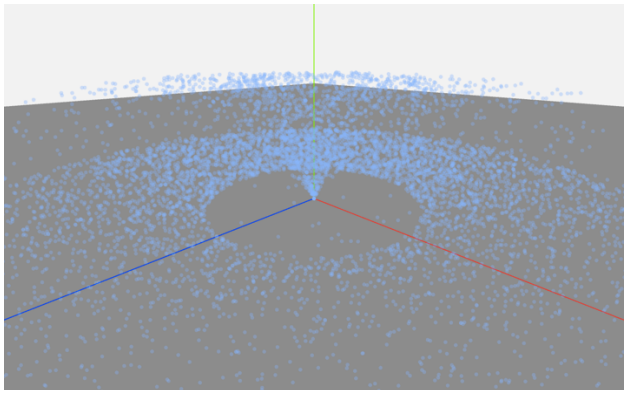
Particle systems

Yecheng Chu

Analysis of the fidelity of your chosen laws of motion – how well do they simulate the behaviour you have chosen to model? Fidelity of chosen laws [3 marks]. RUBRIC: 1 mark for basic description, i.e. can describe what the laws of motion are intended to be, and demonstrate that the particles roughly follow those; 1 mark for ability to describe informally (e.g. without captured data) how accurately the laws of motion are simulated; 1 mark for ability to describe with specific data/examples how accurately the laws of motion are simulated.





In my simulation, my particle system looks default like a fountain, but it could also be changed to look like an irrigation sprinkler by selecting spread mode which has a wider range of ejection angle. For the discussion below, I use the default mode, which is the fountain mode, to demonstrate the law of motion implemented. Although both modes using exactly the same principle, the fountain mode has a better visual. My particle system assumes that, there is no air resistance, so the only factor which influence the particle's velocity in Y direction is the gravity. And there is no external force in X and Y direction, so the velocities in X and Z remain constant.

Initially, the particles are ejected from the center with a fixed initial velocity in Y direction with an ejection angle which is randomly selected given a range. And its velocities in X and Z direction is calculated according to the V_Y and the ejection angle. As mentioned before, the velocities in X and Y direction are constant, so the corresponding displacement is very easy to calculate using the formula:

$$s = ut$$

$$where\ s\ is\ the\ displacement, u\ is\ the\ initial\ velocity\ and\ t\ is\ the\ time\ change$$

The displacement in Y direction is an accelerated motion, the displacement is calculated using the formula:

$$s = ut + \frac{1}{2}at^2$$

$$where\ a\ is\ the\ acceleration, in\ this\ case\ is\ acceleration\ of\ free\ fall\ (g = 9.8ms^{-2})$$

Since the acceleration is in opposite direction to the initial velocity in Y direction, I change the plus sign to minus sign:

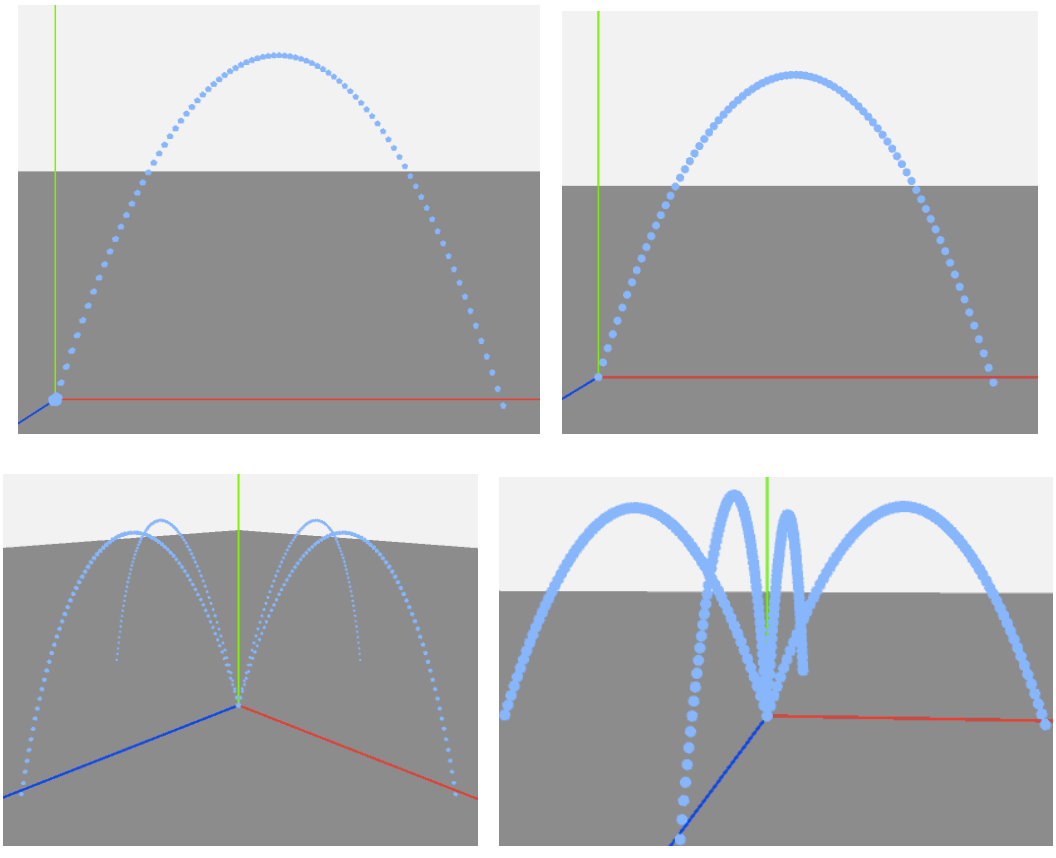
$$s = ut - \frac{1}{2}gt^2$$

I also update the velocity in Y direction using the formula:

$$v = u - gt$$
$$from\ the\ formula\ \ v = u + at$$
$$where\ v\ is\ the\ new\ velocity\ and\ u\ is\ the\ initial\ velocity$$

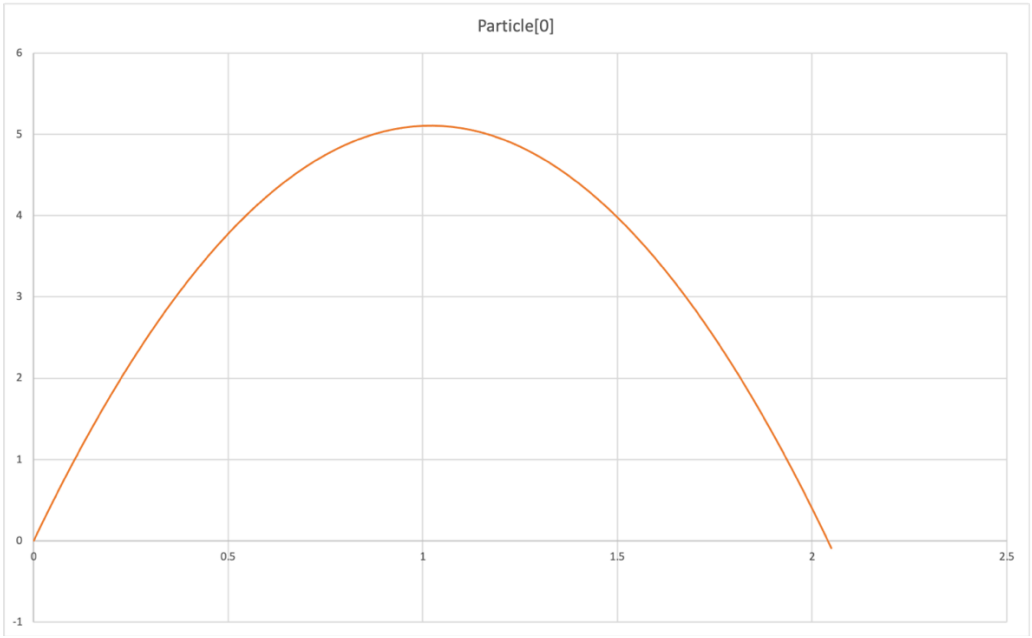
I think my particles follow the rule I implemented accurately, since the particles first go up due to the initial ejection velocity in Y direction, then it reaches the maximum point when its velocity in Y direction reaches 0, after that the particle start to drop down to the ground.

I change my particle system to make it only eject particles from one direction or four directions with a constant ejection angle, it is clearly shown that the particles follow a parabolic shape after being ejected.

