**Fluid Simulator with Particle**

**20141282 Bae JeongUn**

**1. Abstract**

This project is an experiment on a program that simulates fluids with particle. Since fluids are eventually made up of molecules, I started wondering if I could express and simulate fluids using spherical particles instead of molecules. It can be said that sand flows like a fluid, so if I apply the laws of physics to a particle like sand, the particles can flow like a fluid. And if I render to look like fluid based on the position of this particle, particles that look like fluid will flow. With this in mind, I started this project.

**2. Introduction**

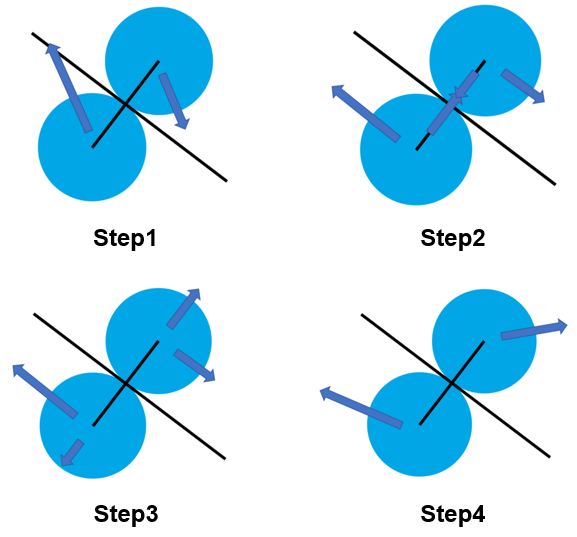
I thought it was important to solve both fluid graphics rendering and fluid motion to create a fluid simulator. While contemplating how to render a fluid, I thought that water is made up of several water molecules. In the human eye, the fluid is a lump, but in reality, it can be said to be like a bunch of very small particles. It's like a bunch of sand grains. So, I thought that if these grains were expressed well enough, they would look like fluid. In addition, I thought that using this granular model could represent the motion of the fluid. For example, people say that sand flows like water in a sandy desert. I thought it would be possible to express both fluid rendering and motion by using particle model.

**3. Method**

**1) Physics**

As explained earlier, in order to express the motion of a fluid, an appropriate physical law between particles is required. In grains of sand, in fact, only the laws of action and reaction exist. Of course, it is not accurate in detail, but for simplicity, we decided to apply only the laws of action and reaction to the particles. When particles collide, they are calculated to exchange their momentum. If particles collide with elasticity without energy loss, they collide endlessly. So, when the particles collide, the kinetic energy is set to gradually decrease.

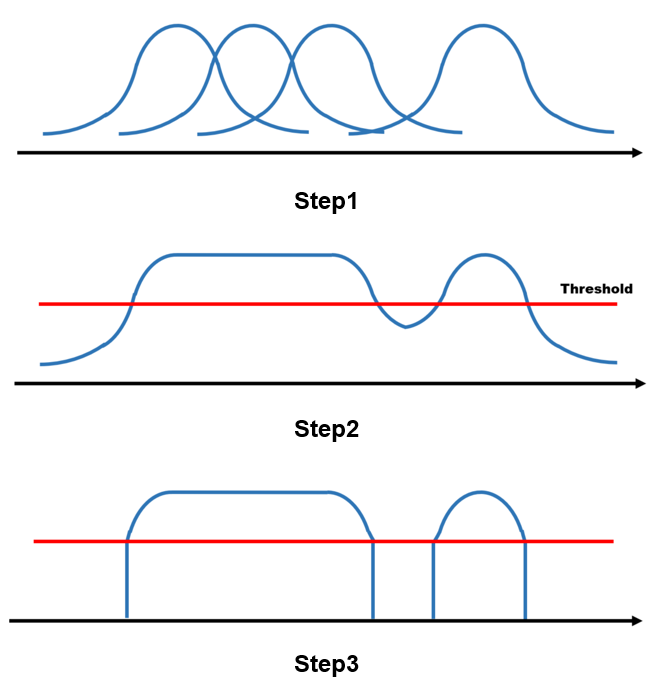
The formula for the action and reaction for both particles is classical mechanics. Using the positions of the two particles, obtain the normal vector of the collision surface. Using this normal vector, the acceleration and velocity vectors of the two particles are decomposed into a vector that is parallel to the normal vector and a vector that is not. Finally, it is done by exchanging the parallel acceleration and velocity vectors of the two particles. The above explanation is shown in the figure below.



