Fluid Simulation Using Instancing and Compute Shader

Yuke Fu* Su Li† Zhenghang Wu‡

Lund University
Sweden

Abstract

1 Introduction

Particle-based fluid simulation is a computational modeling technique that uses particle methods to simulate and analyze the motion and behavior of fluids. In this approach, "particles" carry physical properties of the fluid such as mass, velocity, and pressure. As time progresses, the states of the particles need to be updated according to physical laws and interaction rules, thereby simulating the motion of the fluid.

In order to make the fluid effect more realistic, we need plenty of particles, along with significant computation resources. In 2022, openGL introduced the concept of compute shaders, which means we can distribute a large amount of computation to the GPU, thereby reducing the load on the CPU and increasing overall efficiency.

Our project aims to utilize the compute shader to perform all the computation of particle's properties (velocity, position,force,....) on the GPU in order to implement fluid particle simulation(2D/3D) .

*fufraker@gmail.com
†lisu1017@163.com
‡zhenghangwu01@gmail.com

2 Application

2.1 Particle Spawner

A structure named *ParticleSpawnData* is designed to store the spawned particles' data, including their positions and velocities. The structure is included in a class named *ParticleSpawner*, which has a function *GetSpawnData* to generate the particles, give them initial velocities, and return the data.

2.2 Calculation

The calculation of particles' motion is done within the compute shader, which was supported by OpenGL in 2022. The particles' data and other parameters like value of gravity are passed to the compute shader.

For each particle, it has a *predictedPosition* value to predict its movement. In every calculation loop, the density and near density values at the predicted position will be calculated firstly. The density is only affected by the particles within the smoothing radius. The effects from particles that are farther than the smoothing radius will be ignored. Let D_1 and D_2 be the density and near density. Further, let d and r be the distance between current particle and neighbour particles and smoothing radius. Then the density and near density calculation functions are as follows:

$$D_1 = \frac{6(r-d)^2}{\pi r^4} \tag{1}$$

$$D_2 = \frac{10(r-d)^2}{\pi r^5} \tag{2}$$

Given the target density t, pressure multiplier m_1, and the near pressure multiplier m_2, the pressure P_1 and near pressure P_2 from the density are calculated as:

$$P_1 = m_1(D_1 - t) \tag{3}$$

$$P_2 = m_2 D_2 \tag{4}$$

With two particles' pressure and near pressure, we can calculate the mutual pressure force F between them in the following way:

$$\vec{F} = [\frac{-6(r-d)(P_{11}+P_{21})}{\pi r^4} - \frac{15(r-d)^2(P_{12}+P_{22})}{\pi r^5}]\vec{d} \tag{5}$$

Considering the effect of gravity, the acceleration of particle is:

$$\vec{a} = \frac{\vec{F}}{D_1} + \vec{g} \tag{6}$$

After completing the previous tasks, we have discovered that the particles are too chaotic at the start of the project. To solve this problem, we tried to add some friction (viscosity) between the particles so that the velocity of one particle would influence the neighbour's velocity, leading to the velocity of the fluid regions being blurred together. What we did is iteratively traverse all the particles within the smoothing radius to calculate the influence I between the particles. Then we added the influence of viscosity to the velocity which meant as time went by, the velocities of all particles would become more like their neighbours.

$$I = \frac{4(r^2 - d^2)^3}{\pi r^8} \tag{7}$$

Now let $\vec{v}_i$ be nearby particles' velocities, dt be the time interval between calculations, the velocity of particle can be calculated as:

$$\vec{v} = \vec{v} + 0.0005\vec{a} \cdot dt + 0.06 \sum (\vec{v}_i - \vec{v}) \cdot I_i \cdot dt \tag{8}$$

2.3 Instancing

For the rendering part of the project, since we are using very small circular meshes as individual particles in the fluid simulation, we need to render thousands of meshes in order to achieve a less granular fluid performance, using a loop to traverse the rendering so many times will greatly reduce the performance, which can be reflected in the significant reduction in the frame rate, OpenGL provides instancing rendering techniques like API glDrawArraysInstanced() and glDrawElementsInstanced(), which allow us to render multiple instances in a single draw call. What we need to do is setting the number of instances to be rendered, passing the position and velocity information to the shader, so we can modify the position and color of each instance to achieve the purpose of batch rendering. After using batch rendering, the and performance of the project has been significantly improved.

3 Result

The final result can be seen in figure below where the colors of the particles represent the speed of particles, the redder the faster the particle moves.

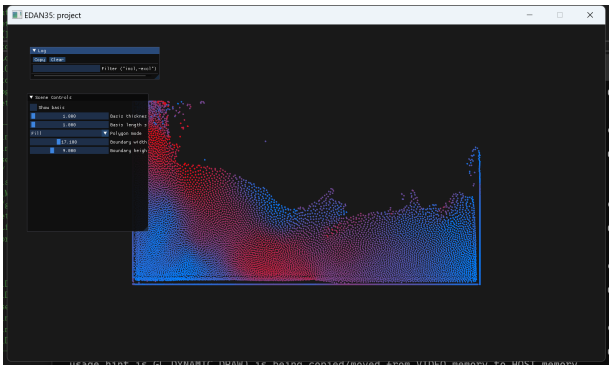

Figure 1: 2D fluid simulation.

4 Discussion

4.1 3D version

The result of 2D fluid simulation are convincing with realistic particle movement. A 3D version would be better for visual effects, and there should be more interaction options. Unfortunately, we encountered much more bugs than we expected when we tried to transfer the project into a 3D one. After debug-

ging, we discovered that the major issue was the mismatch of data size between the CPU and GPU. Although we used methods like forced alignment, each particle still exhibited an anomalous value in the z-direction after rendering, leading to inaccuracies in the following force calculations. Despite this problem, our project required further optimization. Currently, even though all calculations are performed on the GPU, the frame rate is still not very high. We are considering the implementation of advanced methods like spatial hashing to simplify the calculation so we will not have to traverse all the particles.

4.2 Data transfer optimization

For the data transfer, every time the position is updated, the calculated particle position and velocity need to be obtained from the GL_SHADER_STORAGE_BUFFER in the compute shader, and then when rendering, this data is passed to the GL_ARRAY_BUFFER for rendering, due to the fact that every time rendering has been performed two times of data transfer from CPU to GPU, resulting in a much lower final frame per second, if it is possible to directly transfer the data from the GL_SHADER_STORAGE_BUFFER to the GL_ARRAY_BUFFER, performance will be greatly improved. data transfer from CPU to GPU twice for each rendering, resulting in a much lower final frame rate. If we could directly transfer the data from GL_SHADER_STORAGE_BUFFER to GL_ARRAY_BUFFER, the performance would be greatly improved. But unfortunately, we did not find a way to do that in OpenGL.

Reference

Sebastian Lague. Coding Adventure: Simulating Fluids. https://www.youtube.com/watch?v=rSKMYc1CQHE&t=515s.

Joey de Vries. LearnOpenGL-Instancing. https://learnopengl.com/Advanced-OpenGL/Instancing.

Joey de Vries. LearnOpenGL-Compute Shaders. https://learnopengl.com/Guest-Articles/2022/Compute-Shaders/Introduction.

Task Distribution

Yuke Fu: Relization of main function, program logic design, rendering optimization, team code integration and debugging.

Su Li: Realization of compute shader (the algorithms for both 2D and 3D) and help debugging.

Zhenghang Wu: Realization of compute shader (mainly 2D), boundary box scaling function and debugging.