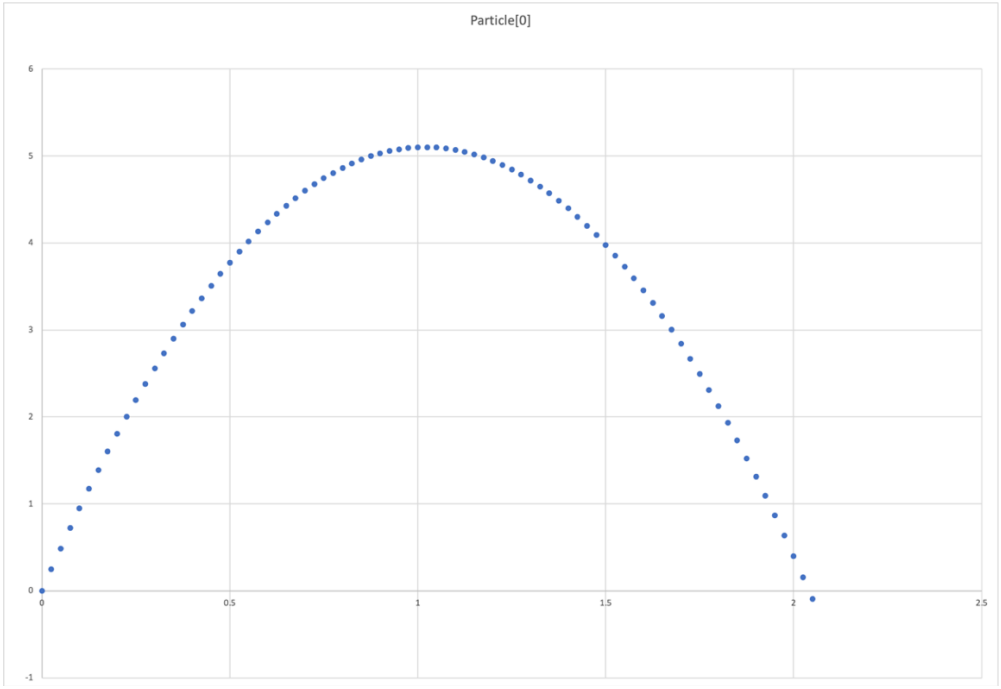


I plot the position of one particle which is particle[0] over the time to see its position change. It is clearly the particle first goes up then falls down. It has its Y position 0 at time 0, then goes up to 2.19375 and 3.775001 at time 0.25 and 0.5. It finally reaches the maximum point at time

1.025 with Y position 5.10194. After that, the particle falls, its Y position drops to 3.002443 and 0.400008 at time 1.675 and 2. It finally drops below 0 at -0.092242 when time is 2.05. At this point, the particle is placed on the ground without further Y position change.

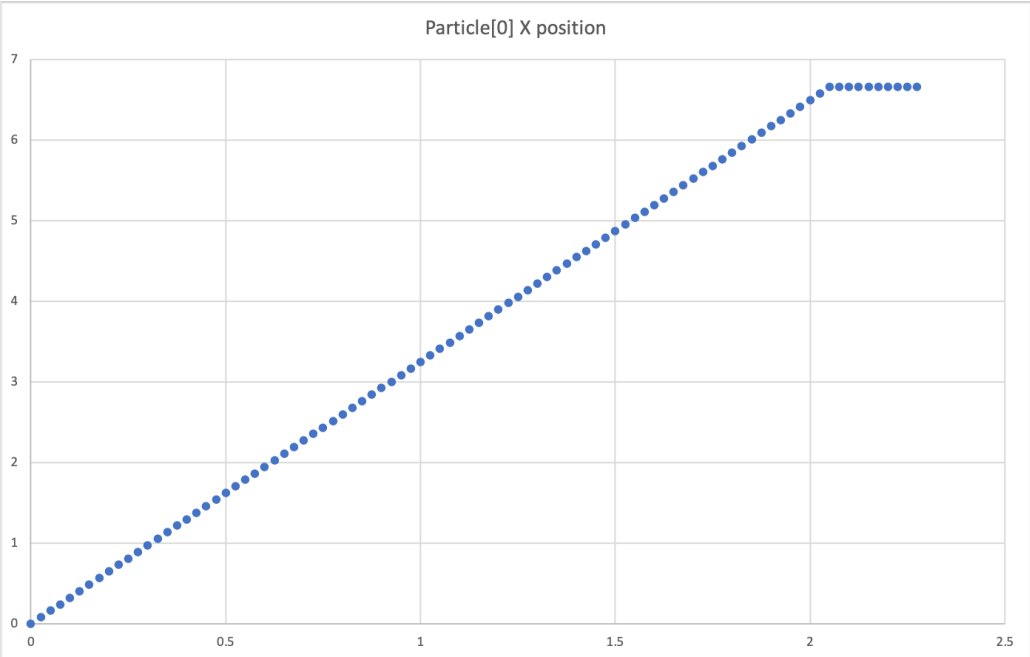


Particle[0] Y position changes over time



As I do not change the X and Z position, its X and Z position increases constantly before reaching the ground.