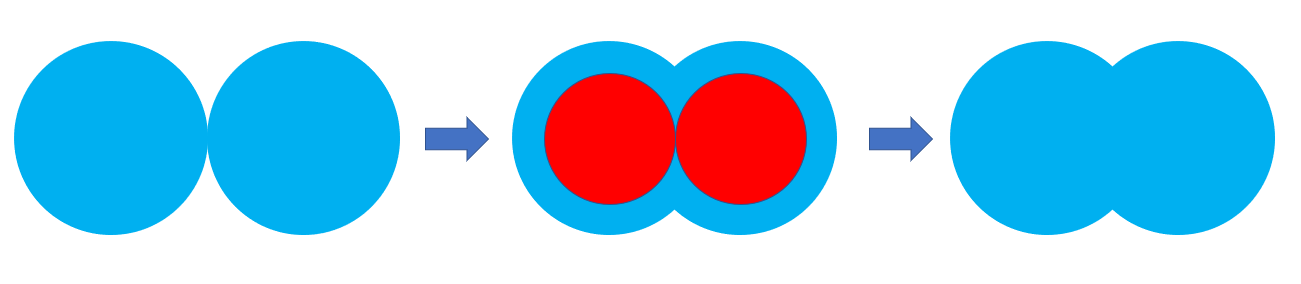
**2) Particle Rendering**

Real molecules are made up of several atoms. For example, water consists of 2 hydrogens and 1 oxygen, a total of 3. It takes a lot of computation to build a simulator with a molecular model with multiple atoms. In addition, when comparing the size of the actual molecule and the distance between molecules, I thought that the distance between molecules was larger than the size of molecules, so it could be sufficiently neglected. So, instead of the molecular model in the simulator, we decided to use a simple spherical model. In the case of using a spherical model, the amount of calculation is reduced to obtain the distance between two particles. If distance measurement becomes less computational, it becomes easier to calculate collisions between particles. This is because, based on the diameter of the spherical model particle, if it is large, it does not collide, and if it is small, it means it collides.

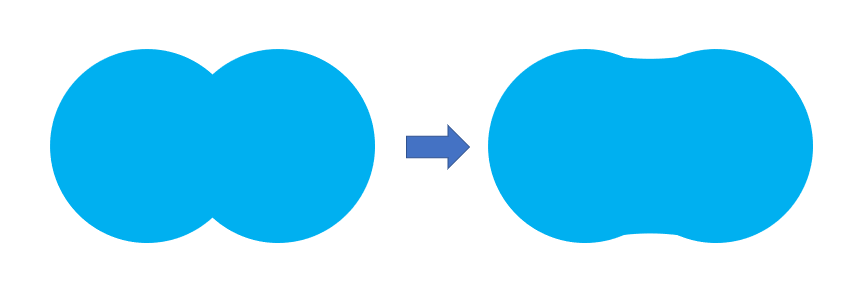
If the particle is rendered as a sphere, it will not look like water at all. The particles will look like beads. So, I decided to use volume rendering instead of face rendering. For volume rendering, the imaginary standard distribution of the particle center is calculated. After adding these values ​​of several particles, a threshold is applied. If the threshold is exceeded, the color is expressed, and if it is not exceeded, the expression is not applied.



If the visible size and the actual size of the particle are the same, the volume value between the two particles will be difficult to exceed the threshold. So, the size of the actual collision determination was made a little smaller than the size of the particle.

