Real-Time 3D Fluid Rendering Engine

COMP 8045 – Major Project 1

Harley Guan – A00999595
4-14-2024

Table of Contents

1. Introduction

1.1. Student Background

A developer with a rich academic background and extensive practical work experience. Previously employed as a DevOps developer at a technology company, accumulating three years of full-time work experience. Possesses solid expertise in both front-end and back-end development. Demonstrates exceptional mathematical proficiency, achieving high scores in mathematical and physical disciplines at BCIT.

1.1.1. Education

- South China University of Technology 2011-2016
  - Information Engineering, Bachelor's Degree
- British Columbia Institute of Technology 2017-2018
  - Computer Systems Technology (Cloud Computing Option)
- British Columbia Institute of Technology 2022-now
  - Computer Systems (Games Development Option), Bachelor of Technology

1.1.2. Work Experience

DevOps Developer
Fortinet
2019-2021

- Designed and maintained automation pipelines for departmental projects, utilizing Jenkins and Docker for seamless code submission, testing, and deployment processes.
- Independently developed and managed the BugScreen project, providing the team and managers with a visual bug submission, tracking, and reporting tool. Front-end development using TypeScript and Angular, backed by Couchbase as the database.
- Collaborated on an internal training game project developed using the Godot engine, assisting in the development of real-time scoreboard components for the web platform using TypeScript.

1.1.3. Mathematics & Physics Ability

- COMP 1113 Applied Mathematics 91
- COMP 2121 Discrete Mathematics 98
- MATH 7908 Linear Algebra & Applications 100
- COMP 8903 Physics for Games Development 99

1.2. Project Description

The primary objective of this project is to create an advanced rendering engine designed to simulate the real-time movement of fluids in a 3D environment, adhering to the laws of physics. This rendering engine can be employed in game CGI production and interactive in-game environments.

The project predominantly relies on computer graphics technology to design and develop the rendering engine. The engine is specifically tailored for simulating and rendering 3D fluids based on physical laws, employing the Position-Based Fluids (PBF) algorithm or Smoothed Particle Hydrodynamics (SPH) algorithm for computational simulation. Under the influence of gravity and other external forces, the

fluids are expected to generate real-time visual effects such as splashes and ripples. Fluids are also capable of dynamic interactions with other objects within the environment, following physics-based motion principles. Additionally, environmental light interacts with the fluid's surface, resulting in realistic optical effects through reflection and refraction, in accordance with physical principles.

While ensuring the fidelity of simulations, optimizations are carried out within the permissible limits of both CPU and GPU capabilities, enabling real-time simulation and rendering at high frame rates. Ultimately, this 3D fluid rendering engine allows for the flexible adjustment of physical parameters such as fluid volume and position, providing an affordable solution for high-quality simulations.

In the end, a real-time rendering demo will utilize this rendering engine to demonstrate the interactive effects between fluids and players or other in-game objects.

1.2.1. Essential Problems

With the rapid advancement of Graphics Processing Units (GPUs), high-tech fields such as the gaming industry, artificial intelligence, and data science are continuously innovating and breaking boundaries. The increasing power of GPUs has brought about robust real-time computational speed and graphic processing capabilities. This development allows various sophisticated rendering techniques to gradually achieve real-time computation, enhancing the visual experience in games.

Fluid simulation has always been a demanding area concerning rendering computations. In non-real-time rendering fields such as film and TV CGI production, different fluid simulation algorithms are utilized for various scenes. These include grid-based Eulerian methods, which use Navier-Stokes equations to track fluid properties, as well as particle-based methods like Smoothed Particle Hydrodynamics (SPH), which represent fluids as interacting particles and simulate liquids through these interactions. Of course, there are many other excellent simulation and rendering algorithms, but these algorithms face limitations when applied in real-time computational scenarios on gaming platforms. Due to the frame rate demands of real-time rendering computations, both grid-based and particle-based algorithms have to compromise by reducing the total number of grids or particles to balance computational capabilities and visual effects.

1.2.2. Goals and Objectives

The goal of this project is to create a real-time physics simulation and rendering engine for 3D fluids based on the PBF algorithm. The engine aims to achieve the following objectives:

- Establish a particle-based fluid simulation system and apply the PBF algorithm for physical simulation calculations.
- Utilize GPU efficiently through graphics APIs to perform parallel computation during the simulation process, achieving real-time simulation effects.
- Develop a rendering engine to handle particle fluids and perform real-time rendering.
- Transform particle fluids into realistic fluid effects through suitable algorithms.

2. Body

2.1. Background

The background of this project lies in the rapid advancements in GPU hardware performance, enabling real-time execution of many complex computations in computer graphics. Particularly in the field of professional game development, the powerful parallel computing capabilities of new GPU architectures allow numerous complex physics simulations to be executed and rendered in real-time.

Over the decades of continuous development in computer graphics, numerous outstanding algorithms have been proposed and experimentally validated for their performance. In the domain of fluid simulation, two primary simulation approaches have emerged: particle-based simulation system design and grid-based simulation system design. Among these, particle-based simulation algorithms such as Smoothed Particle Hydrodynamics (SPH) and Position-Based Fluids (PBF) algorithms are prominent.

The core idea of the SPH algorithm is to decompose fluid into many small particles and simulate fluid behavior by moving these particles in space. Each particle has certain attributes such as position, velocity, density, etc., and local physical quantities are calculated through weighted averages of surrounding particles.

The inspiration for the PBF algorithm comes from SPH, but it adopts a different approach to address some issues in fluid simulation. PBF emphasizes position constraints and iterative solving when computing fluid states, making it more stable and easier to implement. These algorithms have been proven to achieve real-time computational simulations through the powerful parallel computing capabilities of GPUs.

2.2. Project Statement

This project is an individual endeavor aimed at enhancing skills and knowledge relevant to computer graphics for future professional game development. It will focus specifically on the domain of 3D fluid simulation. The project will provide a complete process from applying 3D fluid simulation algorithms, utilizing GPU for real-time computations through graphics APIs, to ultimately rendering realistic fluid effects.

2.3. Possible Alternative Solutions

This project aims to develop a complete process for real-time 3D fluid simulation and rendering. There are several alternative solutions available for each part of this process, which are explained below:

2.3.1. Fluid Simulation Solutions

The solutions include the Smoothed Particle Hydrodynamics (SPH) algorithm and the Position-Based Fluids (PBF) algorithm.

- Smoothed Particle Hydrodynamics (SPH):

SPH algorithm approximates fluid as a continuous medium using a series of particles. Each particle represents a small portion of the fluid and carries physical properties such as mass, density, pressure, and velocity. The steps involved in SPH computation are as follows:

1. Particle Representation: Fluid is discretized into a series of particles, each carrying information about fluid properties.
2. Kernel Function: Particle properties are smoothed using a kernel function, which defines how particle properties are influenced by surrounding particles. The kernel function typically has a finite support range, affecting only neighboring particles.
3. Density Estimation: For each particle, its density is estimated by a weighted sum of mass and kernel function values of neighboring particles.
4. Pressure Calculation: Pressure is calculated based on the particle density using the equation of state, typically derived from ideal gas law or other relevant equations.
5. Force Computation: Force computation involves pressure gradient force and viscosity force, which are used to determine particle acceleration.
6. Time Integration: Numerical integration methods (such as Euler method or Verlet integration) are used to update particle velocities and positions.

- Position-Based Fluids (PBF):

PBF is a position-based fluid simulation method that emphasizes using position constraints to maintain fluid incompressibility and other physical properties. The core idea of PBF is to ensure that particle positions are computed by satisfying predefined constraints that ensure the physical behavior of the fluid. The basic steps of PBF include:

1. Particle Representation: Similar to SPH, fluid is discretized into particles, with each particle representing a small portion of the fluid and carrying position information.
2. Constraint Setup: Define a series of constraints, such as density constraints, to ensure fluid incompressibility. Density constraints ensure that each particle's neighborhood has sufficient particle density.
3. Constraint Solving: Use iterative solvers (such as Gauss-Seidel or Jacobi iterators) to solve these constraints. Adjust particle positions iteratively until all constraints are satisfied.
4. Integration and Collision Handling: After updating particle positions, handle possible collisions and boundary conditions to ensure particles do not enter prohibited areas.
5. Velocity Update: Finally, update particle velocities, usually derived from changes in position.

2.3.2. Fluid Rendering Solutions

For rendering particle information calculated by fluid simulation algorithms, suitable algorithms are needed to process particle information into renderable models. Alternative solutions include Marching Cubes algorithm and Screen Space Fluid Rendering algorithm.

- Marching Cubes:

Marching Cubes is a widely used graphics algorithm for extracting iso-surfaces from 3D scalar fields. The algorithm steps are as follows:

1. Space Partitioning: Divide the entire data volume space into a regular grid, with each cell being a cube.
2. Sample Point Classification: Sample scalar values at the corners (grid points) of each cube, where these values typically represent some property (such as density or temperature).
3. Determine Cube Configuration: Based on the scalar values of cube vertices relative to the selected threshold value, determine the configuration of the cube. There are 256 possible configurations (as each cube has 8 vertices, each vertex can be above or below the threshold).
4. Triangulation: Based on each cube's configuration, obtain the corresponding triangle settings from a predefined lookup table. These settings specify how to connect vertices to form one or more triangles, approximating the intersection of iso-surfaces within the cube.
5. Mesh Generation: Repeat the above steps until all cubes are processed, generating a triangle mesh of iso-surfaces for the entire dataset.

- Screen Space Fluid Rendering (SSFR)

Screen Space Fluid Rendering (SSFR) is a rendering technique designed to efficiently render fluids in real-time applications. This method relies on screen-space computation instead of traditional volume rendering methods, enabling fast generation of realistic fluid visual effects. The algorithm steps are as follows:

1. Particle Rendering: Render particles from fluid simulation into screen space, with each particle generating an influence region on the screen based on its attributes (such as depth and density).
2. Depth and Thickness Textures: Compute and record depth and thickness information of each particle on the screen. The depth texture stores the nearest depth values at each screen point, while the thickness texture accumulates thickness values of particles passing through each pixel.
3. Surface Reconstruction: Smooth the depth texture using image processing techniques (such as Gaussian blur), then use this texture data to reconstruct the geometric shape of the fluid surface in screen space.
4. Normal Computation: Calculate surface normals from the processed depth texture. Normals can be approximated from spatial gradients of depth data.
5. Lighting and Shading: Using computed normals and thickness information, along with traditional lighting models (such as Phong or Blinn-Phong), shade the fluid to enhance visual effects.

2.3.3. Graphic API Solutions

In order to implement the entire computation and rendering process, support from graphics APIs and some other third-party resources is necessary. For rapid simulation and visualization of simulation algorithms, Taichi Lang can be utilized for fast algorithm development, and to produce results processed by the GPU. For actual renderer development, Visual Studio needs to be used in conjunction with a graphics API. Available graphics API solutions include OpenGL, DirectX12, Vulkan, and NVIDIA's parallel computing platform, CUDA.

- Taichi Lang: Taichi Lang is an open-source programming language, developed to enhance productivity and efficiency in high-performance computing graphics and computational physics with succinct coding.

- OpenGL: OpenGL (Open Graphics Library) is a prevalent, cross-platform API employed for rendering 2D and 3D vector graphics, compatible with nearly all operating systems and graphics hardware.
- DirectX 12: DirectX 12 is a robust API provided by Microsoft, offering low-level hardware access aimed at rendering graphics and video, thereby significantly boosting gaming performance on Windows platforms.
- Vulkan: Vulkan is a contemporary graphics and compute API that offers efficient, cross-platform access to modern GPUs, recognized for its superior control over hardware resources and reduced CPU overhead.
- CUDA: CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) designed by NVIDIA, enabling developers to utilize NVIDIA graphics processing units (GPUs) for general purpose computations.

2.4. Chosen Solution

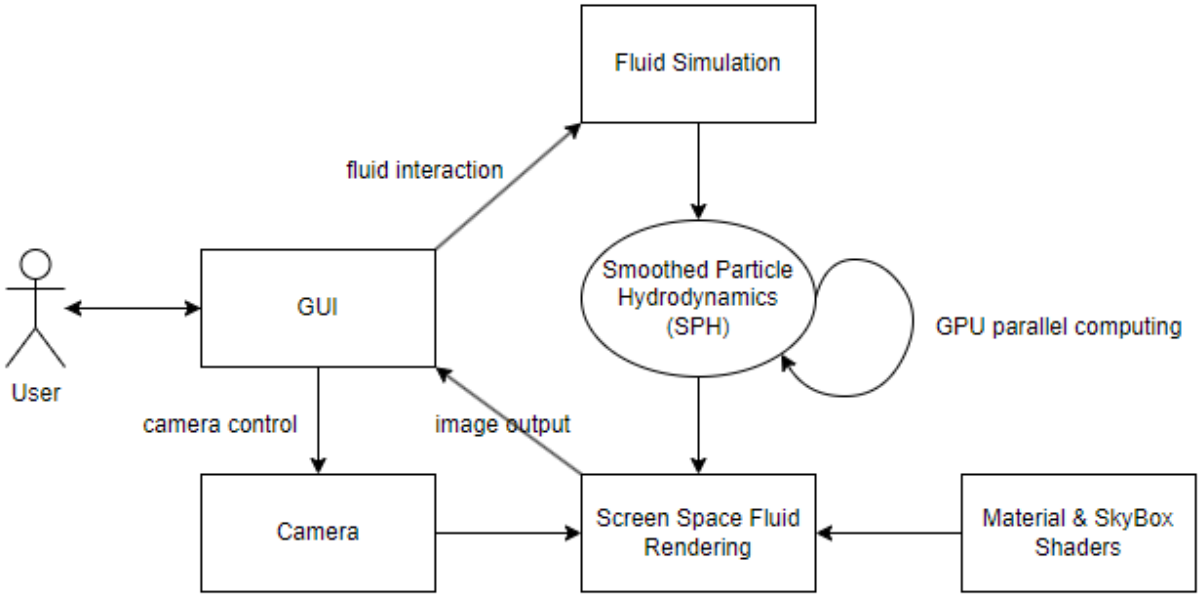
The chosen solution for this project is to use Taichi Lang for rapid development of fluid simulation algorithms based on SPH and PBF, and to visualize the efficiency and simulation performance of the two algorithms using GGui, a visualization tool provided by Taichi Lang. Taichi Lang was chosen because it avoids the cumbersome C++ development environment configuration and the differences in versions of CUDA or other APIs, effectively achieving rapid algorithm implementation and utilizing GPU parallel computation through Taichi Lang kernel calls to verify algorithm efficiency.

According to the comparison, it can be observed that both SPH and PBF algorithms support efficient real-time computation (see Section 2.6 testing details and result). I chose the SPH algorithm as the fluid physics simulation algorithm for the renderer, and combined it with the Screen Space Fluid Rendering (SSFR) algorithm to convert particles into renderable models and output them. The SPH algorithm is highly parallelizable and exhibits good numerical stability. SSFR technology relies on screen-space computation, enabling fast generation of realistic fluid visual effects. SSFR is particularly suitable for real-time simulation because it can render fluid dynamics very efficiently without sacrificing too much visual quality. For the selection of graphics APIs to invoke GPU parallel computation, the project uses OpenGL because it supports multiple operating systems and has extensive platform compatibility. Additionally, it serves as an ideal learning tool due to the abundance of learning resources and community support.

2.5. Details of Design and Development

2.5.1. General System Diagram

The system of this project mainly consists of a fluid simulation module and an SSFR rendering module.



The fluid simulation module defines a particle system for physical simulation. After initializing the particle set, all particle information is loaded into the GPU for SPH computation. Each round of computation results, along with the current camera information, is input to the SSFR module for rendering calculation. The rendering result is then output to the GUI for user observation and interaction.

Users can interact with the system through the GUI using the keyboard and mouse. Users can use the middle mouse button to drag the water for interaction, and the effects on the water will be transmitted to the fluid simulation module to update the fluid's force information. Users can also adjust the camera position by holding down the left or right mouse button and dragging the mouse. The camera position is continuously sent to the renderer and used for rendering, with the output image displayed on the GUI. Users can pause the current scene by pressing the space bar on the keyboard.

2.5.2. SPH Algorithm Principles and Design

The main task of fluid simulation is to solve the Navier-Stokes equations to obtain the acceleration field of the fluid, and then to obtain velocity and position through Euler integration. This project adopts the Lagrangian perspective, considering the fluid as a collection of particles with attributes such as mass and velocity to solve. This method mainly uses the SPH (Smoothed Particle Hydrodynamics) method for spatial discretization to approximate the density, pressure, and acceleration of each particle. There are many algorithms based on SPH, such as WCSPH, PCISPH, DFSPH, etc. This project uses the simplest WCSPH (Weakly Compressible SPH) algorithm for physical simulation of weakly compressible fluids.

2.5.2.1. WCSPH Algorithm Theory

First, the fluid is regarded as a collection of particles (mass points) with attributes such as mass, volume, density, pressure, velocity, and acceleration. It is important to emphasize that the particles here can be understood as mass points, meaning that mass is constant, while density and volume are variable.

Step 0: Initialize all particles' attributes:

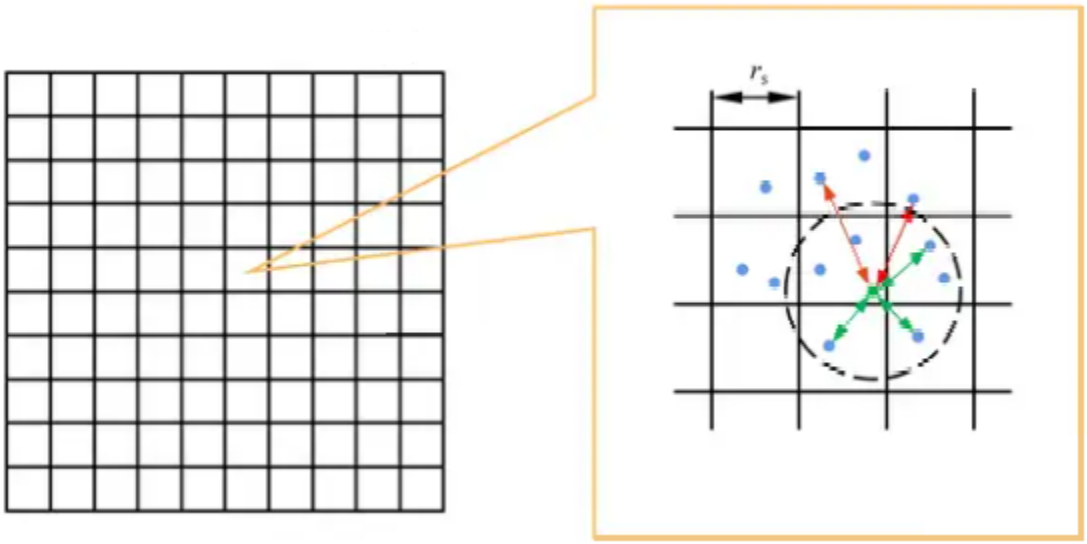
ATTRIBUTE	VALUE	PROPERTY
Particle radius r_p (m)	0.005	Constant
Volume V_{p0} (m^2)	0.005^2	Constant
Support radius r_s (m)	0.025	Constant
Weight m (kg)	$V_{p0} \times \rho_0$	Constant
Position p	-	Variable
Velocity v	-	Variable
Acceleration a	-	Variable

The support radius is chosen to be five times the particle radius. At initialization, the fluid density is exactly ρ_0, and the particles are uniformly distributed. At this point, the sum of the volumes of all particles within the support radius should be equal to the volume of the support domain.

$$V_{p0} = \frac{\pi r_s^2}{n}$$

Step 1: Neighborhood Search

The calculation of physical quantities for particles in SPH relies on weighted approximations based on neighboring particles within their support domain. Therefore, the first step is to find the neighbors of all particles, known as neighborhood search. This is accomplished using a block search approach, where the container containing the fluid is divided into several grids, as shown in the diagram.



Because grids possess a topological structure, neighboring grids' particles can be accessed directly by calculating a hash function, allowing for distance computation to determine if they are neighbors. Here, the grid's side length is set as the support radius, and only the nine neighboring grids (up, down, left, right, and diagonals) of the target grid need to be traversed.

To store this information, a two-dimensional array is used, representing the grid as an array with a length equal to the number of small grids. Each element in the array contains a set of particles "falling into" that small grid. The sequence of small grids follows a row-first arrangement. The hash function is defined as follows:

$$H_3(i, j, k) = j + i \times row + k \times row \times col$$

where i, j, k represents the i-th row, j-th column, and k-th layer, and row and col denote the number of rows and columns, respectively. With this hash function, we can find neighboring grids in constant time complexity.

Step2: Find the density and pressure of all fluid particles

First find the density based on the surrounding neighbor particles:

$$\rho_i = \sum_j \frac{m_j}{\rho_j} \rho_j W(r_i - r_j, h) = \sum_j m_j W_{i,j}$$

Next, find the fluid pressure based on density:

$$p_i = k_1 (\left(\frac{p_i}{p_0}\right)^{k_2} - 1)$$

In the formula k_1 is the fluid stiffness coefficient, k_2 is the pressure index. For simulate water generally takes 50 and 7.

The sum of the volumes of particles in the support domain of particles near the fluid surface may not reach the volume of the support domain. In this case, the calculated density will be less than ρ_0 , causing the pressure calculation result to be huge and causing solution errors. Therefore, after finding the density, we need:

$$\rho_i = \max(\rho_i, \rho_0)$$

This allows liquid compression but prohibits liquid expansion.

