**Fluid Simulator with SPH**

**1. Introduction**

This project is an experiment on a program that simulates fluids. I thought it was important to solve both fluid graphics rendering and fluid motion in order 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, 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 particles. So, I thought that if these particles were expressed well enough, they would look like fluid. Additionally, I thought that using this particle model could represent the motion of the fluid. For example, people say that sand flows like water in a sandy desert. I thought that if I used a particle model like this, I could express both fluid rendering and motion, and I started the project.

**2. Method**

**1) Algorithm**

The algorithm for particle motion is divided into three parts. The first is to apply gravity and the second is to determine the collision between particles and finally move the particles.

The first application of gravity simply adds gravity to the velocity of the particle. At this time, the magnitude of gravity was defined as the gravity acceleration times the time speed.

The second is interparticle collisions. Collision of particles is performed when two particles are below the reference. When colliding, it divides the particle's motion vector into two parts. One is a vector perpendicular to the collision plane of the two particles, and the other is a non-vertical vector. After collision, the vector of two particles coincides with the post-collision motion vector by exchanging a vector perpendicular to the collision plane of each particle. Decomposing a vector is simple. Using the positions of the two particles, I can find the normal vector of the collision plane. And using this normal vector, vector decomposition is possible.

The final step is to add the final velocity of the particle to the location. Again, speed should be multiplied by time speed.

Time speed was still used in the three-step algorithm above. The higher this time speed, the more inaccurate the particle movement. Therefore, time speed must be small enough.

It created terrain for the simulation environment. The terrain must remain in the same position as it collides with the particles in the fluid. The collided particle should then reflect the movement relative to the normal value of the terrain. In order to satisfy this condition, it made particle form meshing process. First, I designed a triangular mesh of terrain. Then, two vectors were created using three points of the mesh. Using these two vectors, create terrain particles at a constant rate. These terrain particles assign the mesh's normal vector to the velocity vector differently from the fluid particles. The normal vector is then used in the collision determination phase.

**2) Data Structure**

It used two classes in this project. One is the class for one particle and the other is for the whole simulation.

Particle class contains information about one particle. This information includes position values and speed values.

The class for the whole simulation consists of Particle Matrix and output. The particle matrix is a three-dimensional array. One section of the matrix represents 1x1x1 space. And the particles for that zone are connected in list form. Why use multiple lists with matrices instead of single lists because of time complexity? If you use a single list, it should check for collisions with one particle and all other particles in the collision detection phase. On the other hand, for multiple lists, the amount of computation required per particle is reduced because only collisions that are part of the list of adjacent regions need to be collision-checked. Instead, because of this structure, it must decide whether to pop and push based on the particle's position each time you run the third step of the algorithm.

The output is a 1-byte array to make a 3d texture. The size of this array is the size of the particle matrix multiplied by the height, width, depth and 9 of the products. The reason for multiplying 9 is to create 9 sampling points every 1/3 x 1/3 x 1/3 in a 1x1x1 space. Each sampling point is the sum of the values obtained by applying the distance of the surrounding particles to the standard distribution function. If it exceeds 1 byte, just assign 255. This 3d texture is used for the volume shader.

**3. Code Description**

**1) physics.h and physics.cpp**

'physics.h' and 'physics.cpp' contain the classes in the code that describe the data structure. Here you can use the class public member function to easily create and modify particles and create calculations for the next frame. Conversely, modifying these functions can apply new laws of physics.

**2) terrain.h**

Points and index values for the terrain are stored. And it contains the functions to make a triangle into a particle triangle and calculate the normal value using these points and indexes.

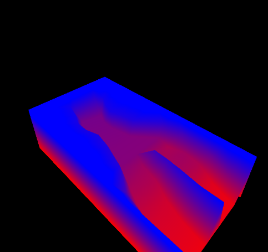
**3) graphic.h and graphic.cpp**

'graphic.h' and 'graphic.cpp' contain the simulation contents. Run the simulation using the functions in physics.h and the functions in terrain.h. Pressing a specific key allows you to rotate the simulation and change the sampling rate terrain removal terrain, stop particle generation, and pause the simulation. QWEASD rotates and 'T' can remove terrain. 'X' can see the terrain when it's sphere shading. 'O' stops particle generation and 'P' stops simulation. '<' and '>' specify the sampling rate for volume shading. If running at the lowest 1, the simulation will run a bit faster. Key input does not matter in lowercase letters. And at the top of 'graphic.cpp', you can decide the shader of the particles according to the define 'SPHERE\_SHADER' or 'VOLUME\_SHADER'.