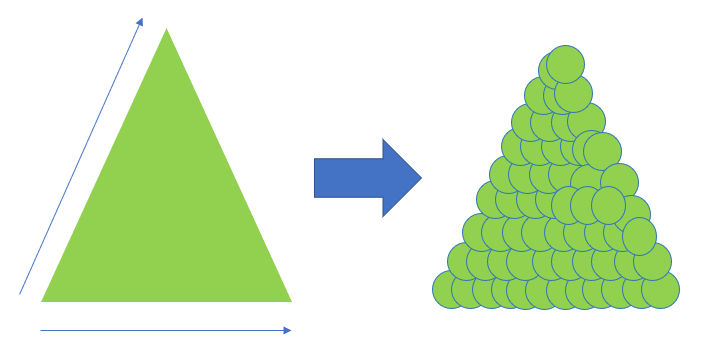


If the above two techniques are applied at once, it is expected to be expressed like a fluid as shown in the figure below. If it looks like the picture, the two particles will look like two water droplets.

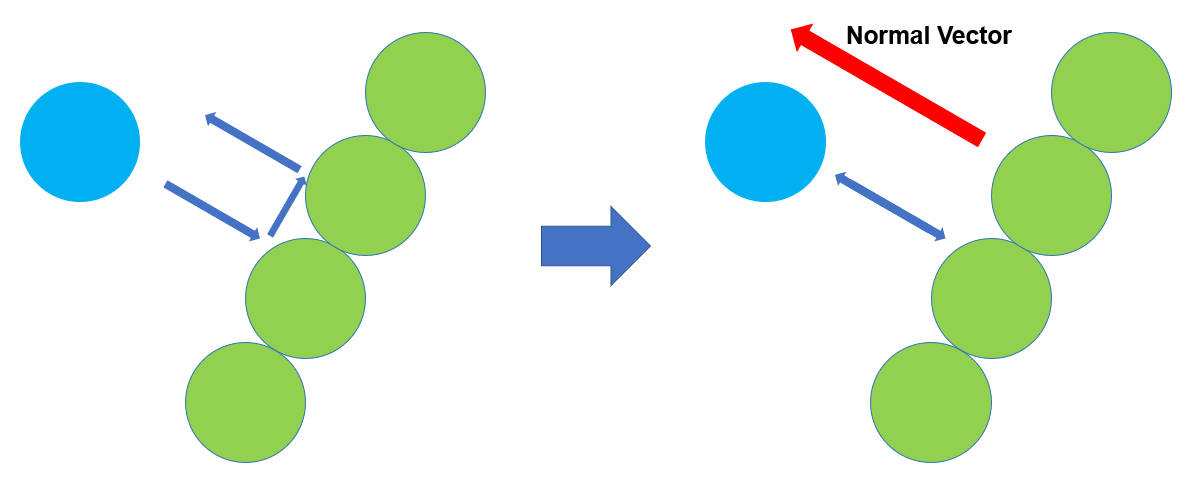


**3) Terrain Rendering**

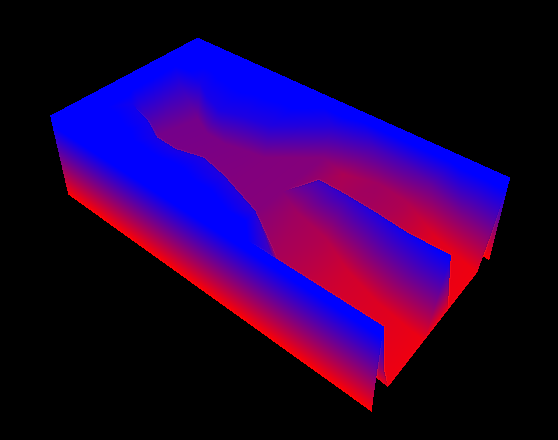
The terrain should also be a spherical particle model so it is easy to calculate interactions with fluid particles. However, in order to create a terrain, rendering it in the form of an existing polygon is easier to create a new terrain. Because it is possible to create one surface with only the coordinates of three points, in order to create a surface in the form of particles, coordinates for a number of particles in the surface are required. Rather than taking the location coordinates of each of the terrain particles, it is necessary to fill the surface with three polygonal triangle points. So, using three points, particles are created at regular intervals in the plane at regular locations.

