Volcano Particle System

Horia Radu

Summary:

I have used https://processing.org/ software, learning from their tutorials and forums how to use it, and writing my system in java.

My particle system is about a lava mountain on a lonely island in the ocean. The volcano for some reason is always smoking and from time to time it decides to erupt. Sometimes the lava goes off the island and lands in the ocean making steam, but not every time, most of the time it slides off the mountain and cools down on the island. It turns into obsidian and then it is cleaned up by the inhabitants of the island. So, life is not that great on the island…

1. **Particles and Movement**

I have two particle systems that move in 3D space that create numerous particles. One is for lava and one is for smoke and water vapours. Both of them have a customizable multiplier for how many particles are spawned.

2. **Rendering**

My particles have 2 ways of rendering, changeable by a button. They can either be rendered as spheres (multiple triangles) or a blank square with a round particle texture on top of it (billboarding approach).

3. **Control of the system**

I have made interactive controls to both particle systems regarding their initial velocity, lifespan of each particle and spawn multiplier to lower or increase the number of particles spawned. I also put 2 controls for gravity, one to turn it off entirely and one to change the intensity of the gravity. Furthermore, I am able to change the type of rendering mid animation (spheres and quads) and also change the colours of the particles to random.

4. **Fidelity of Laws of motion**

Each particle receives an initial velocity when it is made and an acceleration, calculated based on the gravity. Each time it is rendered the acceleration is applied to the velocity, then the acceleration is set to 0 so that next time the acceleration can be changed based on what is going in the environment.

Those particles also have a terminal velocity set which they can not go beyond. This is because, whenever the opposing forces acting on the object equalize each other (drag force = force of gravity acting on the object) it should not increase its velocity.

To make the system look and feel more realistic, while not going into depth into their weights, I have tried to simulate the sporadicity of the particles by giving them a random velocity within a certain circle radius, on the X and Z axis. For the Y axis I also randomized the speed that smoke and lava particles have when coming out of a volcano, with some particles going faster and higher, and some not. Basically, I am faking a form of gravitational attraction on the lava particles, and I use the same faking to imitate the hot smoke particles going up due to the molecular

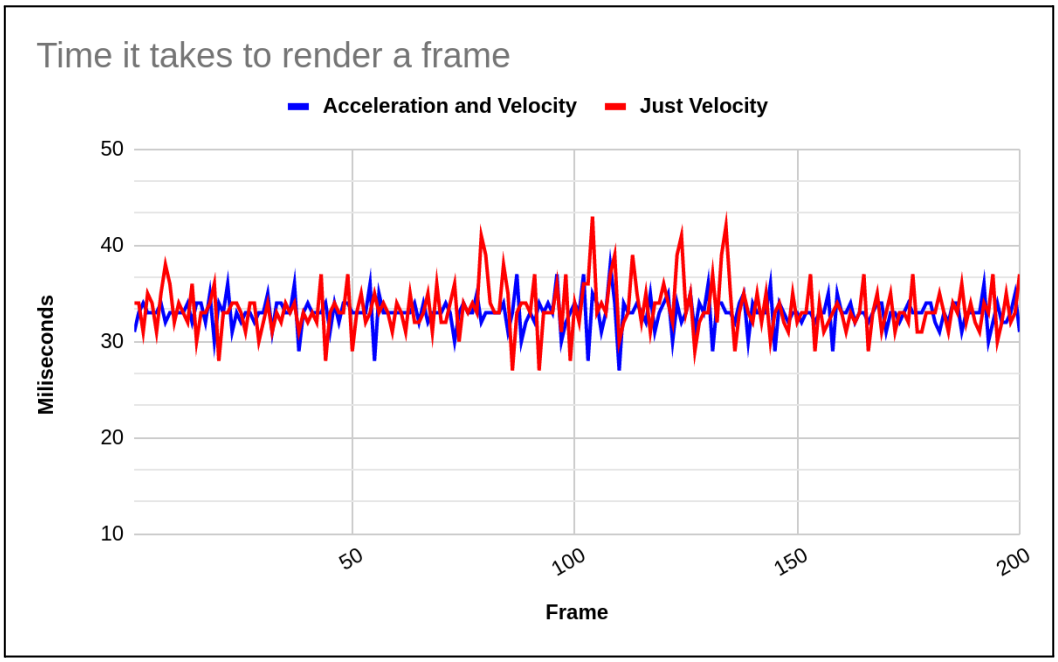
difference with the cold air, by giving them a negative gravity and a random velocity.