Step3: Find the acceleration caused by the viscous force of the particle

The viscous force is proportional to the second derivative of velocity (the Laplace operator). The greater the speed difference between the surrounding particles and this particle, the greater the viscous resistance this particle experiences. The solution formula for viscous acceleration is:

$$\nu \nabla^2 v_i = \nu \sum_j m_j \frac{v_j - v_i}{\rho_j} \nabla^2 W_{i,j}$$

This solution is obtained from the "antisymmetric form" of the SPH derivative. One property of SPH is that when differentiating a certain physical quantity, you only need to differentiate the kernel function in the approximate expression.

Step4: Find the acceleration caused by fluid pressure

Pressure acceleration is proportional to the gradient of fluid pressure. In other words, the greater the pressure difference between a particle and surrounding particles, the greater the pressure on the particle. The formula for solving the acceleration of pressure:

$$\frac{1}{\rho_i}\nabla p_i = \sum_j m_j(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2})\nabla W_{i,j}$$

IIn the formula, p_i, p_j, ρ_i, ρ_j are the particle pressure and density calculated in step 1 respectively. This formula is obtained from the symmetric form of SPH.

Step5: Euler integral updates particle position

In addition to the two accelerations calculated above, the fluid is also subject to its own gravity acceleration $\rho_i g$. At this point, the resultant acceleration of the fluid has been calculated. Add up all the accelerations to get the total acceleration of the particle:

$$\frac{dv_i}{dt} = g + \nu\nabla^2 v_i - \frac{1}{\rho_i}\nabla p_i$$

With the total acceleration of the particle, the velocity and displacement can be solved according to Euler integral. The form of Euler integral is as follows:

$$v_{i+1} = v_i + a_i\Delta t$$

$$x_{i+1} = x_i + v_{i+1}\Delta t$$

First, the initial values of velocity and displacement are given. Assuming acceleration, the velocity can be updated according to the first equation. With the velocity, the displacement can be updated according to the second equation.

Step6: Boundary conditions

In the simulation, the fluid will rebound after colliding with the boundary, that is, the velocity will reverse. When it is determined that the particles have reached the boundary, the speed of the particles reaching the boundary is directly reversed to realize the rebound of the particles.

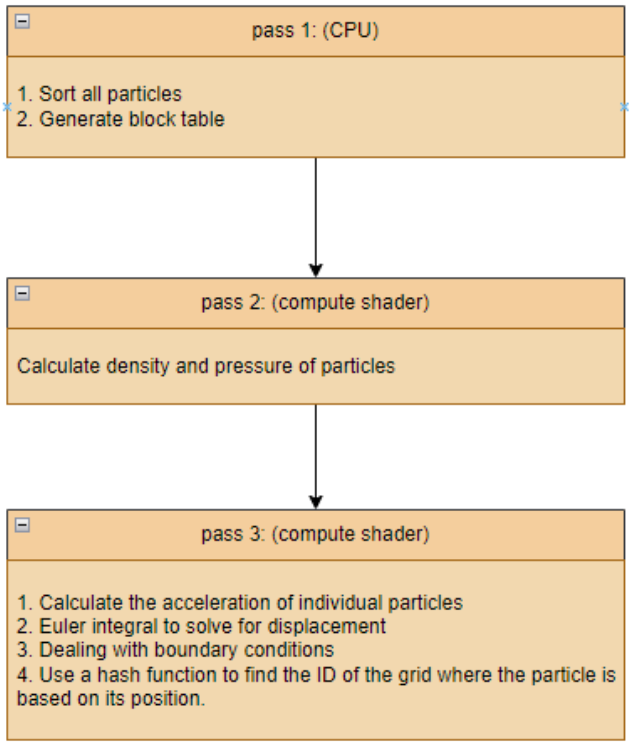
At this point, the positions of all particles have been updated for one round, which means an iteration step has been completed. With this iteration, the movement of the fluid can be simulated.

2.5.2.2. WCSPH Pipeline Design

The fluid simulation module of this project will implement the WCSPH algorithm described above, with different stages processed on both CPU and GPU. Since the task of Step 1, neighborhood search, involves finding all particles within a distance less than r from a given particle and requires acceleration, otherwise each particle would need to traverse all other particles, utilizing a grid partitioning acceleration method only requires searching particles within adjacent grids, eliminating the need to traverse all particles. Thus, a hash function is employed to calculate the grid in which each particle resides based on its position. Then, a table is computed where the index is the grid ID and the value is a list of particle IDs contained in that grid. Finally, for each particle, its grid is calculated using the hash function, and then, exploiting the natural topological relationships of the grid, adjacent grids are found, and particles within those grids are traversed to complete the neighborhood search.

To save memory usage, all particles are first sorted according to their grid IDs to obtain a particle array where particles within the same grid are contiguous. Then, a block table is computed, where each element represents a grid, with the index being the grid ID and the value being a segment of the particle array. The particles within these segments represent all particles within the respective grid.

After updating the algorithm's pipeline, the pipeline design is as shown in the diagram below:



Since CPUs are proficient in sorting operations, particle sorting and generating the block table will run on the CPU. The remaining computational tasks will be handled by the GPU using compute shaders. Since the operations from Step 3 to Step 5 of the algorithm are mutually independent for each particle and can be fully parallelized, they will be combined into a single pass.

2.5.2.3. WCSPH Code

Initialize block size:

```
void ParticalSystem3D::SetContainerSize(glm::vec3 corner, glm::vec3 size) {
    mLowerBound = corner - mSupportRadius + mParticalDiameter;
    mUpperBound = corner + size + mSupportRadius - mParticalDiameter;
    mContainerCenter = (mLowerBound + mUpperBound) / 2.0f;
    size = mUpperBound - mLowerBound;

    // numbers of blocks in x, y, z
    mBlockNum.x = floor(size.x / mSupportRadius);
    mBlockNum.y = floor(size.y / mSupportRadius);
    mBlockNum.z = floor(size.z / mSupportRadius);

    // size of one block
    mBlockSize = glm::vec3(size.x / mBlockNum.x, size.y / mBlockNum.y, size.z / mBlockNum.z);

    mBlockIdOffs.resize(27);
    int p = 0;
    for (int k = -1; k <= 1; k++) {
        for (int j = -1; j <= 1; j++) {
            for (int i = -1; i <= 1; i++) {
                mBlockIdOffs[p] = mBlockNum.x * mBlockNum.y * k + mBlockNum.x * j + i;
                p++;
            }
        }
    }

    mParticalInfos.clear();
}
```

The block size is determined based on the ratio of the boundary value and the support radius. After initializing the container, generate a block id based on the location.

Initialize the fluid:

```cpp
int32_t ParticalSystem::AddFluidBlock(glm::vec2 corner, glm::vec2 size, glm::vec2 v0, float particalSpace) {
    glm::vec2 blockLowerBound = corner;
    glm::vec2 blockUpperBound = corner + size;

    if (blockLowerBound.x < mLowerBound.x ||
        blockLowerBound.y < mLowerBound.y ||
        blockUpperBound.x > mUpperBound.x ||
        blockUpperBound.y > mUpperBound.y) {
        return -1;
    }

    int width = size.x / particalSpace;
    int height = size.y / particalSpace;

    std::vector<glm::vec2> position(width * height);
    std::vector<glm::vec2> velocity(width * height, v0);
    std::vector<glm::vec2> accleration(width * height, glm::vec2(0.0f, 0.0f));

    int p = 0;
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            position[p] = corner + glm::vec2((j + 0.5) * particalSpace, (i + 0.5) * particalSpace);
            if (i % 2) {
                position[p].x += mParticalRadius;
            }
            p++;
        }
    }

    mPositions.insert(mPositions.end(), position.begin(), position.end());
    mVelocity.insert(mVelocity.end(), velocity.begin(), velocity.end());
    mAccleration.insert(mAccleration.end(), accleration.begin(), accleration.end());
    return position.size();
}
```

The volume and position of the loaded fluid particles are defined through the corner variable and size variable. Initialize the physical quantities of all particles: position, velocity, acceleration.

Sort particles and generate block table (pass 1):

```cpp
void ParticalSystem3D::UpdateData() {
    // sort by block
    std::sort(mParticalInfos.begin(), mParticalInfos.end(),
        [=](ParticalInfo3d& first, ParticalInfo3d& second) {
            return first.blockId < second.blockId;
        }
    );

    // calculate block
    mBlockExtens = std::vector<glm::uvec2>(mBlockNum.x * mBlockNum.y * mBlockNum.z, glm::uvec2(0, 0));
    int curBlockId = 0;
    int left = 0;
    int right;
    for (right = 0; right < mParticalInfos.size(); right++) {
        if (mParticalInfos[right].blockId != curBlockId) {
            mBlockExtens[curBlockId] = glm::uvec2(left, right);
            left = right;
            curBlockId = mParticalInfos[right].blockId;
        }
    }
    mBlockExtens[curBlockId] = glm::uvec2(left, right);
}
```

Determine the order and sort the particle array according to the block id corresponding to the particle. Make particles of the same block close together. Then calculate the block table, use the block id as an index, and store the set of all particles inside the block.

Upload information to GPU and execute ComputeParticals.comp:

```
void RenderWidget::UploadParticalInfo(Fluid3d::ParticalSystem3D& ps) {
    // Apply for a buffer containing particle information
    glBindBuffer(GL_SHADER_STORAGE_BUFFER, mBufferParticals);
    glBufferData(GL_SHADER_STORAGE_BUFFER, ps.mParticalInfos.size() * sizeof(ParticalInfo3d),
        ps.mParticalInfos.data(), GL_DYNAMIC_COPY);

    // Apply for block table buffer
    glBindBuffer(GL_SHADER_STORAGE_BUFFER, mBufferBlocks);
    glBufferData(GL_SHADER_STORAGE_BUFFER, ps.mBlockExtens.size() * sizeof(glm::uvec2),
        ps.mBlockExtens.data(), GL_DYNAMIC_COPY);
    glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0);
    mParticalNum = ps.mParticalInfos.size();
}
```

```
void RenderWidget::SolveParticals() {
    if (mParticalNum <= 0 || mPauseFlag) {
        return;
    }

    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 4, mBufferParticals);
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 5, mBufferBlocks);
    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_1D, mTexKernelBuffer);
    glBindImageTexture(0, mTestTexture, 0, GL_FALSE, 0, GL_READ_WRITE, GL_RGBA32F);
    mComputeParticals->Use();
    mComputeParticals->SetVec3("gGravityDir", -Glb::Z_AXIS);
    mComputeParticals->SetVec3("gExternelAccleration", mExternelAccleration);
    mComputeParticals->SetInt("particalNum", mParticalNum);
    for (int pass = 0; pass <= 1; pass++) {
        mComputeParticals->SetUInt("pass", pass);
        glDispatchCompute(mParticalNum / 512 + 1, 1, 1);
        glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT | GL_SHADER_IMAGE_ACCESS_BARRIER_BIT);
    }
    mComputeParticals->UnUse();
}
```

Call the OpenGL API to transfer particle information to the GPU, and then use ComputeParticals.comp performs subsequent parallel operations.

Calculate density and pressure of particles (pass 2):

```
void ComputeDensityAndPress(inout ParticalInfo3d pi) {
    uint particalId = gl_GlobalInvocationID.x;

    for (int i = 0; i < blockIdOffs.length(); i++) {     // for all neighbor block
        uint bIdj = pi.blockId + blockIdOffs[i];
        for(uint j = blockExtens[bIdj].x; j < blockExtens[bIdj].y; j++) {   // for all neighbor particals
            vec3 radiusIj = pi.position - particals[j].position;
            float diatanceIj = length(radiusIj);
            if (particalId != j && diatanceIj <= gSupportRadius) {
                pi.density += texture(kernelBuffer, diatanceIj / gSupportRadius).r;
            }
        }
    }
    pi.density *= (gVolume * gDensity0);
    pi.density = max(pi.density, gDensity0);
    pi.pressure = gStiffness * (pow(pi.density / gDensity0, gExponent) - 1.0);
    pi.pressDivDens2 = pi.pressure / pow(pi.density, 2);
}
```

See step 2 for formula details. This function is defined in the shader file and executed by the GPU.

Calculate the acceleration (pass 3):

```
void ComputeAccleration(inout ParticalInfo3d pi) {
    uint particalId = gl_GlobalInvocationID.x;
    if (particalId >= particalNum) {
        return;
    }

    float dim = 3.0;
    float constFactor = 2.0 * (dim + 2.0) * gViscosity;
    vec3 viscosityForce = vec3(0.0);
    vec3 pressureForce = vec3(0.0);
    for (int i = 0; i < blockIdOffs.length(); i++) {     // for all neighbor block
        uint bIdj = pi.blockId + blockIdOffs[i];
        for(uint j = blockExtens[bIdj].x; j < blockExtens[bIdj].y; j++) {   // for all neighbor particals
            vec3 radiusIj = pi.position - particals[j].position;
            float diatanceIj = length(radiusIj);
            if (particalId != j && diatanceIj <= gSupportRadius) {
                float dotDvToRad = dot(pi.velosity - particals[j].velosity, radiusIj);
                float denom = diatanceIj * diatanceIj + 0.01 * gSupportRadius * gSupportRadius;
                vec3 wGrad = texture(kernelBuffer, diatanceIj / gSupportRadius).g * radiusIj;
                viscosityForce += (gMass / particals[j].density) * dotDvToRad * wGrad / denom;
                pressureForce += particals[j].density * (pi.pressDivDens2 + particals[j].pressDivDens2) * wGrad;
            }
        }
    }

    pi.accleration += viscosityForce * constFactor;
    pi.accleration -= pressureForce * gVolume;
}
```

See step 3 and step 4 for formula details. This function is defined in the shader file and executed by the GPU. The acceleration of particles is composed of the viscous force under the influence of surrounding particles and the fluid pressure.

Euler integral updates particle position (pass 3):

```
void EulerIntegration(inout ParticalInfo3d pi) {
    pi.velosity = pi.velosity + gDeltaT * pi.accleration;
    pi.velosity = clamp(pi.velosity, vec3(-gMaxVelocity), vec3(gMaxVelocity));    // velocity limit
    pi.position = pi.position + gDeltaT * pi.velosity;
}
```

See step 5 for formula details. This function is defined in the shader file and executed by the GPU. In order to avoid excessive speed causing particles to rush out of the container and causing the program to crash, the speed is clamped to the [-100, 100]m/s range.

Boundary condition processing (pass 3):

```
void BoundaryCondition(inout ParticalInfo3d pi) {
    bool invFlag = false;
    if (pi.position.x < containerLowerBound.x + gSupportRadius) {
        pi.velosity.x = abs(pi.velosity.x);
        invFlag = true;
    }
    if (pi.position.y < containerLowerBound.y + gSupportRadius) {
        pi.velosity.y = abs(pi.velosity.y);
        invFlag = true;
    }
    if (pi.position.z < containerLowerBound.z + gSupportRadius) {
        pi.velosity.z = abs(pi.velosity.z);
        invFlag = true;
    }
    if (pi.position.x > containerUpperBound.x - gSupportRadius) {
        pi.velosity.x = -abs(pi.velosity.x);
        invFlag = true;
    }
    if (pi.position.y > containerUpperBound.y - gSupportRadius) {
        pi.velosity.y = -abs(pi.velosity.y);
        invFlag = true;
    }
    if (pi.position.z > containerUpperBound.z - gSupportRadius) {
        pi.velosity.z = -abs(pi.velosity.z);
        invFlag = true;
    }
    if (invFlag) {
        pi.velosity *= gVelocityAttenuation;
    }
    pi.position = clamp(pi.position, containerLowerBound + vec3(gSupportRadius + gEps),
        containerUpperBound - vec3(gSupportRadius + gEps));
    pi.velosity = clamp(pi.velosity, vec3(-gMaxVelocity), vec3(gMaxVelocity));    // velocity limit
}
```

See step 6 for details. This function is defined in the shader file and executed by the GPU. Checks whether particles collide with boundaries and are reflected.

Calculate block ID (pass 3):

```
void CalculateBlockId(inout ParticalInfo3d pi) {
    vec3 deltePos = pi.position - containerLowerBound;
    uvec3 blockPosition = uvec3(floor(deltePos / blockSize));
    pi.blockId = blockPosition.z * blockNum.x * blockNum.y + blockPosition.y * blockNum.x + blockPosition.x;
}
```

Use the current position of the particle to determine which block it belongs to, and update the particle's block id.

The entire iteration of GPU computing:

```
void main() {
    uint particalId = gl_GlobalInvocationID.x;

    if (pass == 0) {
        ComputeDensityAndPress(particals[particalId]);
    }
    else if (pass == 1) {
        particals[particalId].accleration = gGravity * gGravityDir + gExternelAccleration;
        ComputeAccleration(particals[particalId]);
        EulerIntegration(particals[particalId]);
        BoundaryCondition(particals[particalId]);
        CalculateBlockId(particals[particalId]);
    }

    imageStore(imgOutput, ivec2(particalId % 100, particalId / 100), vec4(1.0, 1.0, 0.0, 1.0));
    return;
}
```

In each iteration, pass 2 is processed first and density and pressure are calculated. Then the operations in pass 3 are calculated in parallel for all particles, and the updated particle information is returned.

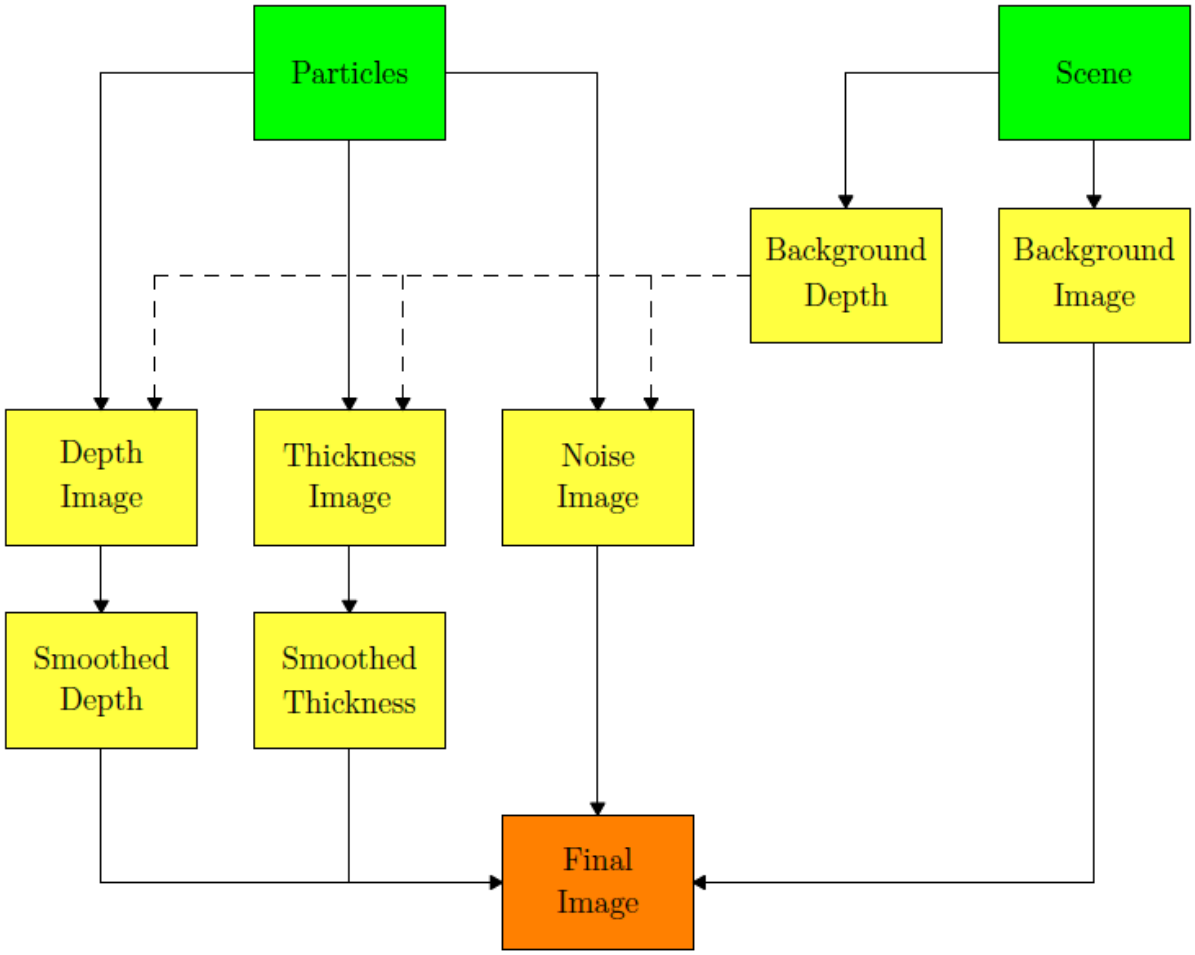
2.5.3. SSFR Algorithm Principles and Design

After the above-mentioned SPH operation module is completed, all particle information making up the fluid is updated. There still needs to be a pipeline that can handle this particle information and render it in the appropriate way. Since the final fluid needs to construct a non-particle model surface, the Screen-space Fluid Rendering (SSFR) algorithm is used here for fluid rendering.