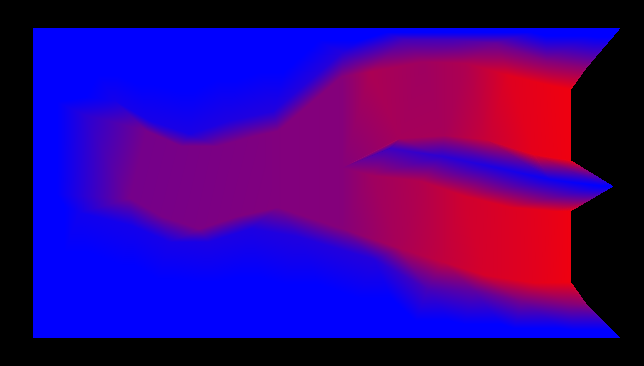


If a terrain particle is calculated like a normal fluid particle, and the reaction is calculated, the fluid particle colliding with the terrain particle may have some error. The reason is that the collision model is spherical, so fluid particles are not calculated in the direction of the normal vector of the terrain. Specifically, it is shown in the figure below. So, a normal vector is stored in the terrain particle so that when the fluid particle collides, the vector can be used to calculate the next motion.



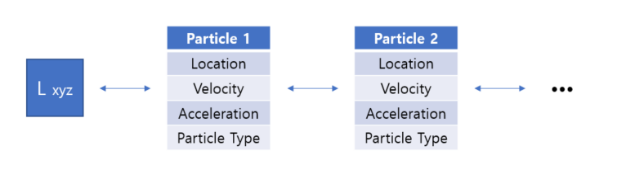
The simulated terrain is a terrain where fluid flows from a meandering large valley and is divided into two valleys in the middle. The two forked valleys have different shapes. In this terrain, fluid continued to create over a large valley like a waterfall. The figure below shows the rendered terrain. In the figure, blue represents high terrain and red represents low terrain.