5. **Efficiency of approach**

The velocity of each particle is a point vector on the 3D axis, which determines how fast an object will move in what direction. Then the acceleration is calculated for the particle system and then added to the velocity for the rendering frame. Each particle also has another point vector to keep track of its position in the 3D space.

One way this calculation could have been possibly replaced, while reducing the need of calculations between acceleration and velocity, is if we set a static initial velocity that gradually increases or decreases directly, mimicking the acceleration without applying it at all. This scenario though would also provide limitations, as we would not be able to change the velocity based on random happenings in the environment. I believe this would be good for a preset background animation, maybe further away from the point of view, like the smoke of a fire in the distance, unaffected by wind, or a water fountain or river, which has a random but constant motion.Another method would be creating the path for the particle right when the particle was created, and it would just follow it, but which would have limitations similar to the method stated above.

I personally have not noticed a significant difference from doing this, and I wanted to keep the changeable gravity, so I stuck with the first option, but for larger scale particle systems, this might prove useful. Below is a graph between the 2 methods, one the milliseconds it takes to load a frame, over 86 frames. In both scenarios I had 10 000 particles of smoke rendering.



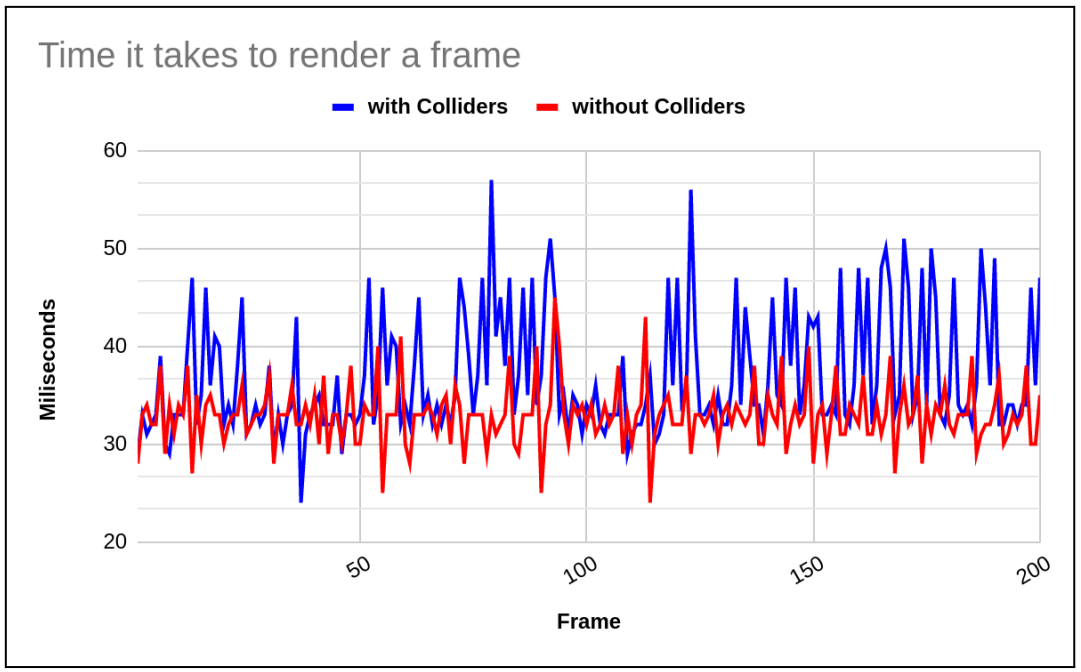
6. **Analysis of overall performance**

With the program that I used for this project I only had access to the CPU. I believe that all the rendering could have been exponentially increased by accessing the GPU because of the possible parallelisation of the graphics processing on the multiple cores of the GPU, this being

what the GPU intended use is. Similar to the CPU vs GPU in Machine Learning, the GPU can do more calculations in parallel, which would help with rendering my thousands of particles, compared to the CPU which focuses more on serial processing.