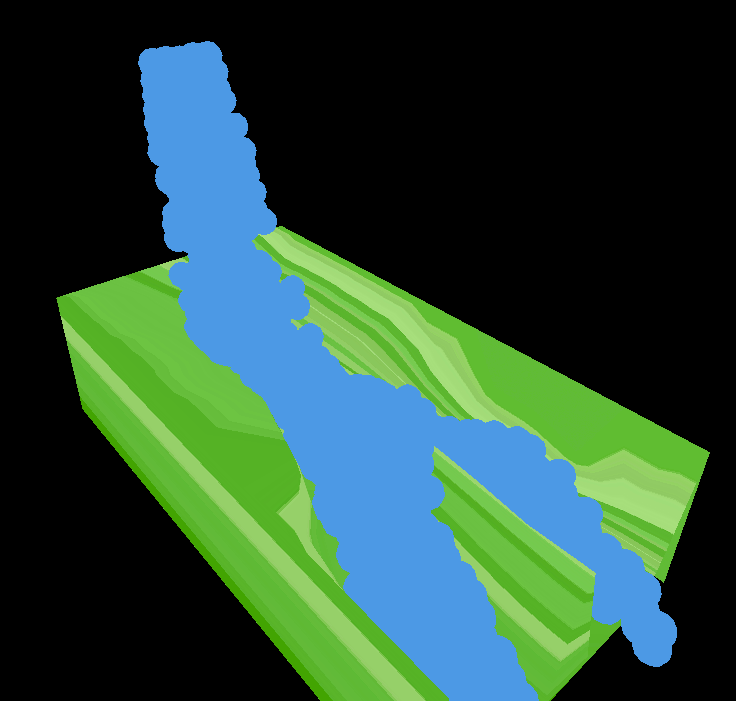
**4) etc.**

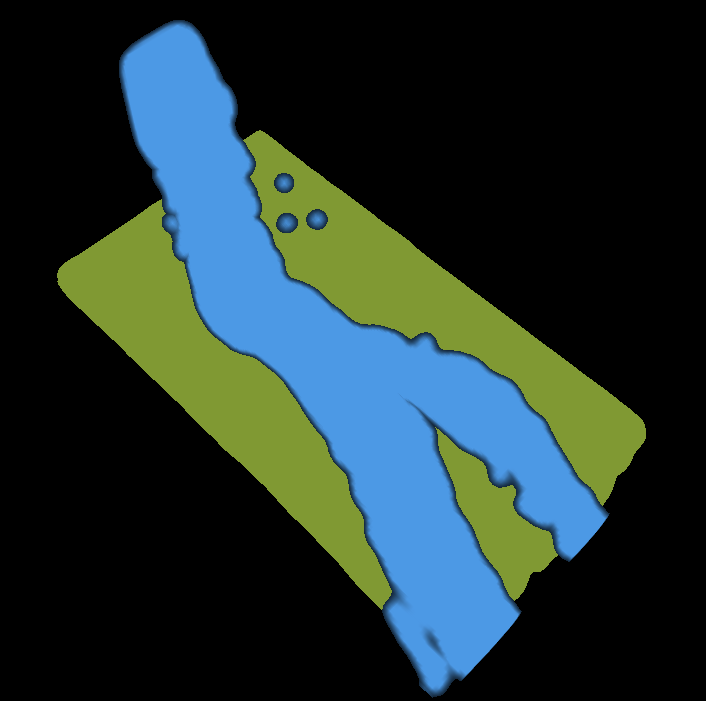
'bmploader.h' and 'textfile.h' are the files given in the last assignment.

**4. Result**

 The results were simulated on the terrain below. Blue means high terrain, red means low terrain. And fluid is a situation where particles, like a waterfall, are continuously generated at regular intervals rate from locations (1,20,7) to (1,27,12) with velocity (3,0,0). Position and velocity values are slightly different each time because they use the rand function.

One result is simply a particle representation of a sphere without using volume shading, and the other is the result of volume shading. The particle is applied on the terrain, so the particles are projected. The final picture shows how shading is applied as intended. When running the program, the simulation is very slow, so I attached a video file with speed up after recording for a long time.

-Without Volume Shading

-With Volume Shading

-Reason why applied threshold

**5. Improvement**

Originally, it was intended to apply viscosity. It is actually in the 'viscosity' function in 'physics.h'. However, I did not include it because there was not big difference in results. Also, the simulation speed becomes low. I also planned to use open MP to speed up the simulation through parallel computing. But I spent more time on accuracy than the speed of the simulation.

When particles are trapped in a certain space, they continue to collide with gravity constantly. So, the particles don't stop and pop. It is anticipated that a way to correct this problem would be to create a more accurate fluid simulator.