2.5.3.1. SSFR Algorithm Theory

The screen-space fluid rendering algorithm directly reconstructs the fluid surface in screen space, reducing the operation dimension from three-dimensional to two-dimensional. This is a rendering algorithm tightly integrated with graphics processing. The algorithm is divided into two steps, with emphasis on the first step. The first step is the screen-space fluid processing step. Initially, fluid particles are rendered to screen space based on the current projection matrix and view matrix to obtain depth contour information and fluid thickness information, which are stored in textures. Subsequently, we perform some post-processing operations on the texture storing depth information to blur out the bumps and pits of the spherical particles, making the depth information smoother. Finally, based on the depth texture and thickness texture, we perform the lighting calculation for the fluid.

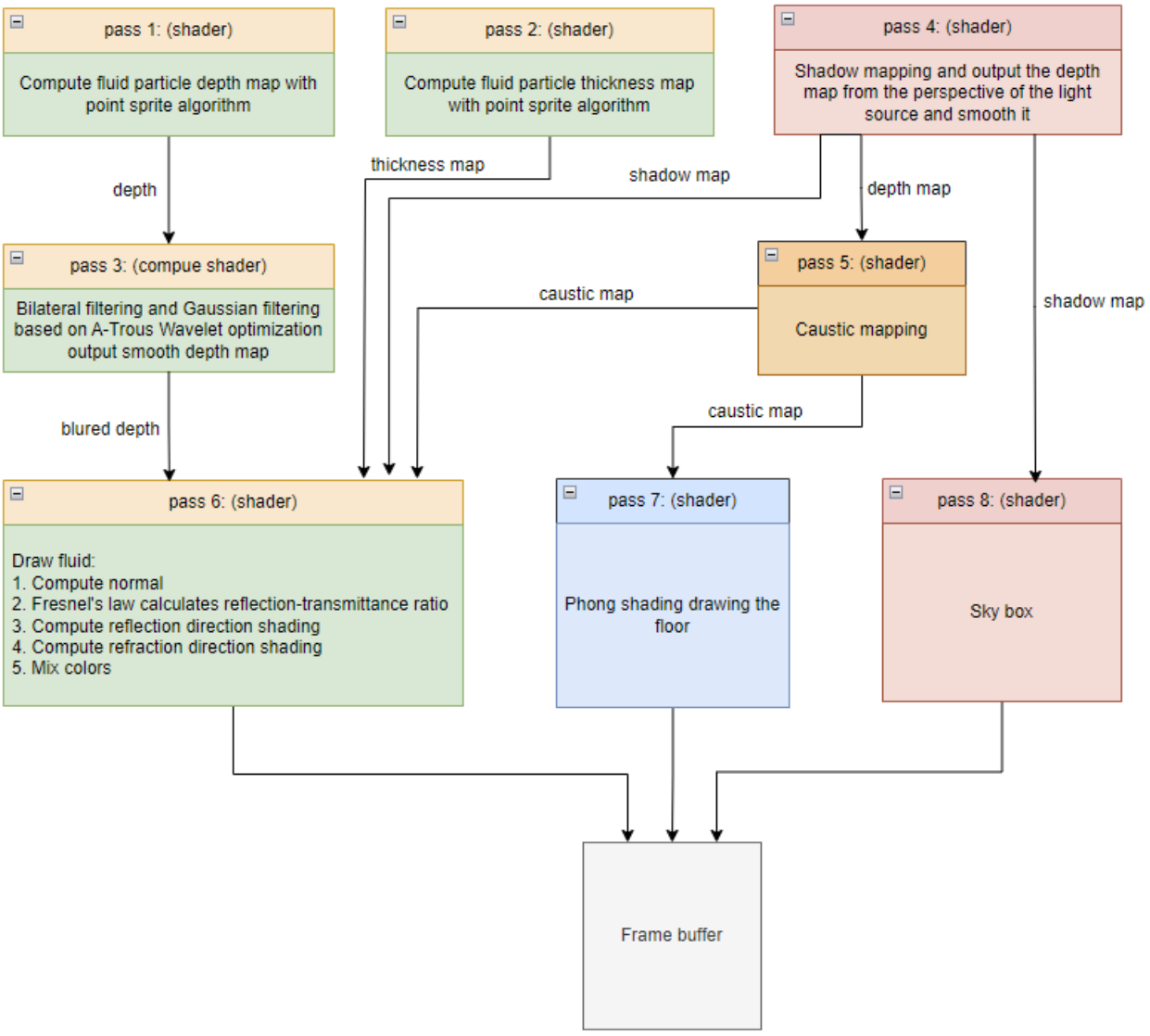
The overall process of the algorithm is illustrated as following:



The background image is used to compute the refraction of the fluid; therefore, the fluid should typically be rendered last. This algorithm can be viewed as a deferred rendering approach for fluids. Surface normal information of the fluid cannot be directly obtained; it needs to be reconstructed from the depth map, which also undergoes some special processing beforehand. In the screen-space fluid rendering step, how to process the depth map to accurately reflect the characteristics of the fluid is the core of the algorithm. The remaining lighting part directly employs the Blinn-Phong lighting model, considering both refraction and reflection of the water, where the thickness map mentioned above is used for refraction calculation, as thicker fluids have lower translucency.

2.5.3.2. SSFR Pipeline Design

Based on the above principle flow chart, the following pipeline diagram of the entire process of fluid rendering is formulated.



In the entire process pipeline, the optional noise image module was cancelled, because it is hoped that the surface of the water body will be smooth after rendering rather than bumpy. In this pipeline diagram, the green part is the screen space fluid rendering part, which is also the core part. The gray part is the rendering of the shadow map. In addition to providing the shadow map, this part also provides the depth map for the subsequent caustics rendering. Pass 5 is the caustics rendering part, which outputs a caustics map. The pass 7 part is responsible for drawing the floor; the pass 8 part is responsible for drawing the sky box.

Pass 1 Depth map rendering:

The point sprite method is used to render the depth map. The idea is to use the geometry shader to render each fluid particle into a square. In the fragment shader, the depth is calculated for each pixel in the square and written into the depth texture.

Pass 2 thickness map rendering:

The purpose of rendering the thickness map is to record the thickness of the fluid in the Z direction of the camera, which is needed when calculating the color of the fluid. The thickness map is also rendered using the point sprite method. The difference is that you need to turn off the depth test, turn on alpha blending, set both the source color coefficient and the target color coefficient to 1, and then output the thickness value to the frame buffer, so that the thickness value will continue to accumulate.

Pass 3 depth map smoothing:

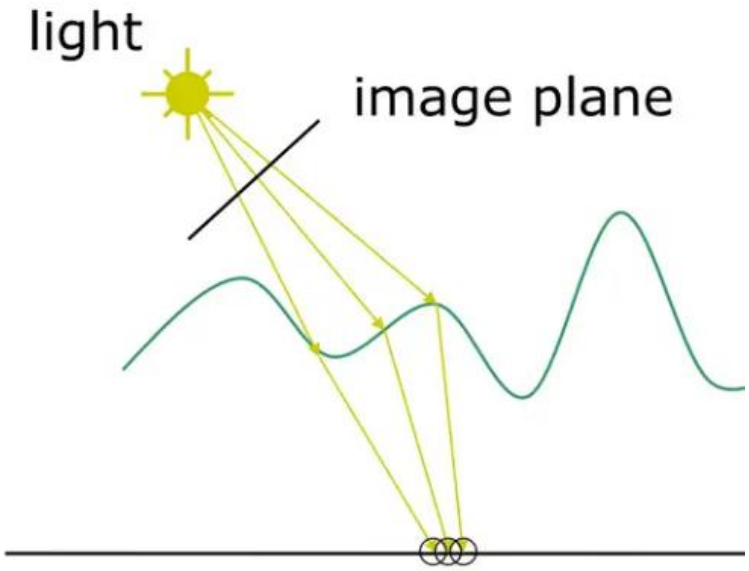
Bilateral filtering is used to smooth the depth map to eliminate the particle feeling and obtain a smooth fluid surface. The reason for using bilateral filtering is to avoid smoothing out the edges between particles that are too far apart.

Pass 4 Shadow rendering:

In order to enhance the reality of the fluid, it is necessary to render the soft shadow cast by the fluid on the floor. A real-time shadow mapping algorithm is used here, and the PCF (Percentage-Closer Filtering) algorithm is used to soften the shadow.

Pass 5 Caustic rendering:

The phenomenon of caustics is the unevenness of the water surface, which results in some bright spots caused by gathering some light rays together and hitting the bottom of the water. A better algorithm for simulating caustics in graphics is photon mapping. Based on the depth map obtained by shadow rendering, the fluid surface can be reconstructed, the refraction direction can be found, and the light intersection operation can be performed. If there is an intersection with the ground, the position of the point in the world coordinate system is recorded, and this position is the "photon". The process is shown below:



Then redraw these "photon" coordinates to the light-view in the form of points, obtain a "caustic map" and save it to obtain a caustic map.

Pass 6 render fluid:

First perform normal reconstruction, the process is as follows:

Back-project all depth values into 3D points according to the internal parameters of the camera, and then calculate the difference according to the coordinates of adjacent 3D points to obtain the tangent direction:

$$T_{u0,i,j} = p_{i+1,j} - p_{i,j}$$

$$T_{u1,i,j} = p_{i,j} - p_{i-1,j}$$

$$T_{v0,i,j} = p_{i,j+1} - p_{i,j}$$

$$T_{v1,i,j} = p_{i,j} - p_{i,j-1}$$

In the formula, p is the 3D point coordinate obtained by back-projection, and T is the forward difference and the backward difference in the direction of u or v. By comparing the difference in Z value between p and neighboring points, select the side with the smaller Z value difference.

After obtaining the two tangents u and v, the normal direction can be obtained by cross product:

$$n = Cross(T_{u,i,j}, T_{v,i,j})$$

After the normals are reconstructed, they can be rendered according to the direction of view and the position of the light source. Specifically, the reflection direction is obtained according to the law of reflection and the law of refraction:

$$w_{o,reflect} = reflect(w_i, n)$$

$$w_{o,refract} = refract(w_i, n)$$

After getting these two directions, go to the scene to find the corresponding color. Here it needs to perform a ray tracing operation.

$$L_{reflect} = RayTrace(p, w_{o,reflect})$$

$$L_{refract} = RayTrace(p, w_{o,refract})$$

In the formula, L is the color obtained by ray tracing operation.

Next, the two colors are mixed. The basis for mixing is Fresnel's law. Here, Schlic's approximate Fresnel coefficient method is used:

$$F = F_0 + (1 - F_0) * (1.0 - dot(w_i, n))^5$$

Mix two colors according to the Fresnel coefficient:

$$L = FL_{reflect} + (1 - F)L_{refract}$$

At this point the color mixture looks like the fluid has been formed, but the fluid is completely transparent. In reality, fluids are colored, and the transparency decreases as the thickness increases. Beer-Lambert's law describes this relationship:

$$A = max(e^{-kd}, A_{min})$$

In the formula, A is the transparency; k is a coefficient; d is the fluid thickness, obtained from the previous thickness diagram.

Modify the color value of the transmitted part:

$$L_{reflect} = A \cdot RayTrace(p, w_{o,reflect}) + (1 - A) \cdot L_{fluid}$$

In the formula, L is the color of the liquid itself.

Pass 7 – end:

After adding caustic rendering, the final fluid rendering effect is obtained. At this point, the fluid rendering is completed.

2.5.3.3. SSFR Code

Draw depth map (pass 1):

```
// draw depth map
mPointSpriteZValue->Use();
mPointSpriteZValue->SetMat4("view", mCamera.GetView());
mPointSpriteZValue->SetMat4("projection", mCamera.GetProjection());
mPointSpriteZValue->SetFloat("particalRadius", 0.01f);
mPointSpriteZValue->SetVec3("cameraUp", mCamera.GetUp());
mPointSpriteZValue->SetVec3("cameraRight", mCamera.GetRight());
mPointSpriteZValue->SetVec3("cameraFront", mCamera.GetFront());
glBindVertexArray(mVaoParticals);
glDrawArrays(GL_POINTS, 0, mParticalNum);
mPointSpriteZValue->UnUse();
```

```
float DepthToZ(float depth) {
    return - zFar * zNear / (zNear * depth + zFar * (1.0 - depth));
}

float ZToDepth(in float z) {
    z = abs(z);
    return zFar * (zNear - z) / (z * (zNear - zFar));
}

void main() {

    float dist = distance(fragPosition, particalCenter);
    if (dist > particalRadius) {
        discard;
    }

    float deltaDepthNorm = 2.0 * sqrt(0.5 * 0.5 - pow(texCoordQuad.x - 0.5, 2) - pow(texCoordQuad.y - 0.5, 2) + 1e-5);
    float ZValue = DepthToZ(gl_FragCoord.z);
    ZValue += particalRadius * deltaDepthNorm;

    gl_FragDepth = ZToDepth(ZValue);

    fragColor = vec4(ZValue, 0.0, 0.0, 0.0);
    return;
}
```

Calculate and generate depth map through Z axis.

Blur depth map:

```
// blur depth map
GLuint bufferA = mTexZBuffer;
GLuint bufferB = mTexZBlurTempBuffer;
mDepthFilter->Filter(bufferA, bufferB, glm::ivec2(mWindowWidth, mWindowHeight));
```

Draw thickness map (pass 2):

```
// draw thickness map
glBindFramebuffer(GL_FRAMEBUFFER, mFboThickness);
glViewport(0, 0, mWindowWidth, mWindowHeight);
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glEnable(GL_DEPTH_TEST);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glDisable(GL_DEPTH_TEST);
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ONE);
mPointSpriteThickness->Use();
mPointSpriteThickness->SetMat4("view", mCamera.GetView());
mPointSpriteThickness->SetMat4("projection", mCamera.GetProjection());
mPointSpriteThickness->SetFloat("particalRadius", 0.01f);
mPointSpriteThickness->SetVec3("cameraUp", mCamera.GetUp());
mPointSpriteThickness->SetVec3("cameraRight", mCamera.GetRight());
mPointSpriteThickness->SetVec3("cameraFront", mCamera.GetFront());
glBindVertexArray(mVaoParticals);
glDrawArrays(GL_POINTS, 0, mParticalNum);
mPointSpriteThickness->UnUse();
glDisable(GL_BLEND);
glEnable(GL_DEPTH_TEST);
```

```
void main() {

    float dist = distance(fragPosition, particalCenter);
    if (dist > particalRadius) {
        discard;
    }

    float deltaDepthNorm = 2.0 * sqrt(0.5 * 0.5 - pow(texCoordQuad.x - 0.5, 2) - pow(texCoordQuad.y - 0.5, 2) + 1e-5);
    float thickness = 2.0 * particalRadius * deltaDepthNorm;

    fragColor = vec4(thickness, 0.0, 0.0, 0.0);
    return;
}
```

Draw the floor (pass 7):

```
// draw the floor
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, mShadowMap->GetShadowMap());
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, mShadowMap->GetCausticMap());
glActiveTexture(GL_TEXTURE2);
glBindTexture(GL_TEXTURE_CUBE_MAP, mSkyBox->GetId());
glActiveTexture(GL_TEXTURE3);
glBindTexture(GL_TEXTURE_2D, mSlabWhite->mTexAlbedo);
glActiveTexture(GL_TEXTURE4);
glBindTexture(GL_TEXTURE_2D, mSlabWhite->mTexRoughness);
mDrawModel->Use();
mDrawModel->SetMat4("model", floorModel);
mDrawModel->SetMat4("view", mCamera.GetView());
mDrawModel->SetMat4("projection", mCamera.GetProjection());
mDrawModel->SetMat4("lightView", mShadowMap->mLightView);
mDrawModel->SetMat4("lightProjection", mShadowMap->mLightProjection);
glBindVertexArray(mVaoFloor);
glDrawArrays(GL_TRIANGLES, 0, 6);
glBindVertexArray(0);
mDrawModel->UnUse();
```

```
vec3 ShadeFloorWithShadow(vec3 originColor, vec3 lightPos, vec3 curPosition) {
    // Project to the light source and get the texture coordinates
    vec4 fragNDC = lightProjection * lightView * vec4(curPosition, 1.0);
    fragNDC /= fragNDC.w;
    vec2 texCoord = NdcToTexCoord(fragNDC.xy);

    // PCF method to calculate shadows
    float fragDist = distance(lightPos, curPosition);
    float shadowFactor = 0.2 * Pcf(texCoord, fragDist);
    vec3 colorWithShadow = mix(originColor, shadowColor, shadowFactor);

    // Add caustics
    vec3 caustic = texture(causticMap, texCoord).xyz;

    return colorWithShadow + caustic;
}
```

Draw fluid (pass 6):

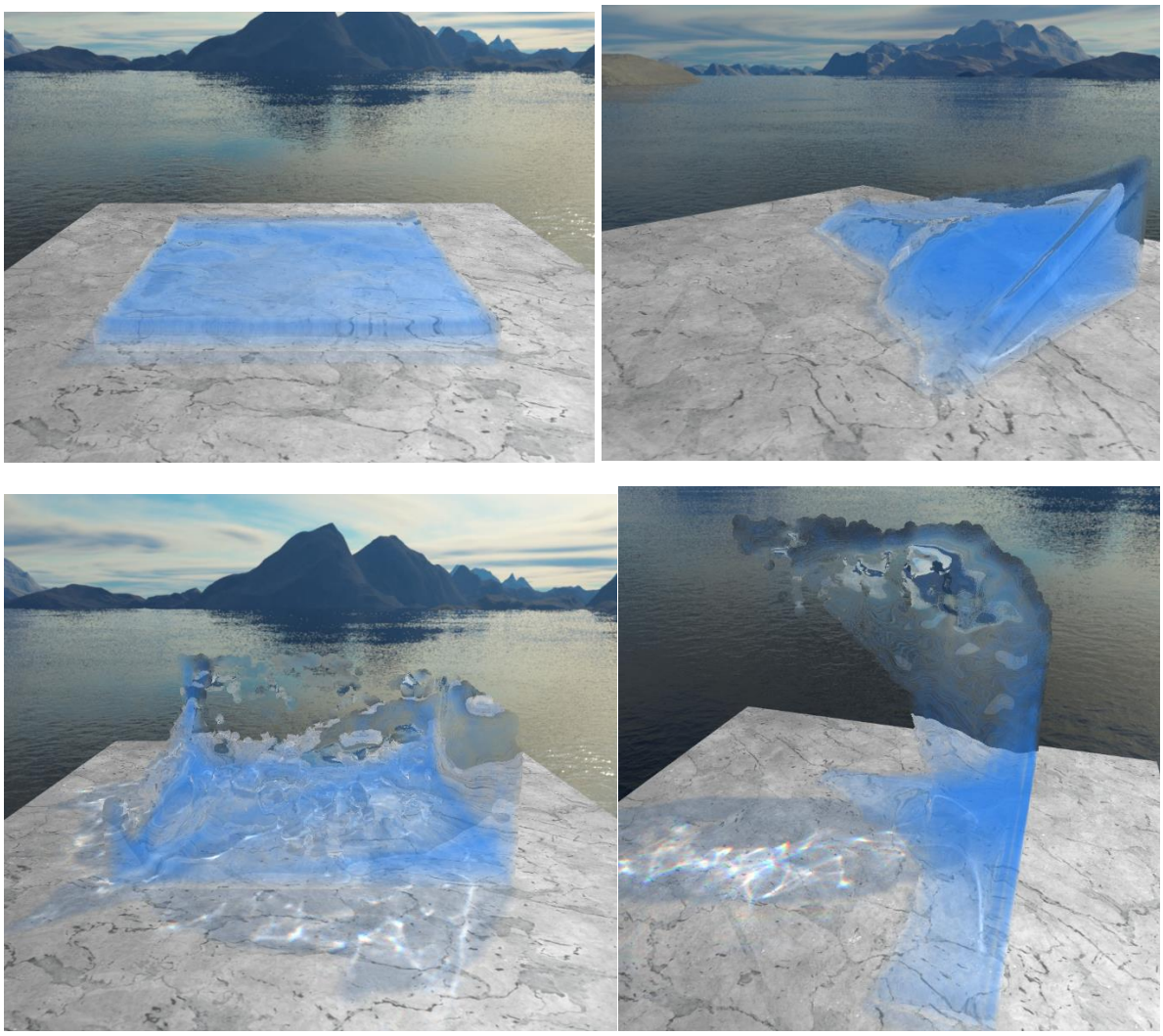
```
// draw fluid
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_CUBE_MAP, mSkyBox->GetId());
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, mShadowMap->GetShadowMap());
glActiveTexture(GL_TEXTURE2);
glBindTexture(GL_TEXTURE_2D, mShadowMap->GetCausticMap());
glActiveTexture(GL_TEXTURE3);
glBindTexture(GL_TEXTURE_2D, mSlabWhite->mTexAlbedo);
glActiveTexture(GL_TEXTURE4);
glBindTexture(GL_TEXTURE_2D, mSlabWhite->mTexRoughness);
glBindImageTexture(0, bufferB, 0, GL_FALSE, 0, GL_READ_ONLY, GL_R32F);
glBindImageTexture(1, mTexThicknessBuffer, 0, GL_FALSE, 0, GL_READ_ONLY, GL_R32F);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, mBufferFloor);
mDrawFluidColor->Use();
mDrawFluidColor->SetMat4("camToWorldRot", glm::transpose(mCamera.GetView()));
mDrawFluidColor->SetMat4("camToWorld", glm::inverse(mCamera.GetView()));
mDrawFluidColor->SetMat4("model", floorModel);
mDrawFluidColor->SetMat4("projection", mCamera.GetProjection());
mDrawFluidColor->SetMat4("lightView", mShadowMap->mLightView);
mDrawFluidColor->SetMat4("lightProjection", mShadowMap->mLightProjection);
glBindVertexArray(mVaoParticals);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
glBindVertexArray(0);
mDrawFluidColor->UnUse();

mSkyBox->Draw(mWindow, mVaoNull, mCamera.GetView(), mCamera.GetProjection());
```

```
void main()
{
    ivec2 curPixelId = ivec2(gl_FragCoord.xy);
    float curDepth = imageLoad(zBuffer, curPixelId).x;
    if (curDepth > 0.0) {
        discard;
    }

    // write depth
    gl_FragDepth = ZToDepth(curDepth);

    vec4 camearOrigin = camToWorld * vec4(0.0, 0.0, 0.0, 1.0);

    // Calculate position
    vec3 curPos = Reproject(curDepth, curPixelId);
    vec4 curPoseOnWorld = camToWorld * vec4(curPos, 1.0);

    // Compute various vectors
    vec4 normal = vec4(CalculateNormal(curPixelId, curDepth, curPos), 1.0);
    vec4 normalOnWorld = camToWorldRot * normal;
    vec3 wiOnCamera = imageCoordToWi(cameraIntrinsic, curPixelId);
    vec4 wiOnWorld = camToWorldRot * vec4(normalize(wiOnCamera), 1.0);
    vec3 woReflect = reflect(wiOnWorld.xyz, normalOnWorld.xyz);
    vec3 woRefract = refract(wiOnWorld.xyz, normalOnWorld.xyz, eta);
    vec3 fresnel = FresnelSchlic(wiOnWorld.xyz, normalOnWorld.xyz);

    mat4 worldToModel = inverse(model);
```

```
    // refracted color
    Ray refractRay;     // In the local coordinate system of the model
    refractRay.origin = (worldToModel * curPoseOnWorld).xyz;
    refractRay.direction = mat3(worldToModel) * woRefract;
    vec3 refractColor = GetRayTraceColor(refractRay, curPoseOnWorld.xyz, camearOrigin.xyz);

    float thickness = imageLoad(thicknessBuffer, curPixelId).x;
    float transparentFactor = TransparentFactor(thickness * 2.0);
    refractColor = transparentFactor * refractColor + (1.0 - transparentFactor) * fluidColor;

    // Reflection color
    Ray reflectRay;
    reflectRay.origin = (worldToModel * curPoseOnWorld).xyz;
    reflectRay.direction = mat3(worldToModel) * woReflect;
    vec3 reflectColor = GetRayTraceColor(reflectRay, curPoseOnWorld.xyz, camearOrigin.xyz);

    // mix
    vec3 outColor = mix(refractColor, reflectColor, fresnel);

    FragColor = vec4(outColor, 1.0);
}
```

After the SSFR rendering is completed, the fluid can be rendered correctly.