I just plot the X position of particle[0] to illustrate the change, the position in Z is exactly the same trend. After the particle is ejected, the X position is increase constantly until the particle reaches ground at time 2.05, the X position remains constantly at 6.656845.



Efficiency of your approach to implementing your laws of motion – how efficient is your computation of particle position between frames? RUBRIC: 1 mark for any evidence that efficiency has been thought of; 1 mark if there is evidence of thinking about efficiency and what measures could be taken to improve it, or a solid argument as to why there is no possible way of improving it; 1 mark for demonstrating that specific measures to improve or maximise efficiency have been implemented.

I did some changes to make my law of motion more efficient. First, when applying the law of motions, I ignore the forces between the particles. These forces are calculated using Newton's law of gravitation which has the formula:

$$F = -\frac{Gm_1m_2}{r^2}$$

$$where\ G\ is\ the\ gravitational\ constant\ and\ m_1\ and\ m_2\ are\ masses\ of\ the\ two\ particles$$
$$r\ is\ the\ distance\ between\ the\ two\ particles$$

Since the masses of the two particles are very small and the mass of the earth is $6.0 \times 10^{24} kg$, so the forces between the two particles could be ignored compared to the force exert by the earth to the particles. As a result, during my calculation of the particle positions, I only take into account the gravitational force, and can save huge amount time. If I need to calculate forces between particles, I need to first record the mass of each particle and then for each particle, I need to consider the particles around it, that is a hugely computationally cost.

I use loop to go through every particle that has a Y position greater than 0 to calculate its position in X, Y and Z. Meanwhile I do an update of the velocity in the Y direction as it is affected by the gravity. Initially my code looks like this:

```
GLfloat displace_x, displace_y, displace_z;
// s -> displacement
// u -> initial velocity
// v -> terminal velocity
// t -> change in time

// s = ut + (1/2)at^2
displace_y = particles[i].velocity[1] * time_delta - (gravity * pow(time_delta,2))/2;
particles[i].position[1] += displace_y;

// v = u + at
particles[i].velocity[1] += (time_delta*(-gravity));

// s = ut
displace_x = particles[i].velocity[0] * time_delta;
particles[i].position[0] += displace_x;

// s = ut
displace_z = particles[i].velocity[2] * time_delta;
particles[i].position[2] += displace_z;

//add time count to determine when the particle will be dead
particles[i].time += time_delta;
```

I found the $garvity * pow(time\_delta, 2))/2$ part could be a constant for every particle when the gravity is constant. So I could just calculate it once and only update its value when the gravity changes. Also, in my initial design, the particle[i].time is only used for lifetime calculation. For the position in X and Z direction, particle[i].time could be directly used to calculate the position. As a result, I modified my code to this:

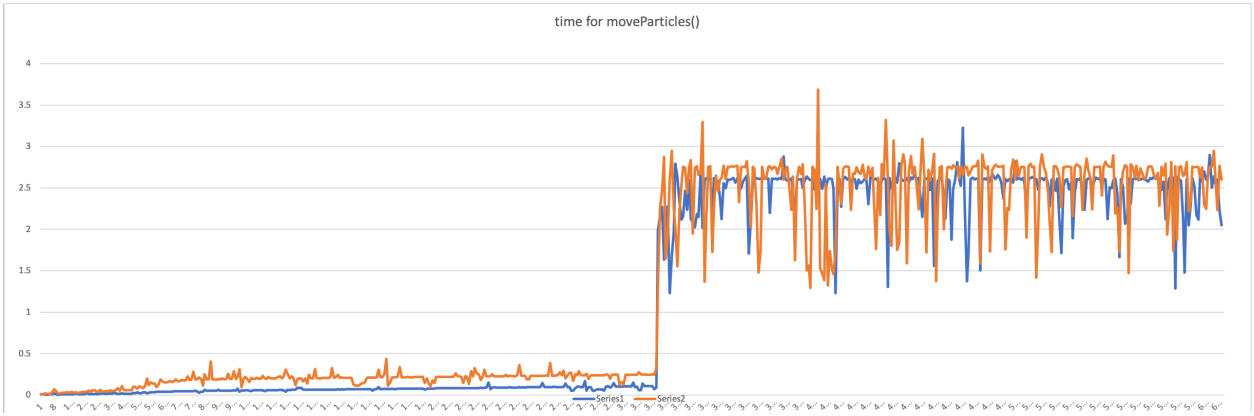
```
GLfloat displace_y;
// s -> displacement
// u -> initial velocity
// v -> terminal velocity
// t -> change in time

// s = ut + (1/2)at^2
displace_y = particles[i].velocity[1] * time_delta - constant_value;
particles[i].position[1] += displace_y;

// v = u + at
particles[i].velocity[1] += (time_delta*(-gravity));

//add time count to determine when the particle will be dead
float new_time = particles[i].time + time_delta;
// s = ut
particles[i].position[0] = particles[i].velocity[0] * new_time;
particles[i].position[2] = particles[i].velocity[2] * new_time;
particles[i].time = new_time;
```

I analysed the moveParticles() function where those calculation is performed using those two methods.