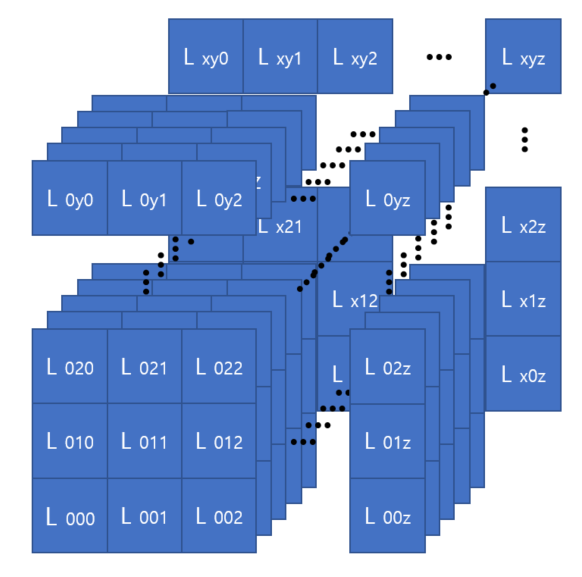




**4) Data Structure**

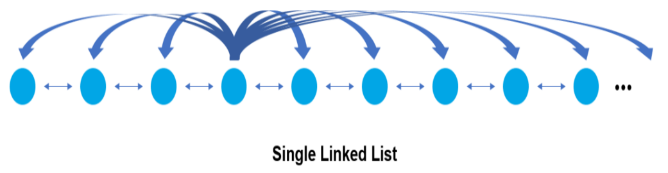
First, we need a data structure to contain the information of each particle. In order to express the movement of an object, three things are needed: position, velocity and acceleration. And you need to indicate whether the particle is a fluid particle or a terrain particle. As the number of particles increases, the number of collision tests between particles increases nonlinearly. So, a more appropriate data structure was needed. If the two particles are far enough apart, there is no possibility of colliding with each other. Therefore, there is no need to check whether two particles are far enough apart to collide. So, using a linked list in a 3D array, add it to a linked list according to the position of the particles. In addition, the amount of computation required is reduced by only checking for collisions for particles in the linked list around the added linked list.

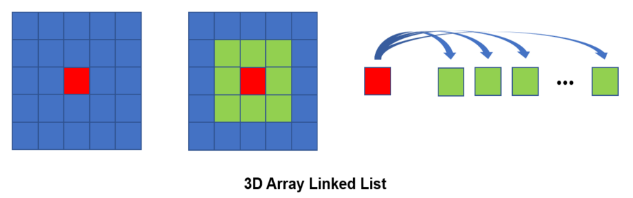




**5) Algorithm**

Each frame, the particle's data is updated and rendered. The particle's data is updated in the order of acceleration, velocity and position. The update process is as follows. When a particle collides, the two particles recalculate the velocity and acceleration of each other as an inelastic collision. Collision checking proceeds as described in the data structure section above. The next step is to add gravitational acceleration to the particle's velocity. In this process, the acceleration value for the particle is updated. When you add the particle's acceleration to the particle's velocity, the velocity data for the particle is updated. Finally, as the particle moves by its velocity, the position data is updated.





When the particle's data is updated, the new position of the particle must be reassigned to the 3D array linked list. This is because it is possible to use it in the next frame and create volume data for volume rendering. The volume data is created and rendered using the reallocated 3D array linked list.

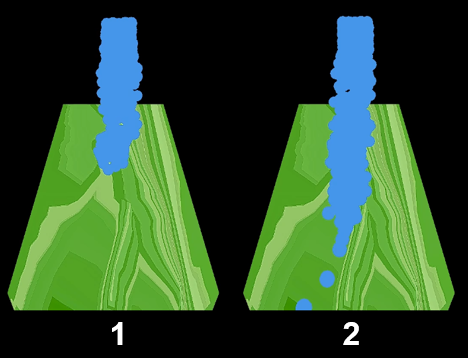
**4. Result**

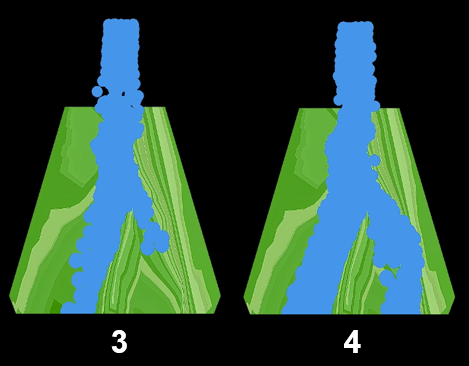
The program was divided into face rendering and volume rendering. Surface rendering is to ensure that particles move as intended, without visual effects, and that particle collisions occur as intended. Volume rendering is the addition of visual effects to surface rendering. In addition, randomness was added to the direction of the initial velocity vector of the generated particle in order to slightly different each simulation result. The orientation of the particle's initial velocity vector is generally similar, but is always exactly different.

In the part of the particle's motion, half the expected result was obtained, and the half was observed to move in an unexpected direction. The results for the particle rendering part were satisfactory. Using the volume rendering using threshold values, the two particles look quite like fluid viscosity.