The final rendering effect:



2.5.4. User Manual

This project contains two independent project folders. “3DFluid” includes python files which is algorithm quick simulation project. “FluidSimulation” includes C++ project which is the 3D fluid rendering engine.

3DFluid:

Taichi Lang simulation for PBF algorithm and SPH algorithm is compiled using python environment. Make sure you have python 3.8 or above running before use.

First install the dependencies:

```
pip install requirements.txt
```

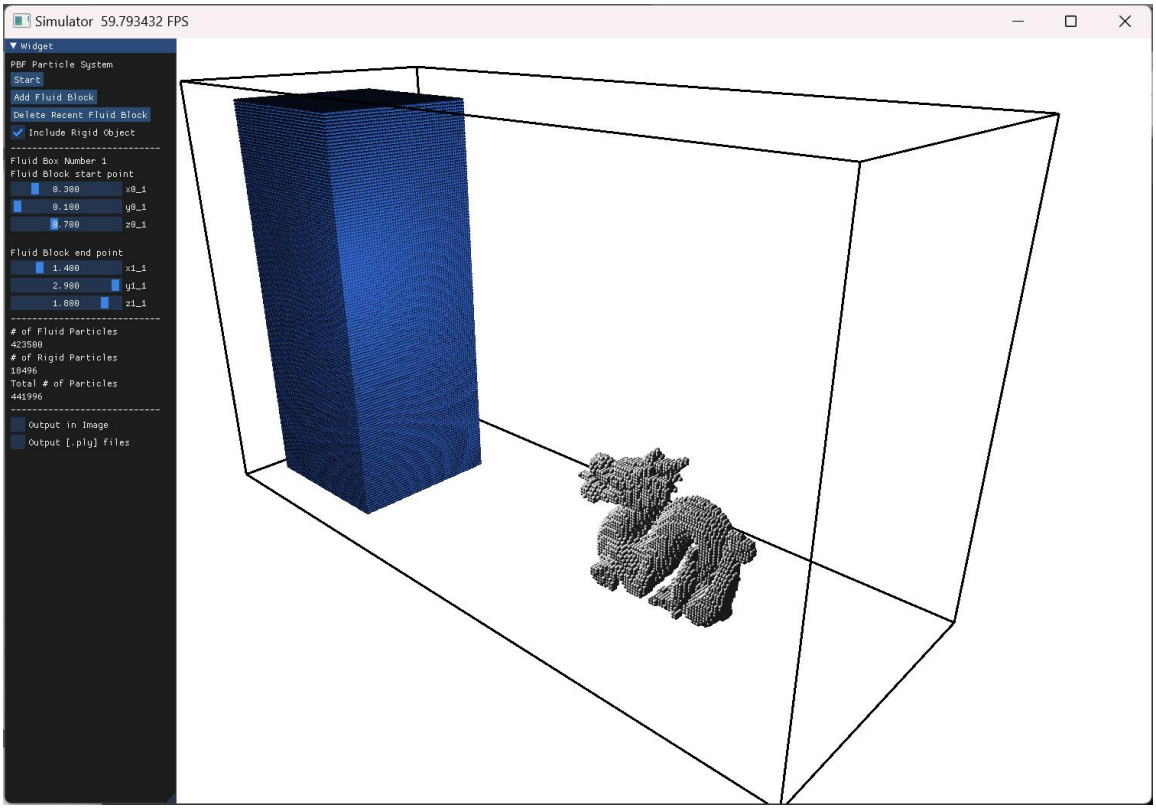
Secondly, use the following command to run the PBF algorithm simulation:

```
python main.py
```

Use the following command to run the SPH algorithm simulation:

python run_simulation.py

The running window will be like:

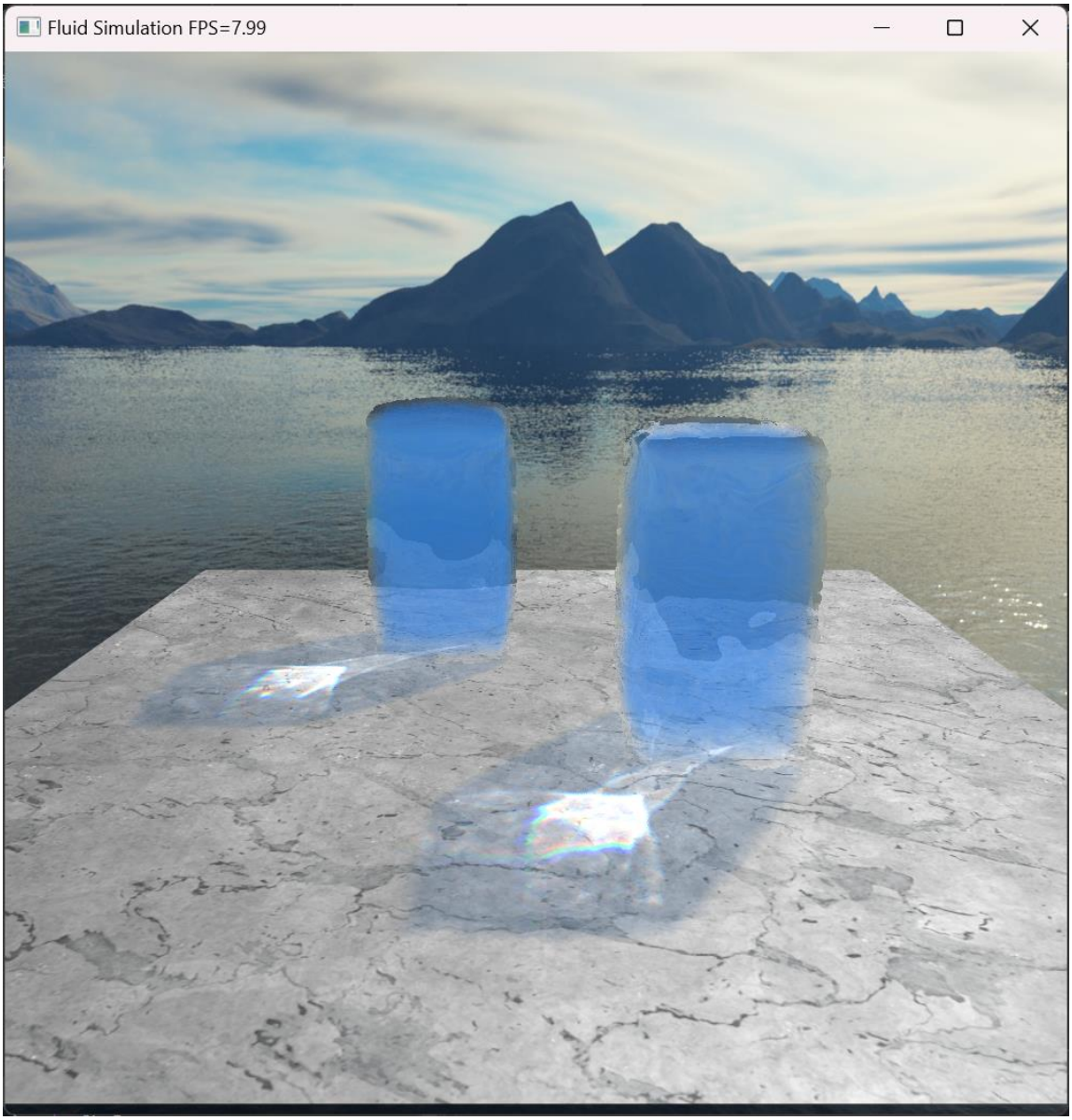


The parameters set up panel will be at life of the window. The value can be set by drag the scroll bar.

FluidSimulation:

A C++ project for 3D fluid rendering engine. Use Visual Studio 2022 to open the solution file located in "FluidSimulation\build\FluidSimulation.sln". Run the code by pressing ctrl+F5.

The running window will be like:



The program will automatically run the simulation unless to be paused. The user key binding as follow:

Pause:	[space]
Exit:	[esc]
Camera rotation:	[mouse left button click]
Camera displacement:	[mouse right button click]
Camera zoom:	[mouse wheel]
Drag fluid:	[mouse middle button click]

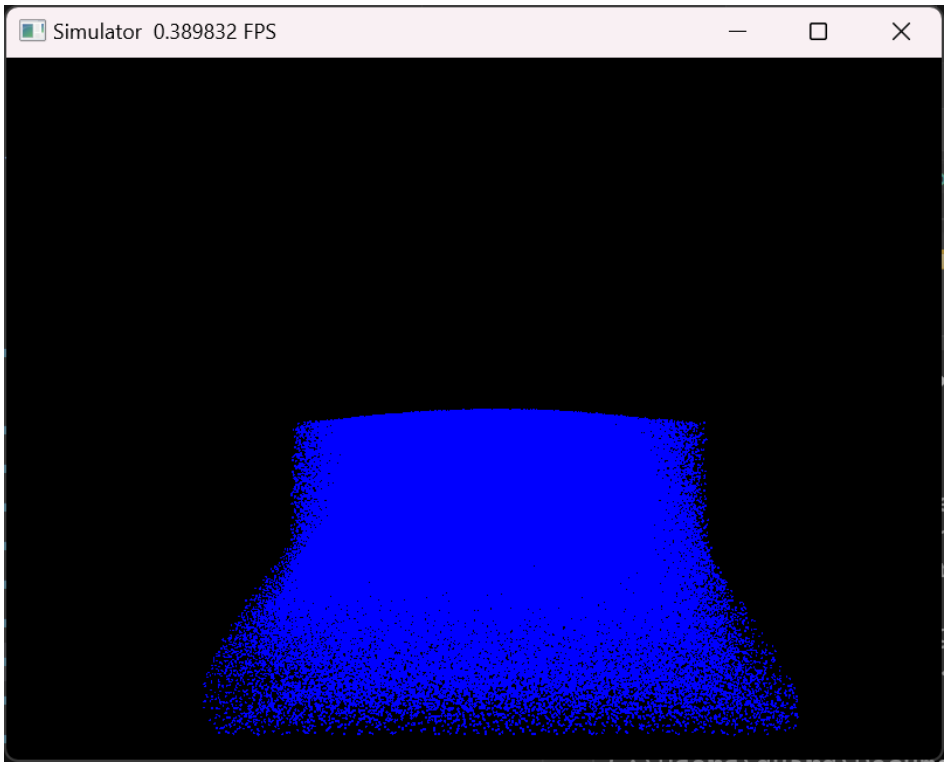
2.6. Testing Details and Results

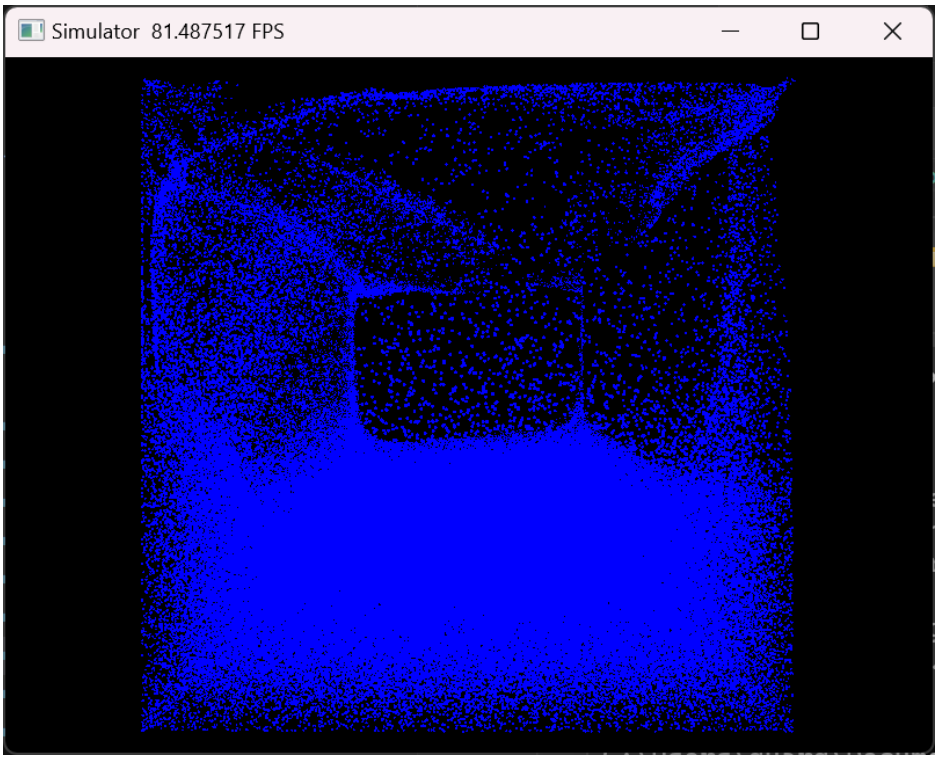
2.6.1. Test 1: Real-time performance of PBF algorithm

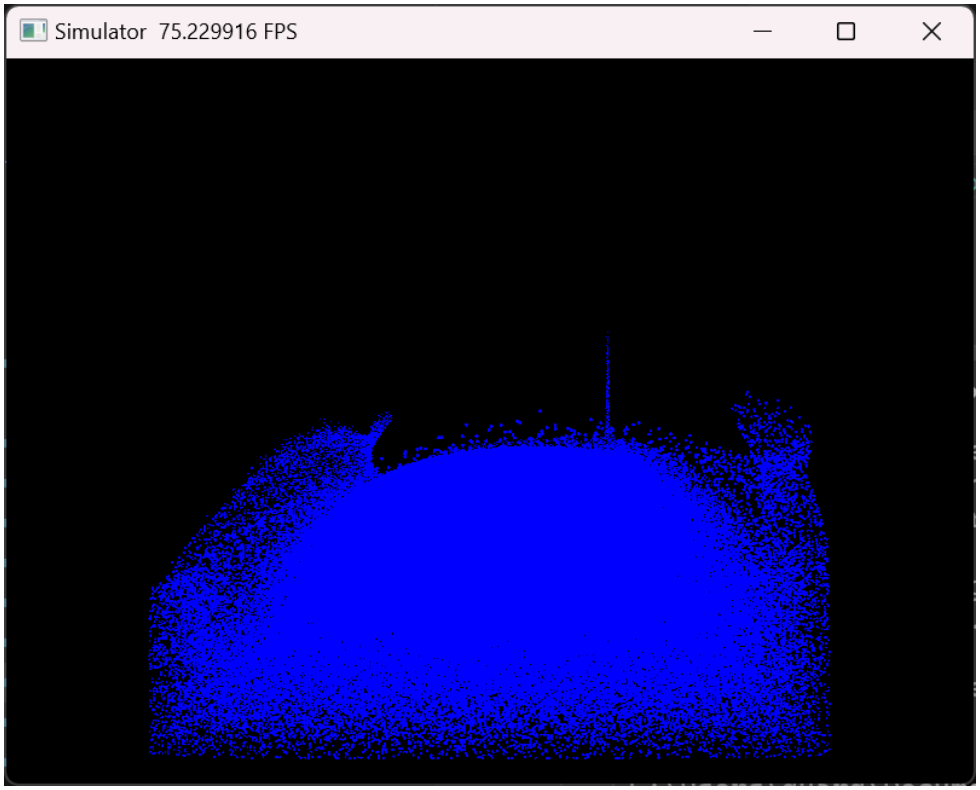
TEST 1	REAL-TIME PERFORMANCE OF PBF ALGORITHM
VERIFICATION	Use Taichi Lang to implement the PBF algorithm, simulate and visualize a 100,000-particle scene, and judge whether the efficiency of the algorithm meets real-time computing standards through the frame rate
PASS CONDITION	1. Algorithms can be implemented on GPU. 2. The algorithm can simulate 100,000 particles. 3. The scene can reach an average of more than 60fps when running.
FAIL CONDITION	1. Algorithm cannot be run in parallel on GPU. 2. The algorithm cannot simulate and visualize 100,000 particles. 3. The scene cannot reach above 60fps on average when running.

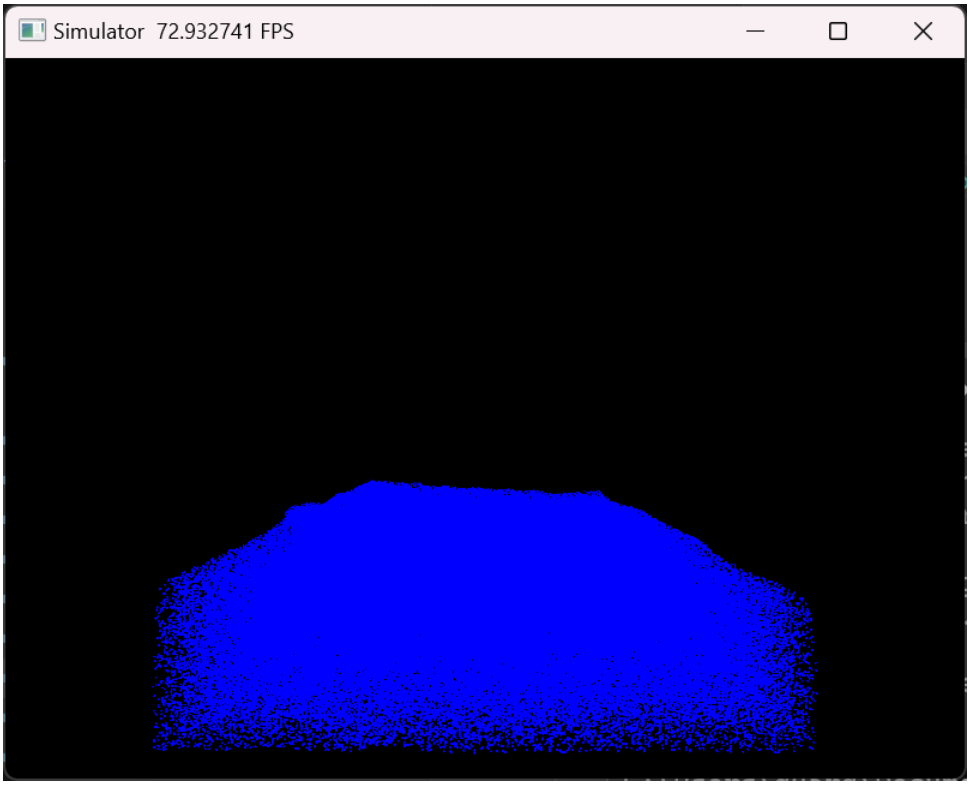
Test 1 result: Pass

The PBF algorithm is implemented through Taichi Lang's architecture and uses GPU parallel computing. The simulation screenshot is as follows:




Simulator 81.487517 FPS


Simulator 75.229916 FPS



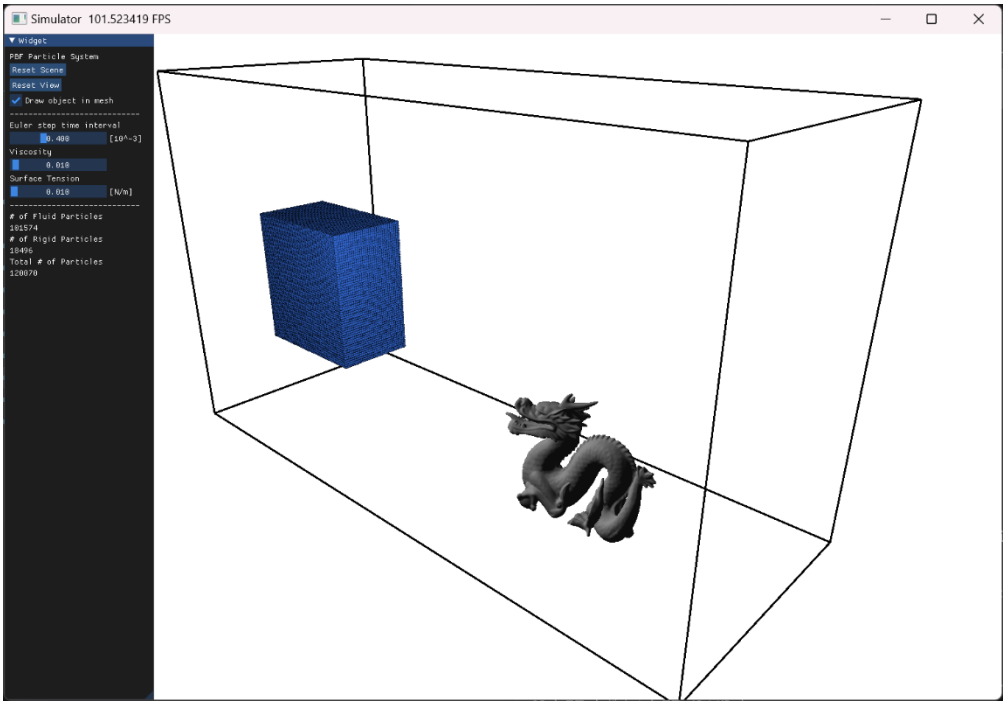
The simulation uses 50*50*40=100000 particles. Running average fps reached 74.62 in 10 seconds.