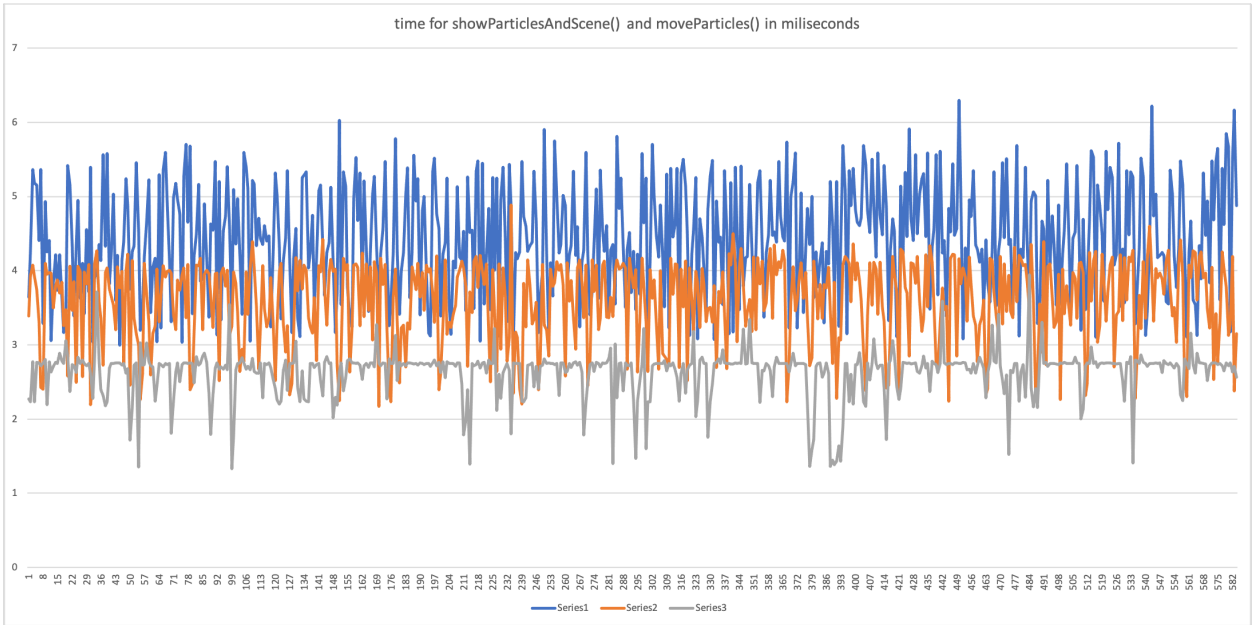
It shows that, after the modification(the blue curve), the running time(on y-axis) for moveParticles() improves slightly and is more stable compared to the old one. However, since there are also other codes in the moveParticles() and the particles generated has a random range and ejection angle selection, the experiment could just give a straightforward comparison for the change and not guaranteed to reflect the real improve in efficiency.

Analysis of overall performance / rendering speed, and discussion of efficiencies implemented. Here we'll be expecting you to be able to comment on the performance of your system; is the performance limited by the CPU? the GPU? How do you know that? Are there any particular bottlenecks in your design, and if so, how could you (hypothetically) improve the performance? This is as much about understanding as it is about implementation so if your system is performing poorly compared to your peers', but you know why that's the case and what could be done about it, then you will be able to get some of the marks for this task. For full marks we would expect evidence of experimentation such as graphs of particle numbers versus framerates [4 marks]. RUBRIC: 1 mark for sensible discussion of how performance is bound by the limitations of data structures/CPU/use of GPU/transfer of data between CPU-GPU; 1 mark for evidence of exploring performance/rendering bounds by performing experiments; 1 mark for analysis/discussion supported by some data; 1 mark for a rigorous performance analysis supported by graphs or other data visualisations.