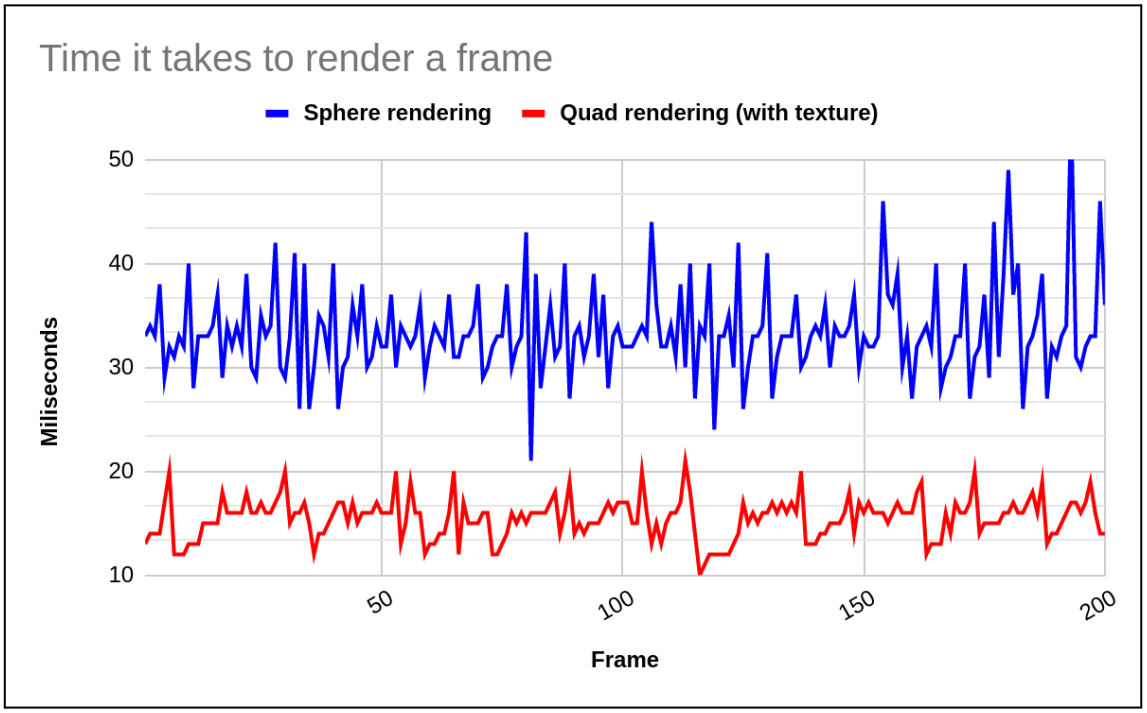
The textures for the static objects that I load do not play an exponential part in the performance as they are loaded only once in the setup and then kept that way. With that being said, those static objects, volcano, island and sea, have a collision system in place for the particle system. This has a considerable impact on the performance as for every render I have to check if any of the particles in the system have touched a collider surface. This means that I have to iterate through all the particles and then through all the colliders to verify this. To keep the performance as high as I could, I decided to only iterate through the lava particles for the collision, leaving the smoke particles without a collision. I figured that in my system, smoke would go up and disappear at some point, having no contact with any surface. Indeed, this only works in my current scenario as I have no wind which can send the smoke in some random direction, and neither do I have collider objects above the volcano which can stand in the way of the smoke.

Some collision algorithms are faster than others, and their usability depends a lot on the scenario you want to use it in, for example in a game where you want to restrict a player to not move outside of certain bounds, you could only allow him to move in a predefined space or calculate every position that a player or a particle could be standing at and what type of collision could occur at the specific points. It all depends on the situation, but I chose my solution as it was easier to implement and would work for all my random particles, in maintaining realism. Graph below is the comparison between systems with and without colliders, both having 10 000 particles



The most significant rendering performance “cheat” I did was on how the particles were formed. Initially I implemented them as spheres, made up of multiple triangles, which looked very good but performance took a big hit, because for every particle it would have to render at least 20

triangles. The triangle number varies, and depends on how small or far the particle is, but still, the impact on speed of rendering was clear. So, to solve this problem, I changed the particles into transparent quads which have a texture on them, faking a particle, basically a 2D billboard representation of a 3D particle. This proved to have a significant improvement on the rendering speed of each frame. Below is a graph for the difference, both scenarios have 10 000 particles



Overall, with the spheres and 10 000 particles I had 28-30 framerate, while with the quads my framerate was kept at 60 (which is where I capped the framerate at), so from this and the graph the difference can be clearly seen,

I have left both implementations in my code, changeable by a button. I was personally not able to make the quads continuously face the camera because of the free cam that I use from the processing libraries. This breaks the illusion of them being round particles when looking from the side, but with all that said, it can be done and it certainly helps performance. Furthermore, this method loses a lot of its realism when the particles are too big or too close to the camera, so a function to switch between those rendering types, based on the situation, seems most appropriate.

7. **Sophistication and flair**

For sophistication and flair I did multiple particle systems (lava and smoke), a free look camera, objects/planes with textures, textured particles and collision between lava particles and objects.