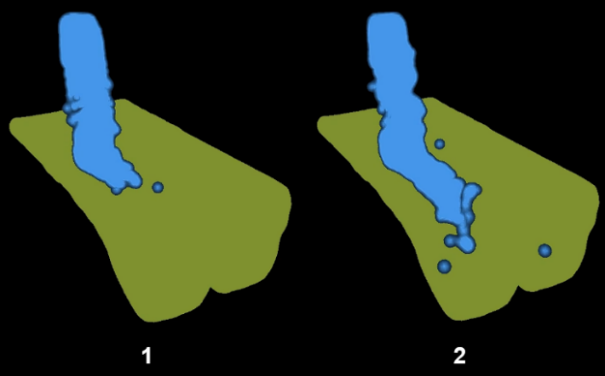
The code for the simulator is at 'https://github.com/bae4969/SPH'. You can debug using visual studio. When you press the keyboard key, there are various functions such as rotation, zoom in, zoom out, pause, and check the number of particles. Keystrokes are detailed in the README.md file of the link. It also describes how you can choose between face rendering and volume rendering. There may be a problem with debugging, so the result video is saved in the result video.

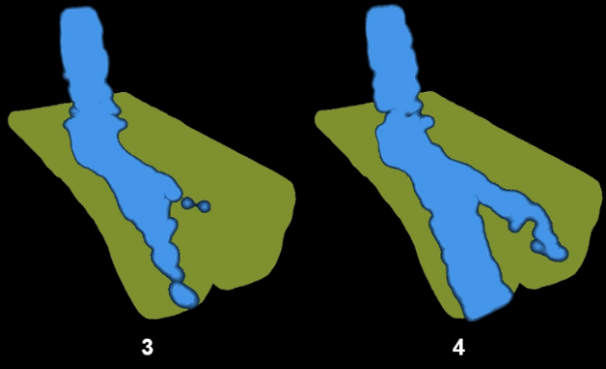
The figure below is the simulation result for surface rendering. Particles flow to the left valley first and flow to the right valley from a large valley. In the first curve of a large valley, the direction of the particles is deflected to the left and flows first in the small valley on the left. Then, when a certain amount is reached in the left small valley, particle flow to the right small valley. The same result is always shown even in the results of several simulations.





The figure below is the simulation result for volume rendering. Volume rendering also shows a similar trend to face rendering. In order to accurately see the flow of fluid, fluid particles are always visible rather than terrain particles.





The figure below shows the particles as intended as described in 'Particle Rendering' part in volume rendering.



**5. Discussion**

General objects are expressed by applying appropriate textures on the planes that connect the vertices. However, unlike ordinary objects, fluids change shape all the time, and when two fluid masses get closer, they change into one mass. Besides, it is also viscous. The characteristics of these fluids were difficult to express with rendering methods applied to general objects. However, if you use particles like this project, you can calculate fluid flow to some extent. Additionally, it was possible to express the fluid to some extent by using the location of the particles. To some extent means not perfect. Two problems were found in this project.

The first problem with this project is that sometimes particles move in unpredictable directions. Probably, it is expected that the collision determination between particles was not made continuously. Because real particles are continuously collided, the particles do not overlap. However, in the program, collision determination is not made completely consecutively. This is because particles in the program move discontinuously. Taking this point seriously, I tried to update the frames at the shortest possible interval. However, the reduction in the frame update interval did not improve this point. It seems that continuous laws of physics are needed rather than discrete laws of physics. So, it is expected that there will be better results if using the attraction and repulsion force according to the distance of particles. This is because, when the force is integrated in time, it becomes the moving distance, and even if the attraction and repulsion are discontinuous, it can have an integral value similar to the case of continuous to some extent.

The second problem is the running speed of the simulator. In this simulation, neither large terrain nor many particles were used. However, this simulator runs slowly. As a result of the analysis, the bottleneck was found in the CPU rather than the GPU. The frame update rate seems to be slow because each particle also checks for collisions, moves, and creates data for volume rendering. So, if you program using multi-core instead of single-core, it is expected to show better performance. Or, it would have been better if the CPU had been programmed so that the GPU could replace the computation.