I use the time count for performing the main functions(update particles and show particles and scene) for the particle system and the FPS count as a general measure for overall performance evaluation. The FPS is around 60 when the particles are rendered in point however, if it is changed to sphere, the FPS becomes very unstable and most of the time it is in the range of 40 to 59.

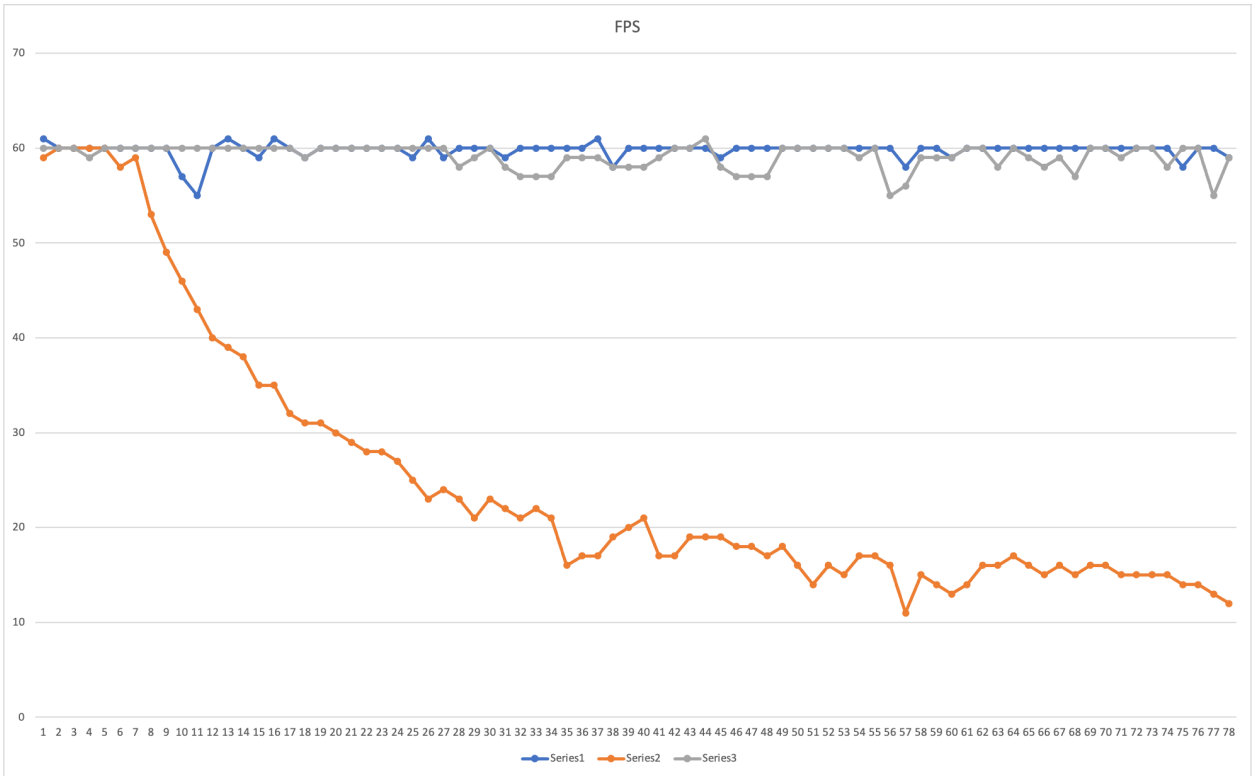
In my initial design, each particle structure has a Boolean to indicate whether it is alive. I use an array to store the particles created and loop around the array to do the update and show functions. When a particle is not alive, I do not update its position and velocity as well as not draw it. This has a drawback that since my dead particles are not really removed in my array instead, they are labelled to be not alive, when the particle generated started to grow, the time for doing the main functions of my particle system increases since the array size will always

grow until it reaches the maximum particles set and will be reset. I did something to improve its efficiency by add a count_begin to indicate the starting count position for looping the particles array. So when doing update and draw particles it do not need to start from index 0 as when the time increases the old particles are already dead so it is pointless to loop those index. I made a further modification. Which is remove the Boolean to indicate the particles is alive or not, but instead, in the update function, when I found a particle is dead, I delete it in the array. However, it also has a drawback that, deleting the element in array is time consuming so when the particles created are not that much, the previous versions have better performance. However, when particles created started to grow the advantage of my new version is shown. Below I record the time(on y-axis) to perform the main functions of my particle systems after it has run for three minutes.