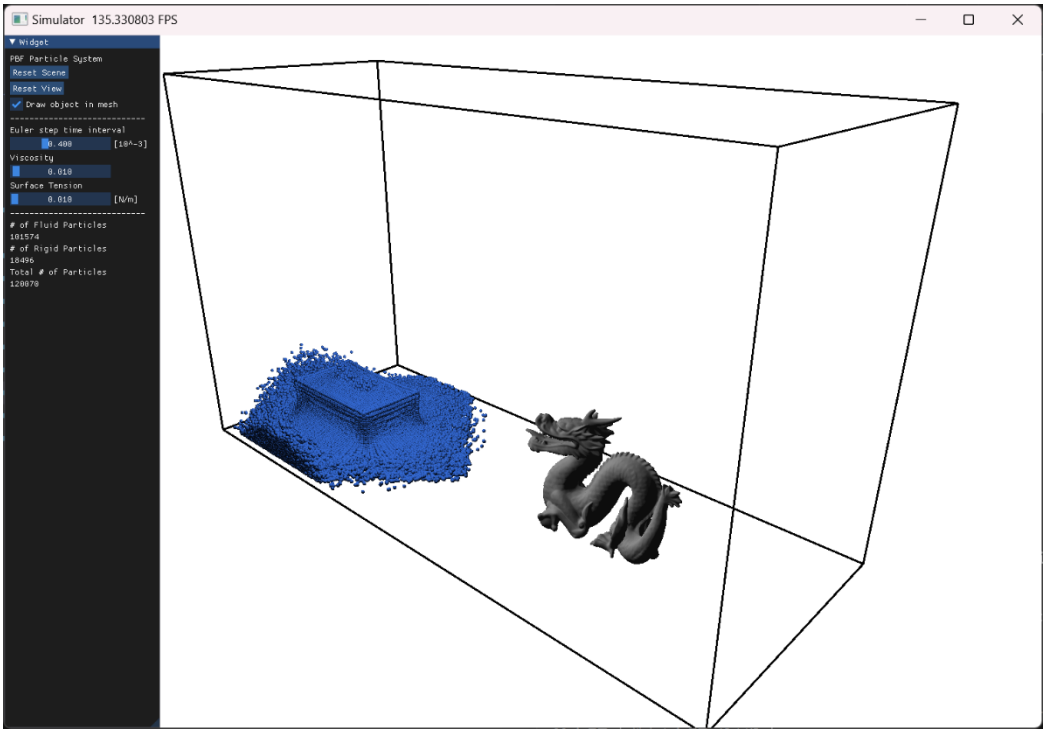
2.6.2. Test 2: Real-time performance of SPH algorithm

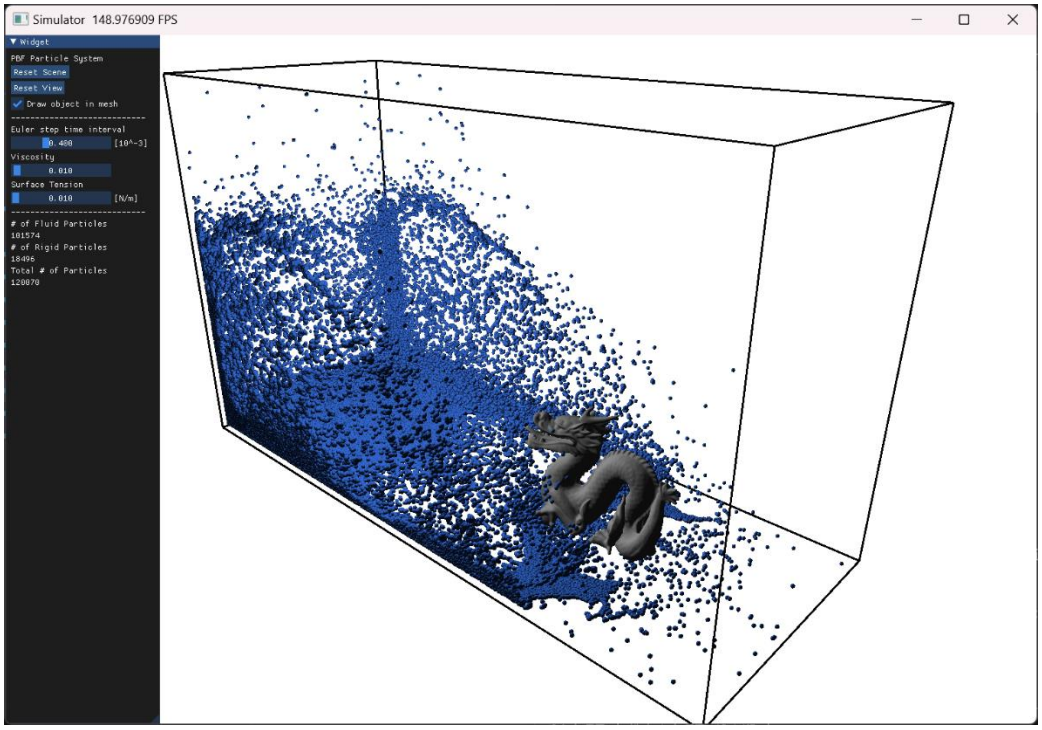
TEST 2	REAL-TIME PERFORMANCE OF SPH ALGORITHM
VERIFICATION	Use Taichi Lang to implement the SPH algorithm, simulate and visualize a 100,000-particle scene, and judge whether the efficiency of the algorithm meets real-time computing standards through the frame rate.
PASS CONDITION	1. Algorithms can be implemented on GPU. 2. The algorithm can simulate 100,000 particles. 3. The scene can reach an average of more than 60fps when running.
FAIL CONDITION	1. Algorithm cannot be run in parallel on GPU. 2. The algorithm cannot simulate and visualize 100,000 particles. 3. The scene cannot reach above 60fps on average when running.

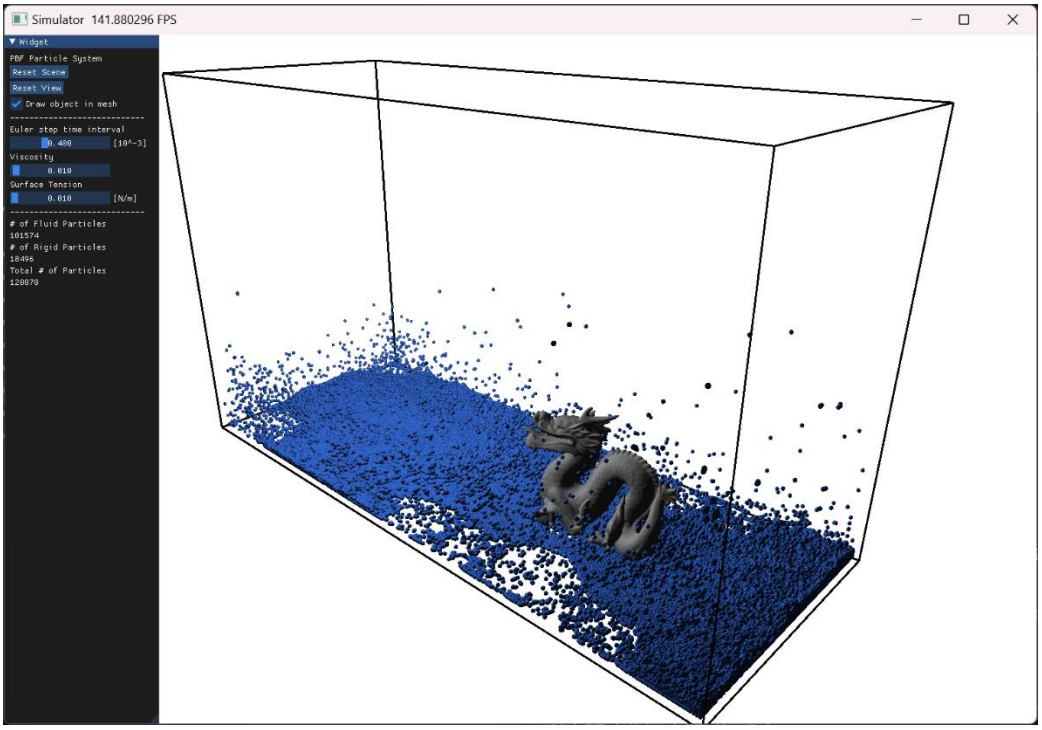
Test 2 result: Pass

The SPH algorithm has multiple implementation methods, and the WCSPH algorithm is chosen here. Use Taichi Lang to help implement the WCSPH algorithm in parallel in the GPU. The simulation screenshot is as follows:


Simulator 101.523419 FPS
Widget
PBF Particle System
Reset Scene
Reset View
Draw object in mesh
Euler step time interval
0.400
[10^-3]
Viscosity
0.010
Surface Tension
0.010
[N/m]
# of Fluid Particles
101574
# of Rigid Particles
18496
Total # of Particles
120070


Simulator 135.330803 FPS
Widget
PBF Particle System
Reset Scene
Reset View
Draw object in mesh
Euler step time interval
0.400
[10^-3]
Viscosity
0.010
Surface Tension
0.010
[N/m]
# of Fluid Particles
101574
# of Rigid Particles
18496
Total # of Particles
120070





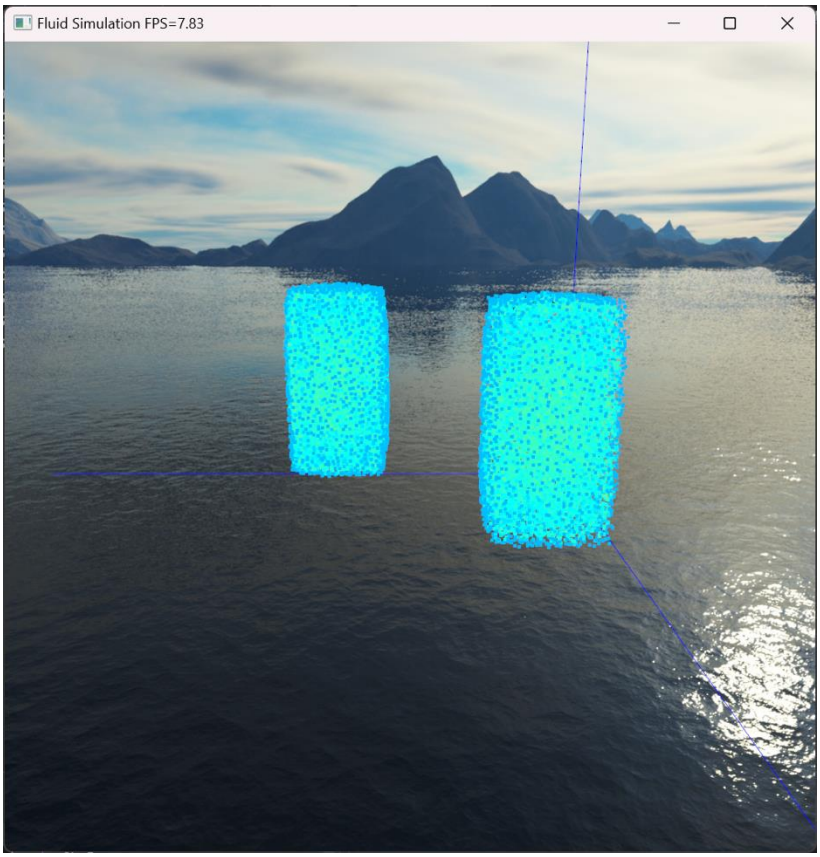
This simulation uses 101574 particles for fluid simulation. The average frame rate can reach 141.27 fps in ten seconds. SPH algorithm simulation can reach real-time simulation standards.

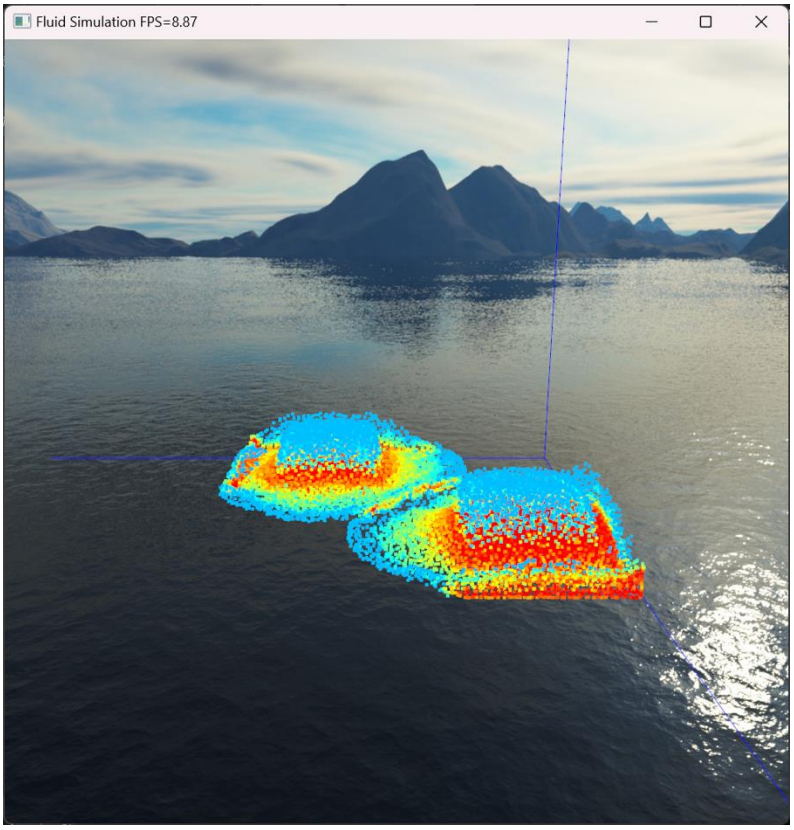
2.6.3. Test 3: Run WCSPH algorithm using OpenGL API

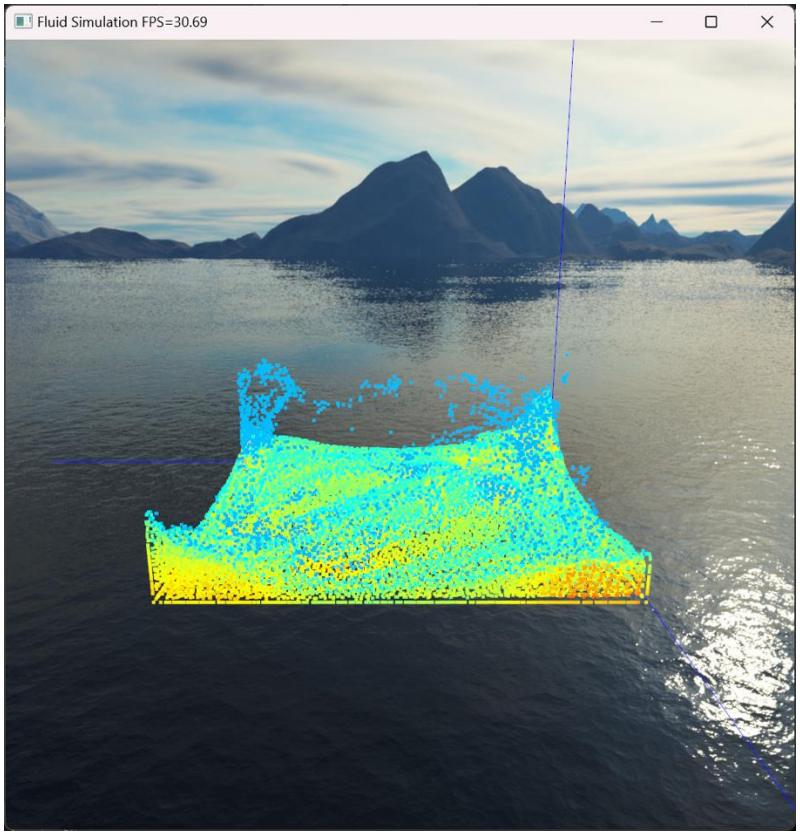
TEST 3	RUN SPH ALGORITHM USING OPENGL API
VERIFICATION	Based on the C++ architecture, the OpenGL API is called, the particle information parameters are passed into the GPU, and the GPU is used to perform parallel calculations on particle motion and update particle information. The process should be within an acceptable time frame.
PASS CONDITION	1. Particle information can be passed into the GPU. 2. Particle information updates can be correctly processed by the WCSPH algorithm. 3. The update time of particle information can reach real-time computing standards.
FAIL CONDITION	1. Particle information cannot be correctly transferred to the GPU, or the OpenGL API call fails. 2. Particle information updates are not correctly processed by the WCSPH algorithm. 3. The update time of particle information does not meet the real-time computing standard.