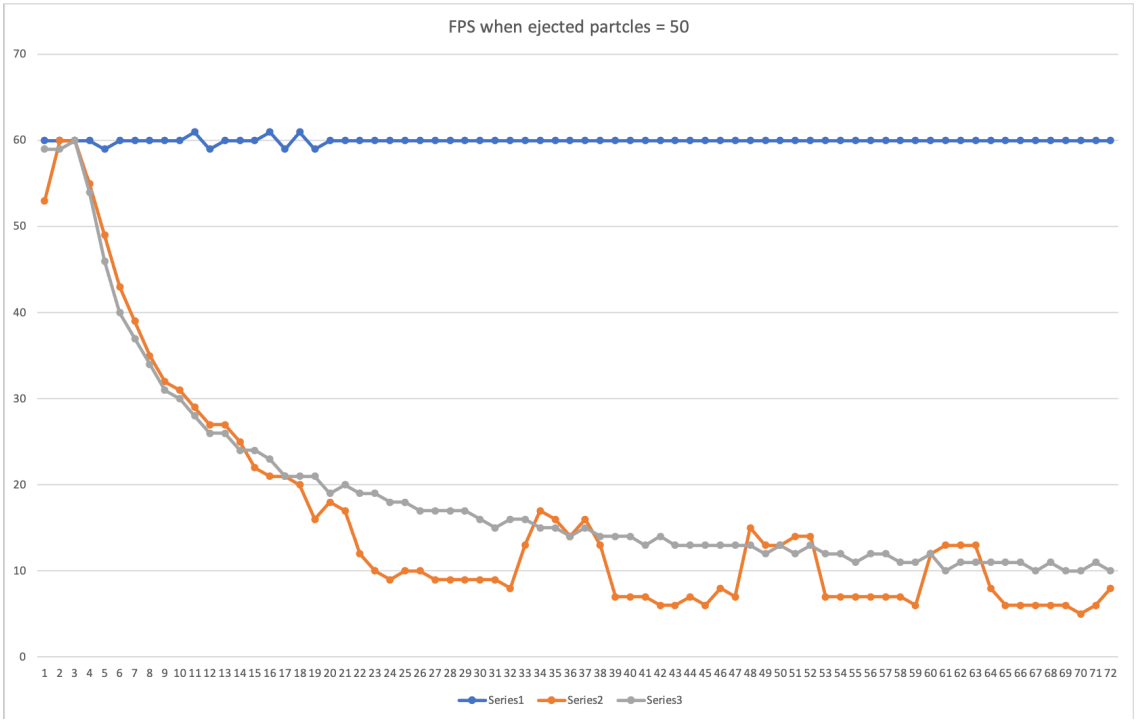
It is clearly shown that my new version(shown in grey) performs better than my original versions(shown in blue and orange respectively). I could further improve my particle system using a better function to remove the element from the particles array.

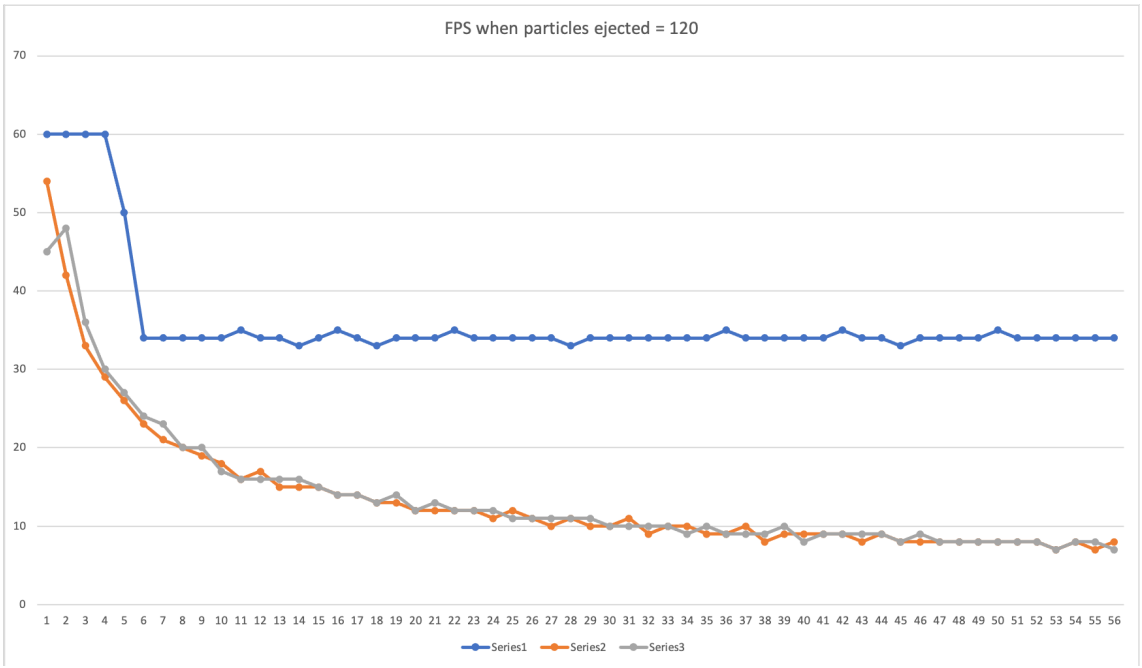
I also did some more experiment to exploring the CPU/GPU limit. For this part, I used sphere rendering since during previous experiment, I found the FPS for sphere is much lower than that of points. I did some experiment can collected 78 samples each to do the comparison.

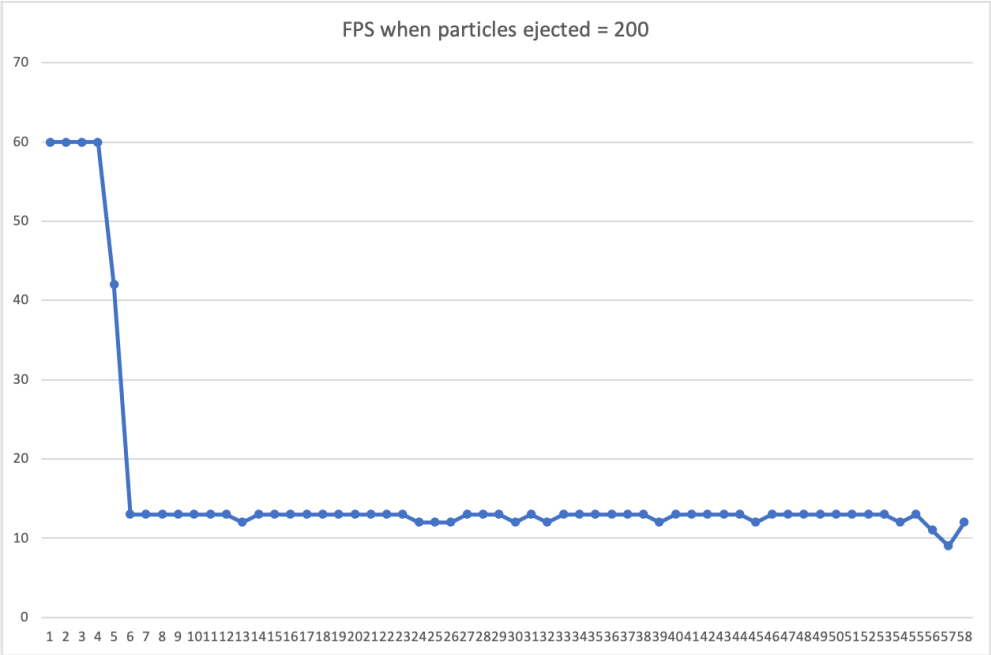


First I disable the part for drawing particles and scene, the FPS count shows it is very stable at 60 every time(blue line). Then I disable the part for updating the particles(orange line), the FPS falls continuously from 60 to 12. However, it cannot shown that all my performance limit is due to GPU as when I disable the update function, I disabled the code to do the delete of particles, it could lead to a poor performance, so I did a third trial, in this time I enabled some code in update particles to remove dead particles(grey line). It shows that the FPS is comparatively less stable compared to disable GPU part, so GPU has effect on the limitation of the performance of my particle system.

However, the FPS count doesn't change a lot. That could due to the point ejected by my particle system each time is fixed at the value of 25 and doesn't really reached the performance bottleneck, so I decide to repeat the above three trials with an ejection of points manually set at 50, 120 and 200 to do a further test.


FPS when ejected partcles = 50
70
60
50
40
30
20
10
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
Series1
Series2
Series3


FPS when particles ejected = 120
70
60
50
40
30
20
10
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
Series1
Series2
Series3



In the ejection particles = 200 case I only tried with the CPU only case since the FPS is too low for the other two cases.

In conclusion, from the experiment I did, it is very clearly shown that the performance is limited mainly by the GPU. The CPU also has some effect on performance but not too much. (NOTE: The above conclusion is made using the MacBook Air early-2015 to run the code.)

How to compile and run? (command to compile and run the code in Mac terminal)
$ gcc -DMACOSX -framework OpenGL -framework GLUT -o particle_system particle_system.c
$ ./particle_system