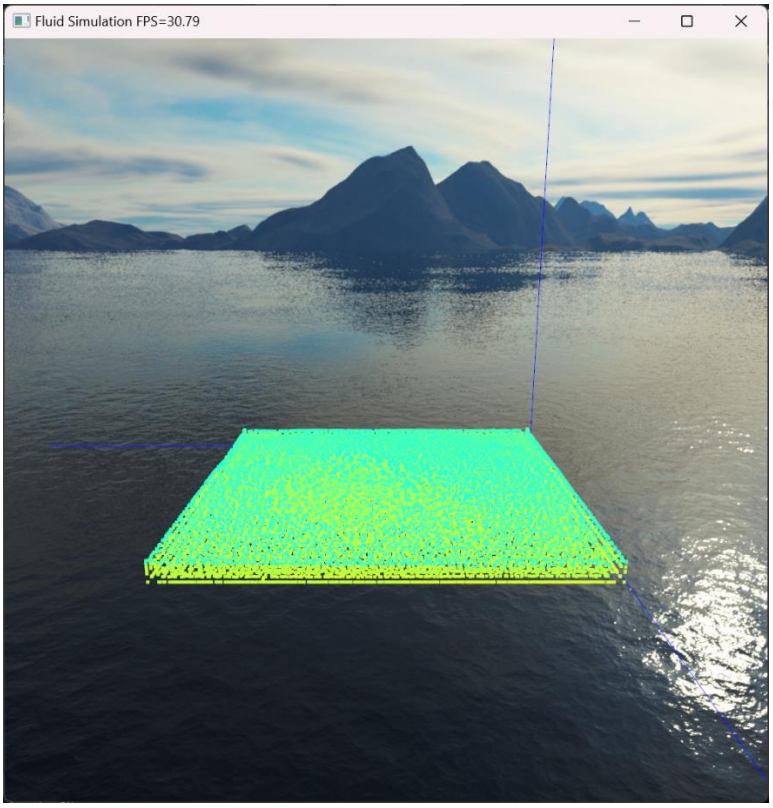
Test 3 result: Pass

Use the OpenGL API to load ComputePartical.comp and pass the particle position variables. Use the GPU to perform parallel operations and return after updating the particle position. Draw the picture in the form of particles and display the results visually. The screenshot is as follows:


Fluid Simulation FPS=7.83


Fluid Simulation FPS=8.87


Fluid Simulation FPS=30.69

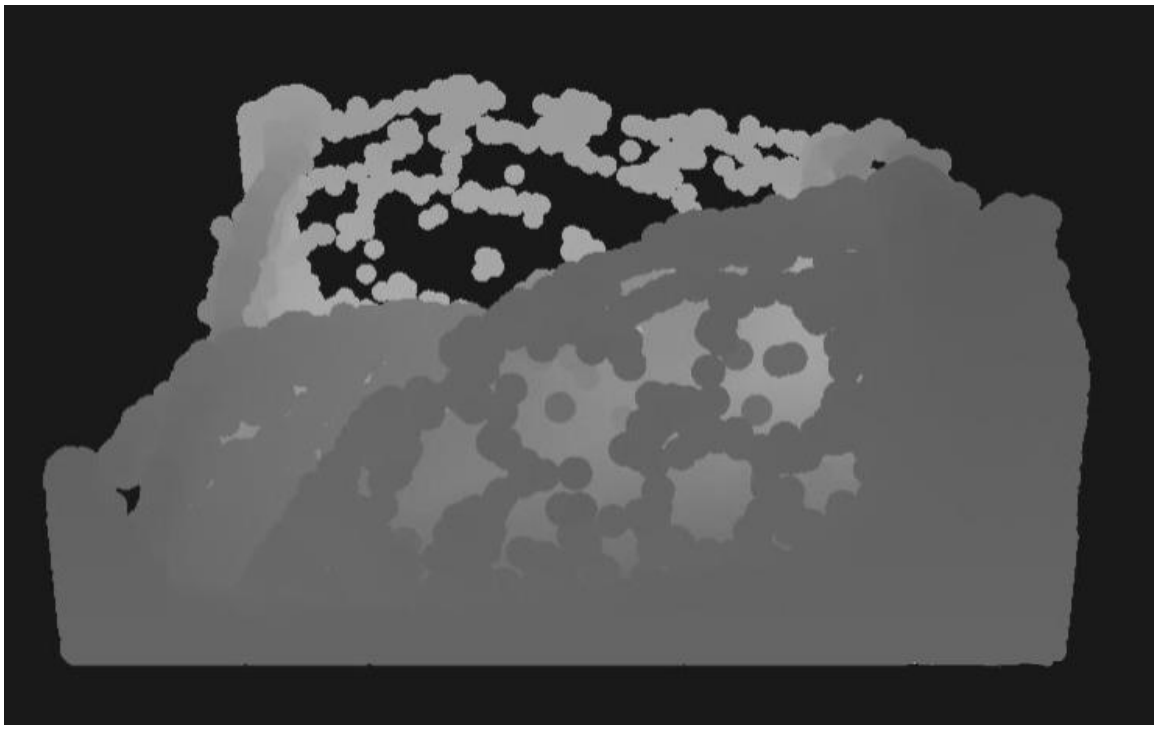

Fluid Simulation FPS=30.79

2.6.4. Test 4: Compute depth and thickness maps

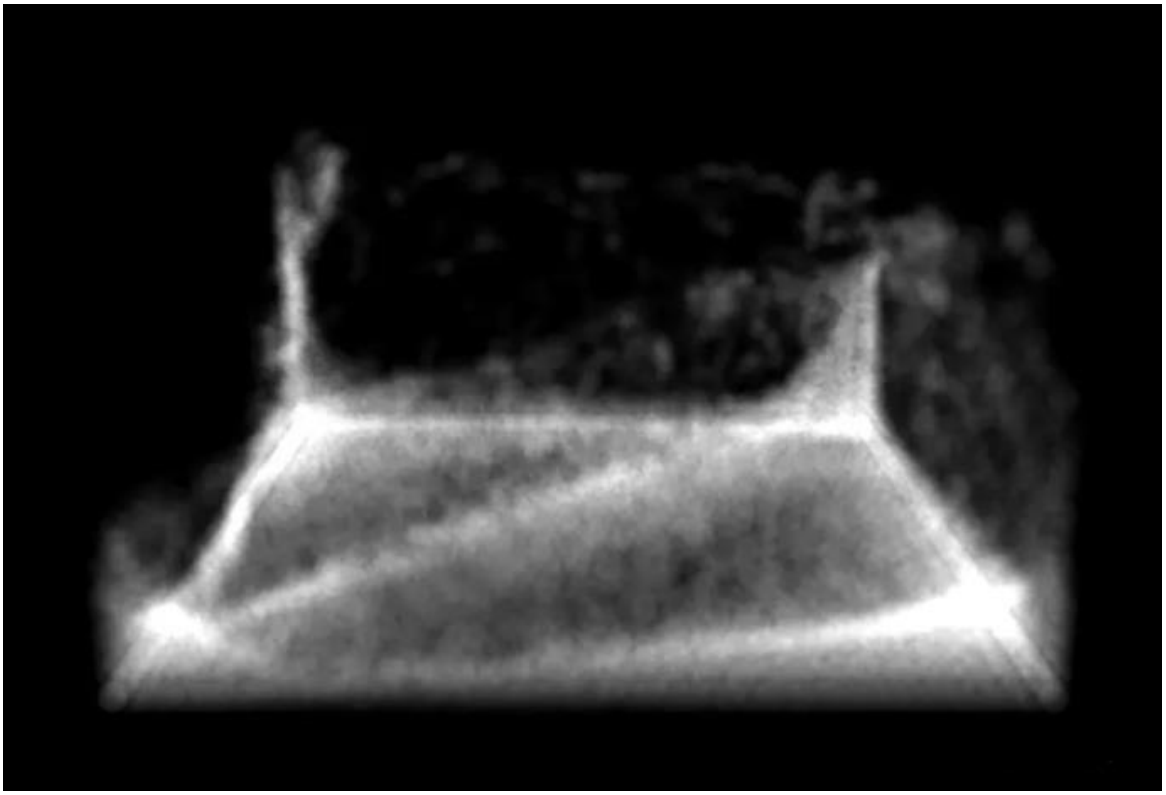
TEST 4	COMPUTE FLUID PARTICLE DEPTH AND THICKNESS MAPS
VERIFICATION	In order to implement the SSFR algorithm, the depth map and thickness map need to be calculated in real time based on particle information.
PASS CONDITION	1. Depth maps can be calculated in real time based on particle information. 2. Thickness maps can be calculated in real time based on particle information.
FAIL CONDITION	1. Depth maps cannot be calculated in real time based on particle information. 2. Thickness maps cannot be calculated in real time based on particle information.

Test 4 result: Pass

Render the depth map based on the particle information, and draw a screenshot of the resulting depth map:



Render the thickness map based on the particle information, and take a screenshot of the resulting thickness map:



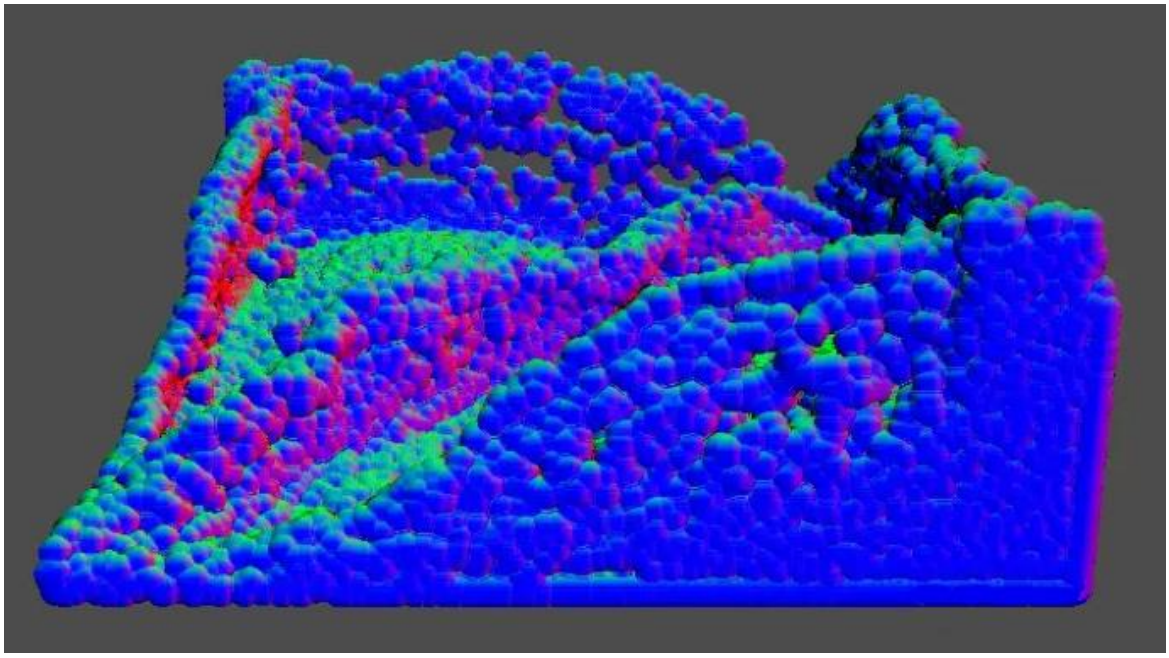
2.6.5. Test 5: Depth map smoothing

TEST 5	DEPTH MAP SMOOTHING
VERIFICATION	The depth map needs to be smoothed to remove the graininess. Smoothing is done using a suitable filtering algorithm. Completing the smoothed depth map should result in a smooth fluid surface after normal reconstruction.
PASS CONDITION	1. Smoothing algorithms can process depth maps in real time. 2. The smoothed rendering should match real fluids.
FAIL CONDITION	1. Smoothing algorithms cannot process depth maps in real time. 2. The smoothed rendering does not match real fluids. For example, there is a sense of particles, artifacts appear on the edges, etc.

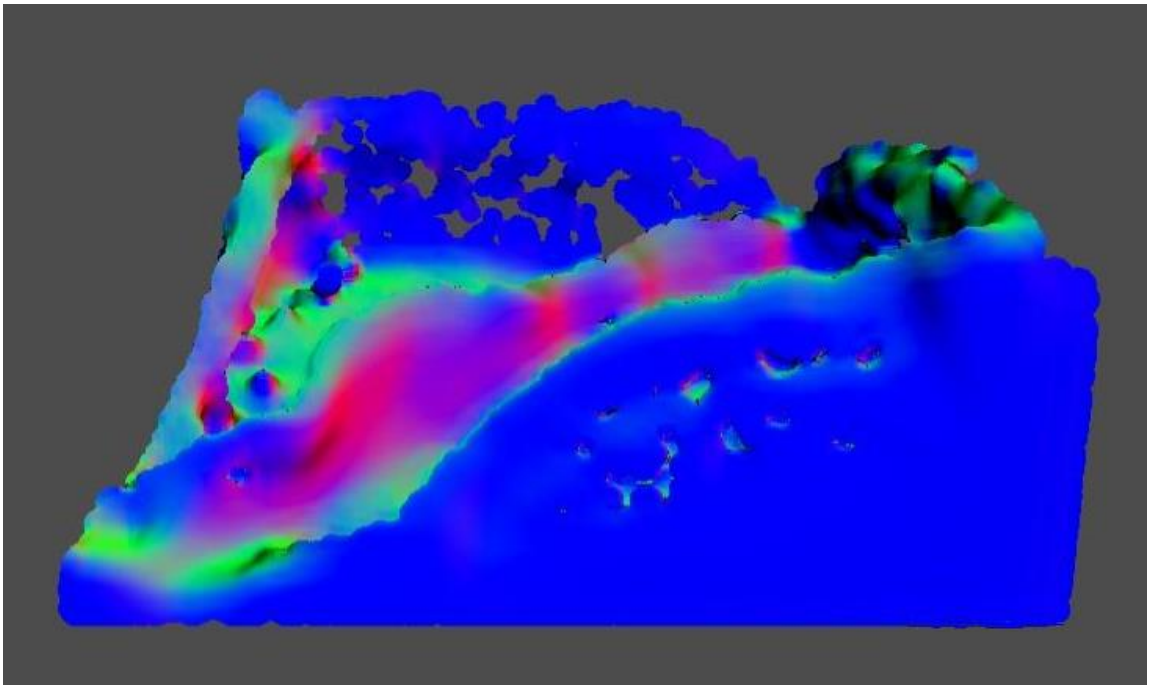
Test 5 result: Pass

The A-Trous Wavelet algorithm is used for smoothing. Smoothing can be done in real time. The smoothed depth map enables smooth fluid surfaces in subsequent renderings. In order to demonstrate the results, here is a comparison between the rendering results of the depth map without smoothing after normal reconstruction and the rendering result of the smoothed depth map after normal reconstruction.

Without smoothing:



With smoothing:



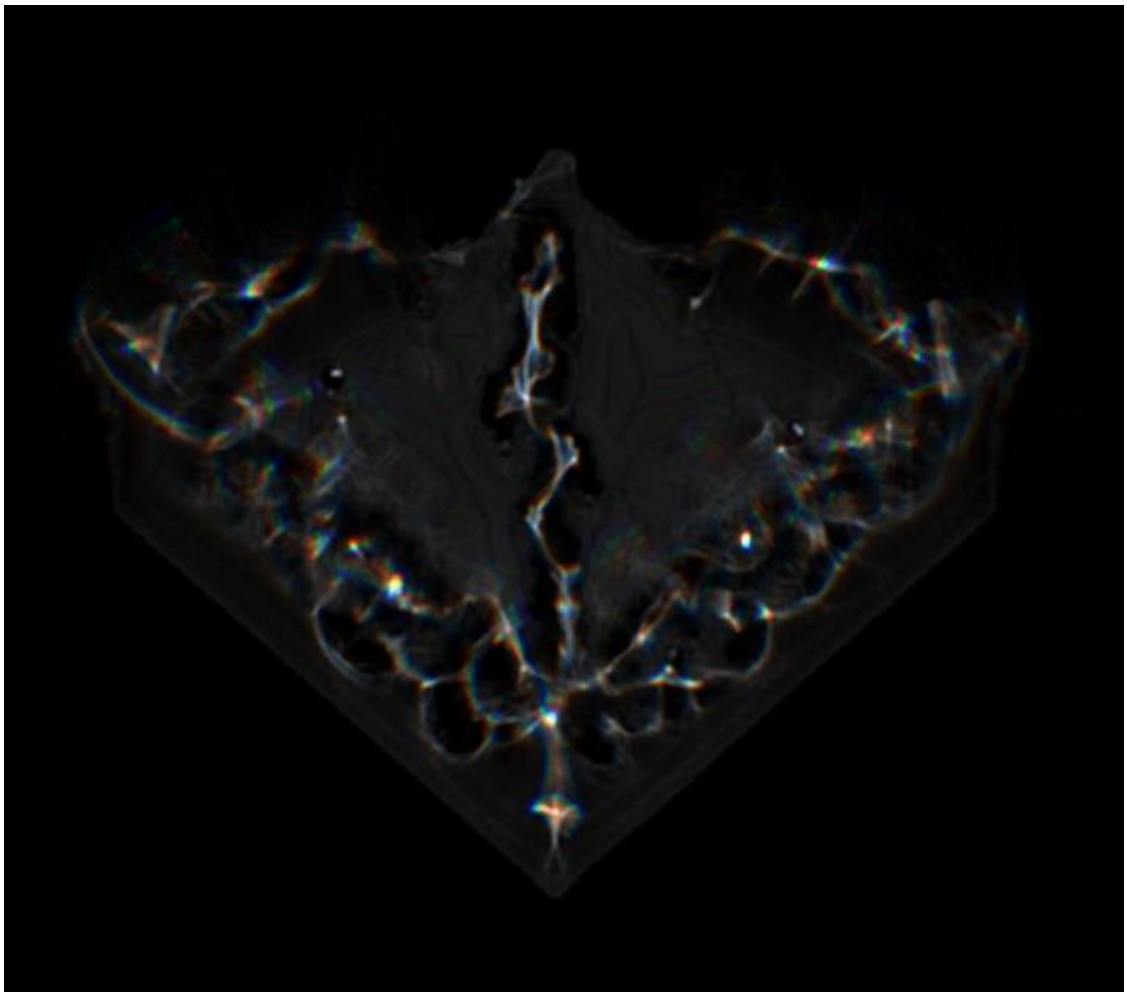
2.6.6. Test 6: Compute caustic map

TEST 6	COMPUTE CAUSTIC MAP
VERIFICATION	Use photon mapping to calculate the correct causal map to simulate the image of light reflected on the ground after being refracted by the water surface.
PASS CONDITION	Caustic map is generated correctly.

FAIL CONDITION	Caustic map is generated incorrectly. Ray casting effects are not physically correct.

Test 6 result: Pass

The caustic map reconstructs the fluid surface through the previously generated shadow map and then calculates the photon mapping algorithm. The calculated photon coordinates are plotted in the form of points to obtain a causal map. The resulting causal map is shown as follow:



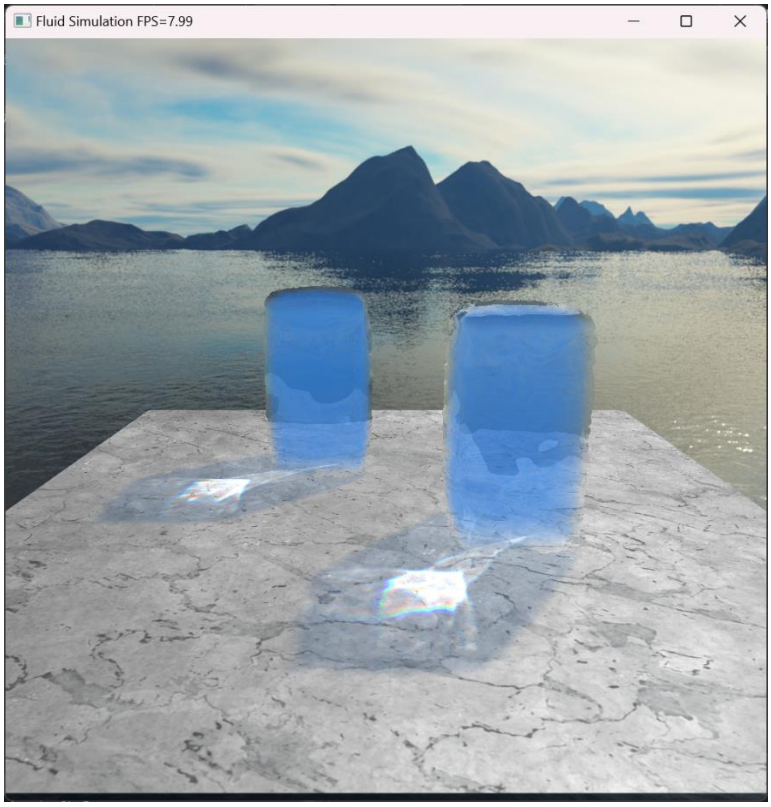
2.6.7. Test 7: SSFR rendering process implementation

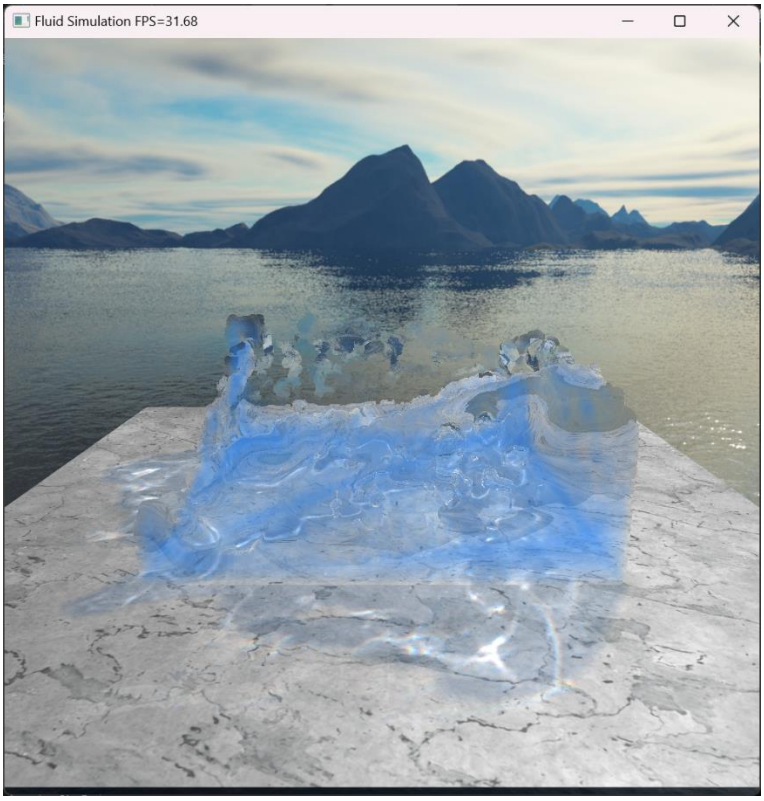
TEST 7	**SSFR RENDERING PROCESS IMPLEMENTATION**
VERIFICATION	The entire process of the SSFR algorithm should be implemented correctly. Rendering results should match real fluids. The rendering process should be completed in real time.
PASS CONDITION	1. The rendering pipeline works correctly. 2. The rendering effect is close to reality. 3. Rendering runs in real time.

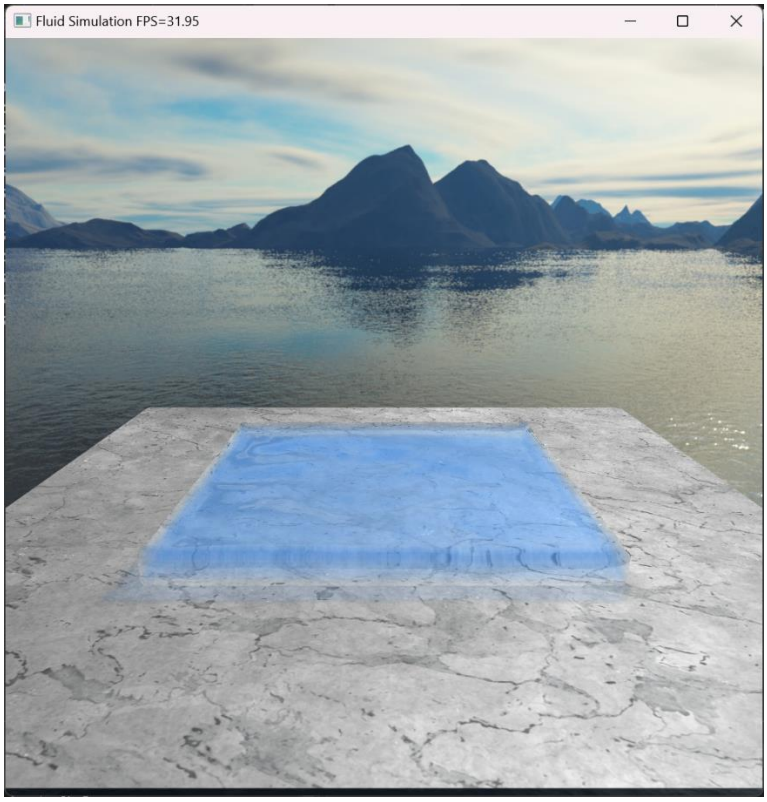
FAIL CONDITION	1. The rendering pipeline is not working correctly. An abnormality occurred in some modules of the SSFR algorithm. 2. The rendering is not close to reality. Fluids do not have smooth surfaces. Fluid shadow or lighting effects are not physically correct. 3. Rendering cannot run in real time.

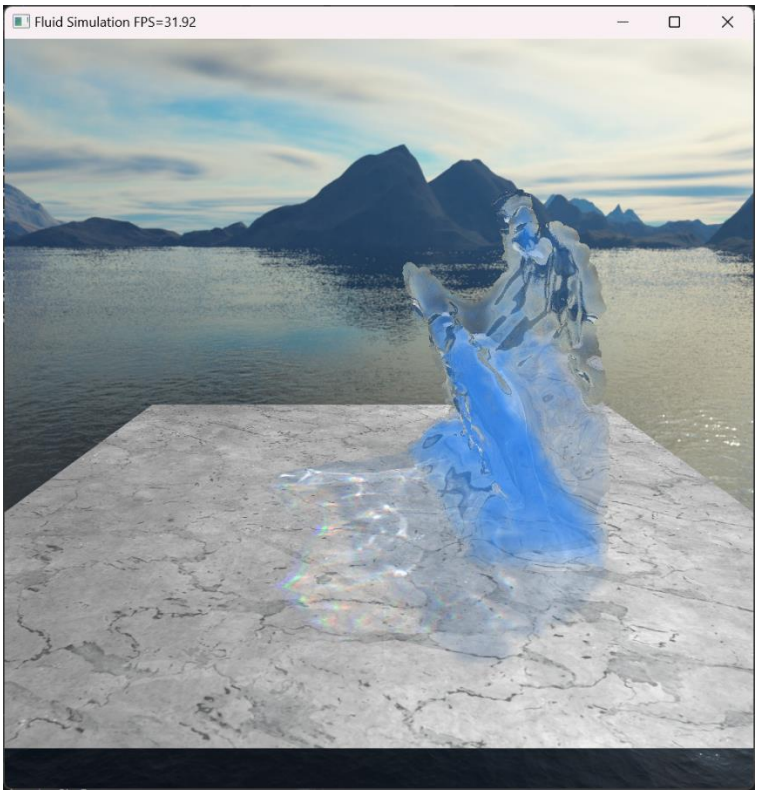
Test 7 result: Pass

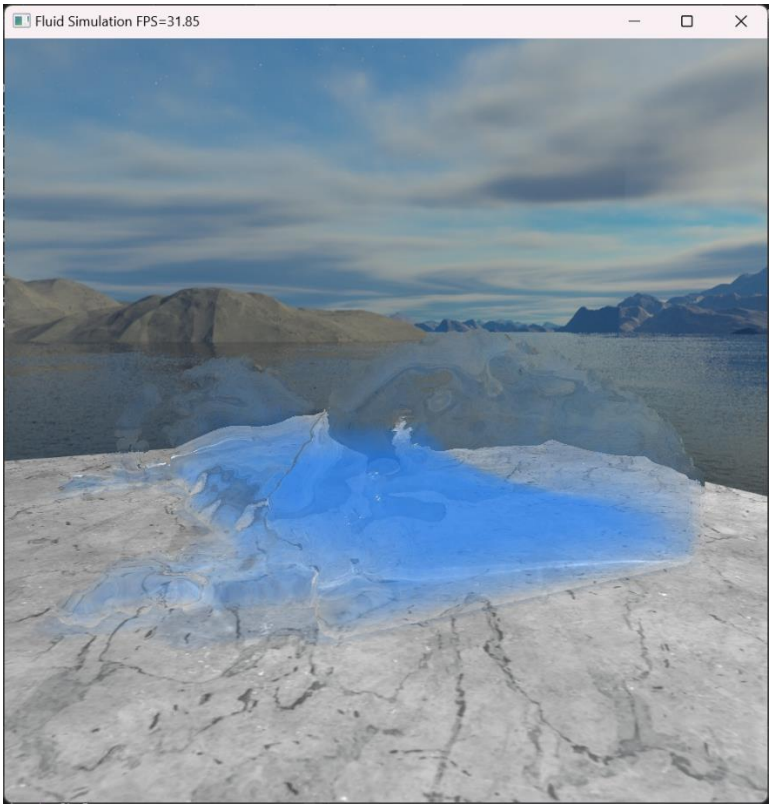
The rendering pipeline of the SSFR algorithm is running correctly. The actual fluid rendering result is as shown in the screenshot below:

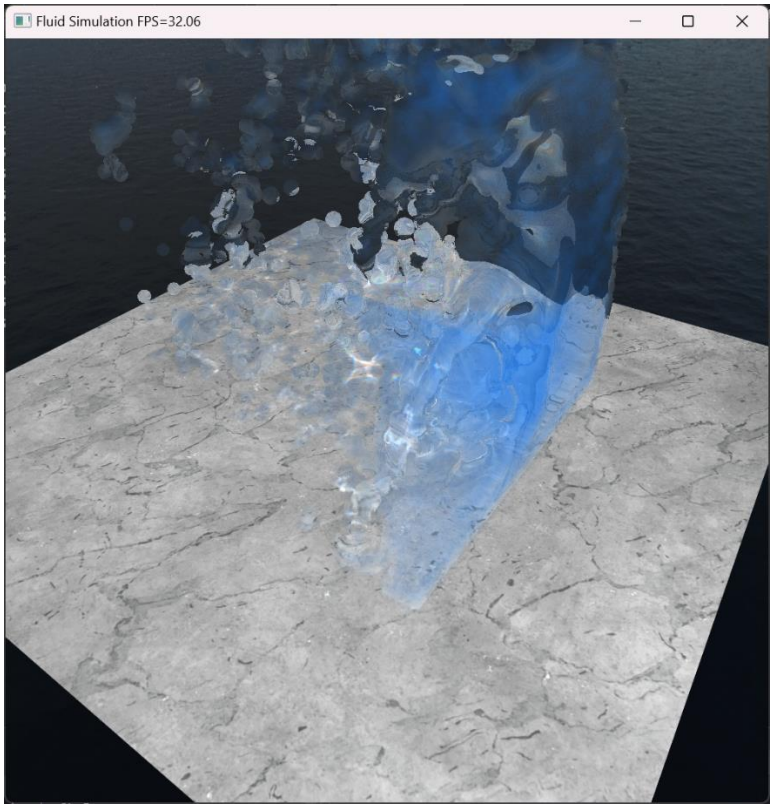



Fluid Simulation FPS=31.68


Fluid Simulation FPS=31.95


Fluid Simulation FPS=31.92


Fluid Simulation FPS=31.85



Fluid rendering effects are realistic. The rendering frame rate is maintained at 30 fps.

2.6.8. Test 8: User interaction

TEST 8	USER INTERACTION
VERIFICATION	Users can interact with the scene through keyboard and mouse operations. Interactive content includes pausing, adjusting the camera, and fluid interaction.
PASS CONDITION	All user interactions run correctly.
FAIL CONDITION	1. Some user interactions run incorrectly. 2. Key binding conflict.

Test 8 result: Pass

Pause: [space]

Exit: [esc]

Camera rotation: [mouse left button click]

Camera displacement: [mouse right button click]

Camera zoom: [mouse wheel]

Drag fluid: [mouse middle button click]

All key presses respond as expected.

2.7. Implications of Implementation

The implementation of this project did not fully follow the plan outlined in the project proposal, and the process of completion was not smooth. This was due to the project's involvement in a significant amount of specialized knowledge in computer graphics. Because I did not have a thorough understanding of the relevant knowledge when initially conceptualizing the project, some parts of the initial ideas were overly simplistic, some parts were difficult to implement, and some aspects were not adequately considered. During the actual project execution, I prioritized extensive learning of computer graphics materials, which consumed a considerable amount of time. Through this learning process, I gradually approached feasible solutions for the project.

Furthermore, by practicing with fast simulation algorithms using Taichi Lang, I deepened my understanding of different simulation algorithms and made judgments on their performance advantages and disadvantages. Finally, during the actual development process of the C++ rendering engine, I selected appropriate algorithms and rendering implementation methods to achieve the desired effects.

2.8. Innovation

This project aims to develop a real-time 3D fluid simulation engine using C++. Its innovation lies in implementing the WCSPH algorithm for fluid physics simulation through GPU parallel computation and utilizing the SSFR algorithm for real-time rendering. The goal is to ensure real-time frame rate requirements while maintaining simulation accuracy and rendering realism.

Compared to common solutions in the market such as the Water Pro component in Unity Engine and standalone water plugins in Unreal Engine, which are shader-based fluid simulation solutions allowing for rapid implementation of basic water simulations without requiring in-depth understanding of specific mathematical and physical computations. These solutions can provide realistic water surface effects including reflections, refractions, and ripples. However, they cannot accurately simulate real physical interactions between fluids and other objects, such as splashes and splatters. Additionally, they come with relatively high performance overheads and limited customization.

This project offers fluid simulation based on the WCSPH algorithm with greater customization capabilities. Developers can freely create water bodies and set desired parameters for simulation using this tool. With real-time rendering capabilities, developers can dynamically showcase rendering effects, quickly assess them, and make adjustments. Moreover, this tool can provide unique solutions for large-scale game projects, particularly in scenarios requiring real-world fluid behaviors for real-time cutscenes or optimizing performance in in-game fluid scenes.

2.9. Complexity

The complexity of this project mainly lies in three aspects: fluid simulation algorithms, fluid rendering algorithms, and the use of GPU frameworks.

Fluid Simulation Algorithms: Fluid simulation is a highly complex problem in the field of physics simulation. In computer graphics, fluid simulation is an important component. Generally, fluid simulation can be divided into particle-based approaches and grid-based approaches. Particle-based approaches have advantages for simulating complex fluid flow patterns, collisions resulting in splashes, and other scenes. Mainstream algorithms include Smoothed Particle Hydrodynamics (SPH) and Position-Based Fluids (PBF), with various branches such as PCISPH and WCSPH. Understanding the specific implementation of these algorithms requires a deep learning and understanding of the fundamentals of fluid simulation theory. Additionally, different algorithmic techniques are still evolving. Performance-improved algorithms or improvements to existing algorithms are relatively new, with limited available resources, requiring review of recent papers for insights. Furthermore, due to the development of GPU hardware, many algorithmic improvements may leverage the advantages of new GPU architectures. Learning and practicing a newer algorithm requires extensive study of various resources.

Fluid Rendering Algorithms: Fluid rendering is an important research area in computer graphics. Since non-grid-based fluid simulations cannot be directly rendered into graphics, specific algorithms are needed to achieve rendering effects. Commonly used algorithms include the Marching Cubes algorithm and the Screen-Space Fluid Rendering (SSFR) algorithm. Learning these algorithms requires a solid understanding of computer graphics fundamentals. Both algorithms also require good implementations of ray tracing, necessitating knowledge of ray tracing in computer graphics. Achieving good rendering effects ultimately requires extensive study of relevant materials.

Use of GPU Frameworks: Complete fluid simulation and rendering involve a large amount of computation. Many complex operations, such as extensive derivatives, perform poorly on CPUs. Achieving real-time simulation and rendering effects requires active consideration of using GPUs for parallel computation. When considering how to effectively utilize GPUs, it is necessary to refer to and study current GPU architectures and API documentation. NVIDIA provides the CUDA development environment, and there are also other graphics APIs such as OpenGL, Vulkan, etc., available for use. To achieve optimal results and understand the different characteristics of various solution choices, it is necessary to consult relevant technical documentation and engage with technical communities for learning.

Overall, this project aims to discuss learning and practical application in the field of fluid simulation and rendering in relatively new areas of computer graphics. The extensive foundational content in computer graphics and the development of new technologies in recent years have posed significant challenges, greatly increasing the complexity of this project.

2.10. Research in New Technologies

During the completion of this project, extensive learning and research in computer graphics were conducted, primarily covering the fields of fluid simulation and rendering, ray tracing, and the use of GPU frameworks.

Fluid Simulation: The research mainly focused on Lagrangian-based fluid simulation algorithms, such as Smoothed Particle Hydrodynamics (SPH) and Position-Based Fluids (PBF). Technical challenges included foundational knowledge of fluid simulation, understanding of algorithm principles, analysis of real-time computation feasibility, algorithmic implementation and optimization methods, etc. Major research

activities included theoretical analysis and coding practice of SPH and PBF algorithms, performance comparison between the two algorithms, and C++ coding for the WCSPH algorithm. Potential areas for further exploration include other optimization implementations of the SPH algorithm, new optimization implementations of the PBF algorithm, etc.

Fluid Rendering: Research was conducted on rendering-related algorithms used to convert particles into continuous surfaces, including Marching Cubes and Screen-Space Fluid Rendering (SSFR) algorithms. Technical challenges included foundational knowledge of fluid rendering, understanding of the principles and specific implementations of the two algorithms, optimization of rendering, and ray tracing issues in rendering. Major research achievements included understanding the principles and specific implementations of SSFR, and lighting calculations for fluids.

GPU Frameworks: Research focused on the utilization of current GPU frameworks, including OpenGL and CUDA. Technical challenges included the use of graphics APIs, deployment and utilization of CUDA, etc. Completed research primarily involved simulating rendering calculations in computer graphics using the OpenGL API on the GPU.

2.11. Future Enhancements

This project has implemented the WCSPH fluid simulation algorithm and the SSFR fluid rendering algorithm using OpenGL, achieving the goal of real-time rendering of 3D fluids. There are several areas where the project can be enhanced:

1. Adaptive kernel sizes for bilateral filtering of the depth map can be implemented, where smaller kernels are used for distant depth points and larger kernels for closer ones.

2. There may be efficiency differences between OpenGL and CUDA. Consideration can be given to switching to CUDA for implementation.

3. Introduce a parameter panel to provide users with a convenient GUI for parameter settings, adjustments, and real-time display of results.

4. Consider implementing the PBF algorithm or other SPH algorithms (such as PCISPH) and compare their performance and effects with those of WCSPH.

2.12. Timeline and Milestones

Timeline	Breakdown Tasks	Duration
Phase 1: Basic Research		
Jan. 8th to Jan. 26th	Study computer graphics courses	60 hours
Jan. 29th to Feb. 2nd	Research PBF algorithm	20 hours
Phase 2: Implement PBF Algorithm		
Feb. 5th to Feb. 9th	Try Taichi Lang architecture	20 hours
Feb. 12th to Feb. 23rd	Implement PBF with Taichi Lang	50 hours

Milestone 1: Implement PBF		
Phase 3: Implement SPH		
Feb. 26th to Mar. 1st	Research SPH algorithm	10 hours
Mar. 4th to Mar. 8th	Implement WCSPH with Taichi Lang	30 hours
Mar. 11th to Mar. 15th	Add objects into scene and adjust interaction	20 hours
Mar. 18th to Mar. 22nd	Add control panel	10 hours
Milestone 2: Implement SPH		
Phase 4: Build C++ engine (WCSPH module)		
Mar. 25th to Mar. 29th	Implement WCSPH with C++	10 hours
	Build GPU compute pipeline	30 hours
	Add material and skybox shader	10 hours
	Build particles renderer	10 hours
Milestone 3: Implement WCSPH module		
Phase 5: Build C++ engine (SSFR module)		
Apr. 1st to Apr. 12th	Research marching cubes and SSFR	30 hours
	Use point sprite algorithm to generate depth/thickness maps	20 hours
	Implement smoothing function	20 hours
	Compute light shading and mix colors	25 hours
	Shadow and caustic mapping generating	15 hours
	General rendering pipeline combine and design	10 hours
	User input handling	5 hours
Milestone 4: Implement C++ rendering engine		

The total completion time of the project is 400 hours.

3. Conclusion

3.1. Lessons Learned

This project is a practical endeavor involving cutting-edge computer graphics. Throughout this project, I've gained hands-on experience and learned a vast amount of foundational knowledge in computer graphics, with a particular focus on fluid simulation and rendering domains. I conducted in-depth research on both the Smoothed Particle Hydrodynamics (SPH) algorithm and the Position-Based Fluids (PBF) algorithm, and implemented them in practice. This allowed me to develop a thorough understanding of fluid simulation algorithms based on the Lagrangian perspective. By implementing these algorithms in actual code, I gained insights into their performance differences and characteristics.

I also explored algorithms such as Marching Cubes and Screen-Space Fluid Rendering (SSFR), with a deep dive into SSFR through practical implementation. By progressively implementing various modules of the

SSFR algorithm, I gained a deeper understanding of areas such as 3D model rendering and ray tracing calculations.

Additionally, I acquired basic knowledge and understanding of the OpenGL API used in the project development process. I learned how to parallelize complex computations using GPUs, thus enhancing my understanding of parallel processing techniques for computer graphics tasks.

3.2. Closing Remarks

With the rapid advancement of GPU hardware and the continuous evolution of computer graphics, the computational capabilities for processing images are becoming increasingly powerful. As a BTech student focusing on game development, there may arise situations in my future professional career where proficiency in computer graphics skills is required. Whether for career advancement or for the development of visual effects in personal game projects, computer graphics is a highly valuable skill set.

Throughout the development process of this project, I have been encouraged to learn and practically apply a wide range of computer graphics knowledge and skills. I have gained a deeper understanding of the subject and experienced firsthand its allure and tremendous potential for the future. I intend to leverage the experience gained from this project to continually challenge and explore the continuously evolving field of computer graphics technology.

4. Appendix

4.1. Approved Proposal

PBF-based Real-Time 3D Fluid Rendering Engine

COMP 8037 – Major Project Proposals

Harley Guan – A00999595
11-6-2023

Table of Contents

1. Student Background

A developer with a rich academic background and extensive practical work experience. Previously employed as a DevOps developer at a technology company, accumulating three years of full-time work experience. Possesses solid expertise in both front-end and back-end development. Demonstrates exceptional mathematical proficiency, achieving high scores in mathematical and physical disciplines at BCIT.

1.1. Education

- South China University of Technology 2011-2016
  - Information Engineering, Bachelor's Degree
- British Columbia Institute of Technology 2017-2018
  - Computer Systems Technology (Cloud Computing Option)
- British Columbia Institute of Technology 2022-now
  - Computer Systems (Games Development Option), Bachelor of Technology

1.2. Work Experience

DevOps Developer
Fortinet
2019-2021

- Designed and maintained automation pipelines for departmental projects, utilizing Jenkins and Docker for seamless code submission, testing, and deployment processes.
- Independently developed and managed the BugScreen project, providing the team and managers with a visual bug submission, tracking, and reporting tool. Front-end development using TypeScript and Angular, backed by Couchbase as the database.
- Collaborated on an internal training game project developed using the Godot engine, assisting in the development of real-time scoreboard components for the web platform using TypeScript.

1.3. Mathematics & Physics Ability

- COMP 1113 Applied Mathematics 91
- COMP 2121 Discrete Mathematics 98
- MATH 7908 Linear Algebra & Applications 100
- COMP 8903 Physics for Games Development 99

2. Project Description

The primary objective of this project is to create an advanced rendering engine designed to simulate the real-time movement of fluids in a 3D environment, adhering to the laws of physics. This rendering engine can be employed in game CGI production and interactive in-game environments.

The project predominantly relies on computer graphics technology to design and develop the rendering engine. The engine is specifically tailored for simulating and rendering 3D fluids based on physical laws, employing the Position-Based Fluids (PBF) algorithm for computational simulation. Under the influence of gravity and other external forces, the fluids are expected to generate real-time visual effects such as splashes and ripples. Fluids are also capable of dynamic interactions with other objects within the

environment, following physics-based motion principles. Additionally, environmental light interacts with the fluid's surface, resulting in realistic optical effects through reflection and refraction, in accordance with physical principles.

While ensuring the fidelity of simulations, optimizations are carried out within the permissible limits of both CPU and GPU capabilities, enabling real-time simulation and rendering at high frame rates. Ultimately, this 3D fluid rendering engine allows for the flexible adjustment of physical parameters such as fluid volume and position, providing an affordable solution for high-quality simulations.

In the end, a real-time rendering demo will utilize this rendering engine to demonstrate the interactive effects between fluids and players or other in-game objects.

3. Problem Statement and Background

With the rapid advancement of Graphics Processing Units (GPUs), high-tech fields such as the gaming industry, artificial intelligence, and data science are continuously innovating and breaking boundaries. The increasing power of GPUs has brought about robust real-time computational speed and graphic processing capabilities. This development allows various sophisticated rendering techniques to gradually achieve real-time computation, enhancing the visual experience in games.

Fluid simulation has always been a demanding area concerning rendering computations. In non-real-time rendering fields such as film and TV CGI production, different fluid simulation algorithms are utilized for various scenes. These include grid-based Eulerian methods, which use Navier-Stokes equations to track fluid properties, as well as particle-based methods like Smoothed Particle Hydrodynamics (SPH), which represent fluids as interacting particles and simulate liquids through these interactions. Of course, there are many other excellent simulation and rendering algorithms, but these algorithms face limitations when applied in real-time computational scenarios on gaming platforms. Due to the frame rate demands of real-time rendering computations, both grid-based and particle-based algorithms have to compromise by reducing the total number of grids or particles to balance computational capabilities and visual effects.

To achieve high-quality real-time fluid simulation and rendering effects affordable on contemporary GPUs, I plan to utilize the Position-Based Fluids (PBF) algorithm to construct a fluid simulation engine. PBF is also a particle-based modeling and simulation algorithm. It calculates the positions and velocities of particles based on their current positions, enforcing physical constraints derived from the Navier-Stokes equations. Compared to another particle-based SPH algorithm, PBF offers higher stability and controllability, making it easier to perform parallel computations. Additionally, PBF excels in handling free-surface fluid effects, providing excellent simulation and rendering effects for common gaming scenes like waves, raindrops, and splashes. PBF's drawback is its limited applicability to other fluid types, such as gases and powders, and its precision is not as high as SPH. However, it saves more computational resources than SPH typically requires, thus reducing performance overhead. In real-time gaming scenarios, striking a balance between simulation quality and performance overhead is achievable by using the PBF algorithm.

This project will also leverage graphics APIs such as Vulkan or DirectX 12 to implement ray tracing for lighting effects. DirectX 12, developed by Microsoft, and Vulkan, a cross-platform graphics API developed by the Khronos Group, both provide low-level programming interfaces, allowing developers better control over graphics and computational resources. They offer excellent multithreaded processing capabilities, high performance, and low overhead. With their support, realistic lighting effects and higher computational performance can be achieved for fluid rendering.

4. Scope and Depth

This project aims to create a real-time 3D fluid simulation rendering engine from scratch. Its scope should encompass the following fundamental implementations:

- Development of a Standalone Real-time Rendering Engine in C++: Create an independent real-time rendering engine capable of adding 3D objects, configuring light fields, and generating real-time rendered images.
- Physical Simulation and Rendering of Fluids Using PBF (Position-Based Fluids) Algorithm: Implement fluid physics simulation and rendering based on the PBF algorithm. Real-time calculations should emulate physical collision effects, achieving effects such as splashes and ripples.
- Integration of Ray Tracing Functionality Using DirectX 12 and Vulkan Graphics APIs: Utilize DirectX 12 and Vulkan graphics APIs to incorporate ray tracing capabilities into the real-time rendering process.

Depending on the learning progress and project completion status, this project may also include the following extensible scope:

- Hardware Optimization According to Existing Hardware Configurations: Optimize the rendering engine to deliver high-quality rendering images in real-time with stable frame rates based on various hardware devices.
- Expansion of Fluid Simulation Scope or Addition of Interactive Fluid Objects: Extend the scope of fluid simulation to include various types of fluids (e.g., quicksand, mud) or add interactive fluid objects like players and other non-rigid fluids.

The primary goal of this project is to develop a fluid rendering engine itself, and to study and practice computer graphics. The scope does not cover:

- The actual production of related games.
- Machine learning or other non-computer-graphics-based optimization methods.

The depth of this project is reflected in the following aspects:

- In-depth Knowledge of Computer Graphics: This project requires in-depth study and research in computer graphics, including understanding and implementing real-time graphics rendering. Additionally, it involves learning and utilizing graphics APIs such as DirectX 12 and Vulkan.
- Mathematics and Physics Proficiency: Proficiency in understanding and applying the PBF algorithm, including knowledge in fluid dynamics, advanced mathematics, and 3D mathematics, is essential for this project.

- Proficiency in C++ Programming and Architecture: The project demands proficiency in C++ programming and utilization of graphics programming APIs. The rendering engine should be implemented using a well-designed programming architecture.
- In-depth Understanding of Hardware: A comprehensive understanding of mainstream hardware, particularly GPUs, is necessary. Deep knowledge of different hardware architectures and optimization methods is also vital for the project's success.

5. Test Plan

The testing plan includes:

1. Functional Testing:
   - Real-time rendering output functionality testing for 3D objects and light fields.
     - Pass: 3D objects should appear with correct shapes, order, and lighting effects.
     - Fail: 3D objects have incorrect shapes or order from different angles, incorrect occlusion relationships, or lighting and shadow effects are displayed incorrectly.
   - Basic fluid simulation functionality testing based on the PBF algorithm.
     - Pass: Fluid simulation operates correctly based on the PBF algorithm and conform to Navier-Stokes equations in any situation.
     - Fail: Fluid simulation behavior deviates from PBF algorithm standards in some situations.
   - Functional testing of fundamental physical properties of fluids such as flow and diffusion.
     - Pass: Fundamental fluid properties like flow and diffusion are accurately simulated.
     - Fail: Deviations in simulating fundamental fluid properties.
   - Interaction testing between fluids and rigid bodies.
     - Pass: Fluids detect collisions with rigid objects and do not pass through them.
     - Fail: Fluids pass through rigid objects or fail to detect collisions accurately.
   - Testing of advanced fluid simulation effects such as splashes and ripples.
     - Pass: Advanced effects like splashes and ripples are simulated accurately.
     - Fail: Deviations in simulating advanced fluid effects.
   - Ray tracing functionality testing.
     - Pass: Ray tracing functions correctly, providing accurate rendering results.
     - Fail: Ray tracing results in inaccuracies or rendering errors.
2. Performance Testing:
   - Frame rate testing, including real-time rendering performance at different hardware configurations and varying particle counts.
     - Pass: Stable frame rates achieved across various hardware configurations and particle counts. Frame rate should be kept at 60fps or above most of the time.
     - Fail: Inconsistent or unstable frame rates under different. Frame rate is below 30fps with appropriate hardware configuration and parameter settings.
   - Memory and resource usage testing, examining memory consumption in different scenarios and ensuring stability.
     - Pass: Memory and resource usage is within acceptable limits, ensuring system stability.

- Fail: Excessive memory consumption or resource usage leading to instability. Long running may cause crash due to overflow.

3. Compatibility Testing:
   - Testing the engine's performance and compatibility on different hardware configurations, including various GPU and CPU models.
     - Pass: Engine performs well and is compatible with different GPU and CPU mainstream models on the market.
     - Fail: Performance issues or incompatibilities observed on specific GPU or CPU models.

For specific testing scenarios, the project may utilize the following testing tools or frameworks:

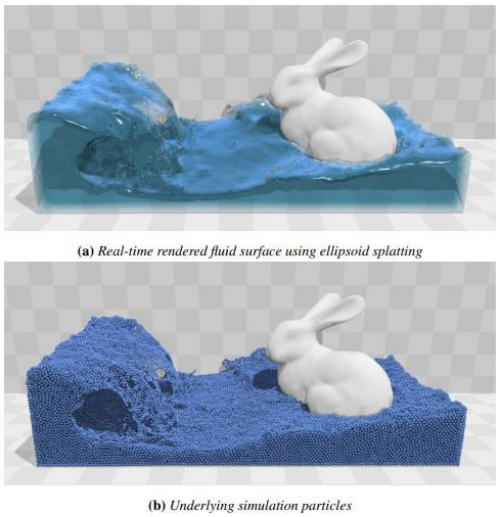
1. Google Test: A unit testing framework for writing C++ unit tests.
2. 3DMark: Used for testing the rendering performance of the engine.
3. RenderDoc: Utilized for testing GPU performance and graphic rendering.

These testing tools and frameworks will be employed to implement the detailed testing scenarios mentioned above.

6. Methodology

The project will utilize a waterfall development approach combined with a modular development methodology. For clearly defined objectives, I will proceed with the development and testing of individual modules, integrating the implemented modules into the project's main body one by one. After completing the initial version using the waterfall development approach, there might be a transition to an iterative development strategy based on the project's needs. Depending on the current version's status, improvements and additional features will be implemented according to the requirements. With each iteration reaching a stable version, a node with a version tag will be released on GitHub.

In this project, 3D fluid simulation is achieved using the Position-Based Fluids (PBF) algorithm. The PBF algorithm is a particle-based computational fluid dynamics method used to simulate fluid behavior. It involves generating and rendering a large number of spherical particle models. The algorithm calculates the movement speed attributes of each particle model based on its position and the pressure gradient around it. This real-time calculation and simulation enable the representation of fluid motion formed by a multitude of particle models.



(a) *Real-time rendered fluid surface using ellipsoid splatting*

(b) *Underlying simulation particles*

Figure 1: Bunny taking a bath simulation by PBF with 128k particles.

The computations are primarily based on the following formulas:

$$\nabla_{p_k} C_i = \frac{1}{\rho_0} \begin{cases} \sum_j \nabla_{p_k} W(p_i - p_j, h) & if\ k = i \\ -\nabla_{p_k} W(p_i - p_j, h) & if\ k = j \end{cases}$$

The rendering engine implementation will utilize OpenGL and Bullet3D to establish fundamental 3D object rendering capabilities and physics simulation. Advanced rendering techniques and ray tracing-based rendering will be achieved by invoking DirectX 12 or Vulkan.

System/Software Architecture Diagram

The system architecture diagram is shown in the following figure.

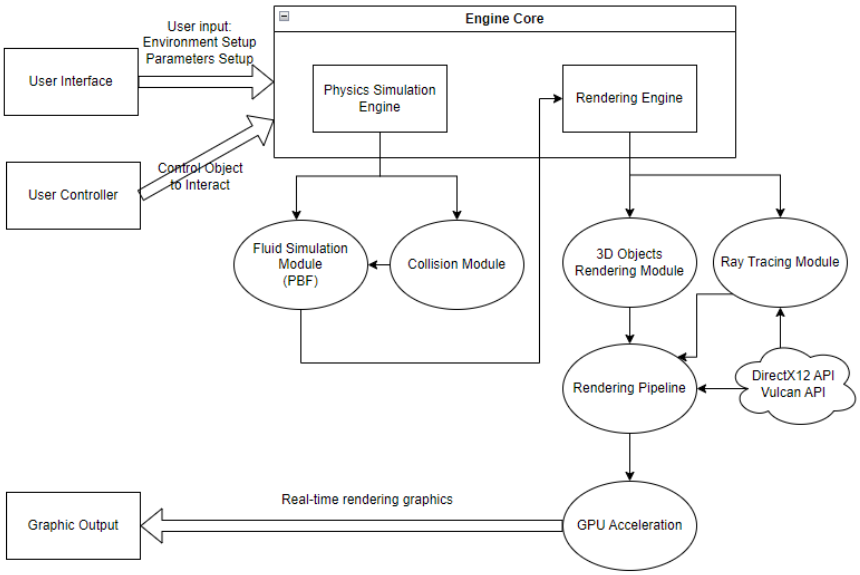


Figure 2: System architecture diagram.

The core of the entire rendering engine comprises two main components: the physics simulation engine and the rendering engine. The physics simulation engine consists of the fluid simulation module, which applies the PBF algorithm, and the collision module, responsible for detecting and computing interactions with objects. The output from the collision module is utilized by the fluid simulation module to calculate the fluid's subsequent motion trends. The updated position information is then transmitted to the rendering engine for unified rendering processing.

Within the rendering engine, the 3D objects rendering module and the ray tracing module operate independently, and their results are harmoniously integrated into the rendering pipeline. Following processing through the rendering pipeline and GPU acceleration, all object information is rendered into images, which are then output to the screen.

7. Innovation

The objective of this project is to develop a widely applicable real-time 3D fluid simulation rendering engine. The innovation of this engine lies in its use of the Position-Based Fluids (PBF) algorithm for GPU-

accelerated, high-quality real-time fluid simulation and rendering, coupled with robust customization capabilities. The project aims to create a fluid rendering engine that is controllable and adjustable, capable of achieving optimal performance based on varying computational power. Additionally, this project will incorporate ray tracing technology to ensure lifelike representations of liquids within a realistic lighting environment.

In comparison to common solutions available in the market, such as Unity Engine's Water Pro component and Unreal Engine's standalone water plugins, which are shader-based fluid simulation solutions, they enable rapid implementation of basic water simulations without the need for in-depth knowledge of specific mathematical and physical computations. These solutions can provide realistic water surface effects, including reflections, refractions, and ripples. However, they fall short of accurately simulating the realistic physical interactions of fluids with other objects, such as splashes and splatters. Furthermore, they come with relatively high-performance overheads, and their level of customization is limited.

This project opts for the utilization of the PBF algorithm, integrating it into a real-time fluid rendering engine to offer higher physical accuracy and greater customization flexibility. This approach provides a unique solution for large-scale gaming projects, especially in scenarios where simulating real-world fluid behaviors for real-time cutscenes or adjusting in-game fluid scenes' performance is necessary. Through the application of the PBF algorithm, this project can achieve highly realistic fluid simulations, opening up new possibilities for real-time rendering scenes, such as in gaming environments.

8. Complexity

The complexity of this project is manifested in the application and simulation of the PBF algorithm. Real-time computation of particle interactions based on the PBF algorithm forms the foundation of fluid simulation. Fluids, in addition to their inherent interactive forces and surface tension, also require collision force calculations based on the surrounding physical environment to simulate motion in accordance with the laws of physics. Fluid motion representation should encompass phenomena like waves, splashes, sprays, and eddies.

Another aspect of the project's complexity lies in the application of ray tracing technology. Real-time computation of fluid visual representation within the light field is achieved through ray tracing. This includes phenomena such as reflection and diffuse reflection on the fluid surface, as well as refraction through the fluid. Objects inside and above the fluid should exhibit good optical shading within the fluid.

The final complexity of the project is in constructing a real-time rendering engine that incorporates the above-mentioned technologies. This engine should allow the free adjustment of simulation parameters and quality, enabling developers to balance computational expenses and the final visual quality based on practical usage scenarios.

9. Technical Challenges

The primary technical challenges of this project lie within the field of computer graphics. Both fluid simulation computations and ray tracing technology applied in this project are applications of computer graphics. Proficiency in advanced mathematics, 3D mathematics, computational geometry, and physics are essential prerequisites in this field. A fundamental understanding of image processing and rendering techniques in computer graphics is also required, which demands self-study and research within the realm of computer graphics.

Moreover, mastering algorithms for particle-based simulations is crucial. The project primarily utilizes the Position Based Fluids (PBF) algorithm to strike a balance between computational efficiency and realism. Understanding the core computational concepts from relevant papers and implementing them in the codebase is necessary.

Furthermore, the technical challenges involve learning and applying graphics APIs such as Vulkan and DirectX12. By studying and using the latest versions of graphics APIs, one can enhance their understanding of computer graphics and apply ray tracing technology to this project.

Lastly, optimizing performance for existing hardware poses another technical challenge. Debugging and optimization efforts will be guided by NVIDIA and AMD's GPU developer manuals, aiming to achieve GPU acceleration and enhance overall performance.

10. Development Schedule and Milestones

Tasks	Estimated Hours
Phase 1: Preparation Phase	
Learn and understand the basics of computer graphics and the principles and requirements of real-time rendering engines	25
In-depth study of PBF algorithm and fluid simulation calculation formula	10
Phase 2: Set-up Phase	
Create project (Visual Studio 2022, C++ based) and development pipeline (GitHub)	1
Import libraries (Bullet, OpenGL, etc)	1
Phase 3: Object Rendering Module	
Make simple 3D objects (cube, cylinder, sphere) generator and editor	8
Build the original rendering function	10
Build 3rd party 3D object import function	10
Controller functions for interactive 3D objects	5
Rendering engine parameter setter and its UI	10
Milestone 1: Initial 3D real-time rendering engine construction	80 in total
Phase 4: Fluid Simulation Module	
Fluid (mass particle) object generator and editor	15
Adjust static fluid rendering performance	5
Apply PBF algorithm to fluid particles	20

Adjust dynamic fluid rendering performance	10
Phase 5: Fluid Collision Module	
Collision detection function between fluid particles and rigid body attribute objects	5
Application of PBF algorithm to calculate the motion of fluid particles	20
Create a pool object to adjust the dynamic rendering performance of fluid in the pool	20
Parameter setter for fluid objects	5
Milestone 2: Construction of fluid 3D real-time rendering method	180 in total
Phase 6: Light simulation Module	
Learn and research the light simulation implementation of DirectX12 and Vulcan API	10
Light object generator and editor	20
Apply ray tracing method	20
Editing methods for applying reflection and refraction coefficients to fluid objects	5
Adjust the overall rendering performance after applying lights	20
Parameter setter for light objects	5
Milestone 3: Construction of real-time fluid rendering method with light source	260 in total
Phase 7: Hardware Optimization (Optional)	
Test rendering efficiency and quality under different hardware conditions	5
Adjust the renderer according to hardware conditions and provide parameter recommendation functions	20
Phase 8: Fluid Scope Expansion (Optional)	
Try to implement fluid simulation with different properties	30
Extended fluid parameter setter	10
Phase 9: Test Phase	
Comprehensive testing: different hardware configurations, different parameter selections	5
Optimize based on test results	40
Phase 10: Report Phase	
Complete project report	10
Milestone 4: Final delivery	380 in total

The estimated hour for this project is 380 hours in total.

According to the project scope (details in section 4), phase 7 and phase 8 are optional based on the completion time of Milestone 3.

11. Deliverables

Source Code

The source code of the project should be version-controlled and stored on GitHub for archival and publication purposes.

Demo Application

The project should generate an executable program suitable for demonstration purposes.

Project Report

The project should undergo continuous updates and culminate in the completion of the final report.

12. Conclusion and Expertise Development

This project involves the practical implementation of the PBF algorithm into a real-time 3D fluid simulation rendering engine. Additionally, it integrates ray tracing functionality using DirectX12 and Vulkan APIs into the rendering engine. Through the development and implementation of this project, I have gained valuable insights and mastery over the fundamentals and applications of computer graphics. I have also acquired practical experience and a deeper understanding of graphics APIs such as DirectX12 and Vulkan. Moreover, this project has provided me with the opportunity to explore the rendering technologies of mainstream GPUs.

Despite my strong foundation in linear algebra and 3D mathematics, I hadn't had the chance to apply this knowledge in practical development scenarios before. Studying the application of the PBF algorithm in simulating and rendering fluids on computers has enhanced my understanding of computer graphics. It has also allowed me to accumulate experience for future development tasks involving low-level rendering functionalities. The experience gained from this project will undoubtedly prove invaluable in potential future roles related to game engine development, renderer development, and graphics engine development.

13. References

- Miles Macklin and Matthias Müller. (2013). *Position based fluids.* ACM Trans. Graph. 32, 4, Article 104 (July 2013), 12 pages. https://doi.org/10.1145/2461912.2461984

14. Change Log

- [List all changes made from previous versions, referencing the page numbers where changes were made. Delete this section if not applicable.]

4.2. Project Supervisor Approvals

Harley Guan project

BN

Borna Noureddin <borna_noureddin@bcit.ca>

收件人: BCIT BScACS

周四 2024/4/18 16:09

抄送: 你; BCIT CST BTech Projects

Kwok project

Dear Michal,

Harley Guan has successfully demonstrated the proposed project to me. I am satisfied that everything in the approved proposal has been completed. I have also reviewed the report and recommend that it be submitted to the Committee for review.

I also think Harley's project and report is excellent, and would recommend it be used as a sample report for future use, if it is approved and he agrees.

Thank you,

Borna Noureddin, PhD
Faculty, Computing
British Columbia Institute of Technology | Computer Systems Technology
3700 Willingdon Ave | Burnaby, BC V5G 3H2
cst_btech_projects@bcit.ca

BCIT

5. References

- Miles Macklin and Matthias Müller. (2013). Position based fluids. ACM Trans. Graph. 32, 4, Article 104 (July 2013), 12 pages. https://doi.org/10.1145/2461912.2461984

- R. A. Gingold, J. J. Monaghan, Smoothed particle hydrodynamics: theory and application to non-spherical stars, Monthly Notices of the Royal Astronomical Society, Volume 181, Issue 3, December 1977, Pages 375–389, https://doi.org/10.1093/mnras/181.3.375
- Green, S. (n.d.). GDC 2003. NVIDIA Developer. https://developer.nvidia.com/gdc-2003
- Wyman, C., & Davls, S. (n.d.). Interactive image-space techniques for approximating caustics. http://cwyman.org/papers/i3d06_imgSpaceCaustics.pdf
- Akinci, N., Dippel, A., Akinci, G., & Teschner, M. (1970, January 1). Screen space foam rendering. DSpace at University of West Bohemia: NO TITLE. https://otik.uk.zcu.cz/handle/11025/6982
- Biedert, T., Sohns, J. T., Schröder, S., Amstutz, J., Wald, I., & Garth, C. (2018, June). Direct Raytracing of Particle-based Fluid Surfaces Using Anisotropic Kernels. In EGPGV@ EuroVis (pp. 1-11).